SAPIENZA
UNIVERSITÀ DI ROMA

Department of Computer, Control and Management Engineering "Antonio Ruberti" (DIAG)
Faculty of Information Engineering, Informatics, and Statistics (I3S)
Sapienza University of Rome

DISSERTATION

Cooperative Scheduling and Load Balancing techniques in Fog and Edge Computing

Gabriele Proietti Mattia

This thesis has been submitted in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy (Ph.D.) in Engineering in Computer Science at Sapienza University of Rome

A.Y. 2021/2022 · Cycle XXXV

Advisor
Prof. Roberto Beraldi

Co-Advisor
Prof. Andrea Vitaletti

PhD Program Coordinator
Prof. Luca Iocchi

pm.gabriele <at> gmail.com · gpm.name

Revision on Thu, 26 Jan 2023 09:21:02 +0000 (4388abe)

The present thesis was submitted for peer review on October 31, 2022, and the final version was submitted on January 05, 2023.

The selected external reviewers were: Prof. **Ravi Prakash** from the Department of Computer Science, University of Texas at Dallas and Prof. **Roy Friedmann** from the Department of Computer Science Technion, Israel Institute of Technology.

The thesis has been defended on January 24, 2023 in the presence of the following commission: Prof. **Alessio Merlo** from Università di Genova, Prof. **Emanuele Menegatti** from Università di Padova and Prof. **Pierluigi Plebani** from Politecnico di Milano.

To my family

Abstract

Fog and Edge Computing are two models that reached maturity in the last decade. Today, they are two solid concepts and plenty of literature tried to develop them. Also corroborated by the development of technologies, like for example 5G, they can now be considered *de facto* standards when building low and ultra-low latency applications, privacy-oriented solutions, industry 4.0 and smart city infrastructures. The common trait of Fog and Edge computing environments regards their inherent distributed and heterogeneous nature where the multiple (Fog or Edge) nodes are able to interact with each other with the essential purpose of pre-processing data gathered by the uncountable number of sensors to which they are connected to, even by running significant ML models and relying upon specific processors (TPU). However, nodes are often placed in a geographic domain, like a smart city, and the dynamic of the traffic during the day may cause some nodes to be overwhelmed by requests while others instead may become completely idle. To achieve the optimal usage of the system and also to guarantee the best possible QoS across all the users connected to the Fog or Edge nodes, the need to design load balancing and scheduling algorithms arises. In particular, a reasonable solution is to enable nodes to cooperate. This capability represents the main objective of this thesis, which is the design of fully distributed algorithms and solutions whose purpose is the one of balancing the load across all the nodes, also by following, if possible, QoS requirements in terms of latency or imposing constraints in terms of power consumption when the nodes are powered by green energy sources. Unfortunately, when a central orchestrator is missing, a crucial element which makes the design of such algorithms difficult is that nodes need to know the state of the others in order to make the best possible scheduling decision. However, it is not possible to retrieve the state without introducing further latency during the service of the request. Furthermore, the retrieved information about the state is always old, and as a consequence, the decision is always relying on imprecise data. In this thesis, the problem is circumvented in two main ways. The first one considers randomised algorithms which avoid probing all of the neighbour nodes in favour of at maximum two nodes picked at random. This is proven to bring an exponential improvement in performance with respect to the probe of a single node. The second approach, instead, considers Reinforcement Learning as a technique for inferring the state of the other nodes thanks to the reward received by the agents when requests are forwarded.

Moreover, the thesis will also focus on the energy aspect of the Edge devices. In particular, will be analysed a scenario of Green Edge Computing, where devices are powered only by Photovoltaic Panels and a scenario of mobile offloading targeting ML image inference applications.

Lastly, a final glance will be given at a series of infrastructural studies, which will give the foundations for implementing the proposed algorithms on real devices, in particular, Single Board Computers (SBCs). There will be presented a structural scheme of a testbed of Raspberry Pi boards, and a fully-fledged framework called "P2PFaaS" which allows the implementation of load balancing and scheduling algorithms based on the Function-as-a-Service (FaaS) paradigm.

Keywords Fog Computing, Edge Computing, Load Balancing, Distributed Scheduling, Randomized Algorithms, Reinforcement Learning, offloading Strategies, Energy Footprint, Testbeds, Software Frameworks

Contents

Chapter 1

Introduction

Necessity is the mother of invention.

Proverb

Moving the computation among different machines is a technique that was born with the birth of the first computer network. Indeed, the simple idea of connecting two computers creates the possibility of making them work together to perform more computations in the same period of time. The first attempt of remotely executing the computation is described in an RFC dated back in 1971 with the term *Remote Job Entry*[1]. The purpose of the protocol was to forward tasks to mainframe computers and retrieve the result in the machine that issued the request, and it was usually used by IBM mainframes. Today, the idea of forwarding tasks to more powerful computers is the general idea that is behind the well-known Cloud Computing concept, a term that was introduced in the 1990s. From there, the beginning of the Millenium was dominated by the birth of the major Cloud provider, such as AWS in 2002, Google Cloud in 2008, Microsoft Azure in 2010. The NIST standardised the term "Cloud Computing", presenting the well-known cloud services types (SaaS, PaaS and IaaS) in 2011 [1]. The flourishing of these services has certainly been promoted by three essential factors: the progressive increase of the computational power of processors and machines, the increase in the number of computers and, most of all, the spread of the internet connection.

During the years, in particular, in the last decade, the progress in the technology for hardware manufacturing made computing devices smaller and smaller, and more powerful at the same time. The first phenomenon to which we have assisted regards the data centres. Indeed, the major companies increased the number of available physical locations for their servers and to which developers can

[1] https://www.rfc-editor.org/rfc/rfc105

deploy the applications. We now wonder which is the core driver of this behaviour. The main reason is that the QoS aspect of a web application is governed by different parameters: the uptime of the servers, the bandwidth that the servers are able to manage and the latency that is required for a packet to reach the server, also called round-trip time[2] (RTT). While the uptime and the bandwidth can be addressed by improving redundancy or the hardware itself, the round-trip time can be reduced only in one way: the server must change its physical location. This is because the RTT is determined by the so-called *network distance* between the user and the server itself. The higher the number of hops that the packet needs to be subjected to, the highest the latency. Indeed, for an application that must be used in Italy, it would be counterproductive to deploy its code on a server that is placed on the west coast of the USA. Providers indeed started, and also continue nowadays, to spread the distribution of their services across the whole planet. Today, it is easy to book an AWS server in Milan and Rome for example, a thing that would have been impossible a few years ago.

Communication latency is, therefore, a critical aspect of web applications. We have also to consider that for the Cloud there is a lower bound on the RTT that is in the order of 20-30ms. However, some delay-sensitive applications may require even lower values. We can think, for example, about shared virtual or augmented reality experiences, industry 4.0 or even health-based applications. Offering a remote-rendered virtual world requires at least 60fps, that is 13.3ms needed for rendering a frame. In order to drastically reduce the latency, up to even 3ms or 4ms, we undoubtedly need to move the servers. In particular, they have to be moved as near as possible to the user. This new paradigm of computation gave birth to the Fog Computing [2] term, introduced by Cisco in 2012. Fog Computing aims to place the servers near to the users, for example, in the same place where the company resides, which also preserves the privacy of the data, or even in proximity or within the 4G or 5G antennas. In this case, we also refer to Edge Computing, a term that was even known before the introduction of Fog Computing. The methods and the algorithms shown in this thesis can be applied in both environments, and for this reason, both terms will be used. However, we will use Fog Computing for targeting Smart Cities and essentially scenarios in which location awareness and the geographic approach are more evident. Instead, we will use the term Edge Computing when smaller devices can be involved, like for example, Single Board Computers (SBCs) as the Raspberry Pi[3].

Fog and Edge Computing Characteristics To have clear the contributions of the thesis, we need to recap which are the main traits of Fog and Edge Computing. This will introduce the scenario for which the proposed methods are designed and can be applied. Fog Computing (or Edge Computing) is a computing model based on the distribution computation among nodes called Fog Nodes (or Edge Nodes). According to NIST [2], the essential traits of the Fog Computing models are the following.

1. **Contextual location awareness**. The nodes in the system are aware of their contextual location, in this way, they can reduce the processing latency.
2. **Geographical Distribution**. Nodes are usually spread in a geographic domain, for example, the typical one is a smart city.

[2]This latency is the one that can be easily measured with the command `ping`.
[3]`https://www.raspberrypi.org/`

3. **Heterogeneity**. Nodes involved in the computing model can have different characteristics in terms of performance, moreover, they can collect and process data of multiple kinds and sources.
4. **Interoperability and federation**. Fog computing components, especially nodes, are able to cooperate, even if from different providers. This is envisioned in order to offer support to low-latency applications. In this way, the services must be federated.
5. **Real-time interactions**. The Fog computing interaction with the clients is real-time rather than batch processing, which is instead typical of Cloud Computing.
6. **Scalability and agility of federated, Fog node clusters**. The nature of Fog computing is typically adaptive and can scale according to the required or available resources.

All of these characteristics guided the design of the methods and algorithms proposed in this thesis. As anticipated, for Edge Computing we assume that the same characteristics hold, but the devices involved may also be small and increase in number.

The Cooperation The leitmotif of the thesis regards a peculiar capability of nodes involved in Fog and Edge Computing environments: they can cooperate. Cooperation is based on the fact that each node can benefit from the possibility of another node executing part of its assigned tasks. This strategy fits particularly well with the environments due to the heterogeneity of the devices and the geographical distribution of the nodes, a node may find itself overwhelmed by clients' requests. Indeed, considering, for example, a smart city, the people distribution in different areas of the city varies over the day. Some areas can be characterised by heavy load conditions on the nodes, and these nodes could simply forward part of their task to other nodes that instead are less loaded and even completely unloaded.

The Task Model The usual model that we address in this thesis and that is recurrent across all the subsections is an online request model. As we have seen when discussing the Fog and Edge Computing characteristics, the interaction model that clients have towards these computing layers is no more batch processing. Batch processing indeed is a typical Cloud service model where a lot of data must be processed by data centres, and the result is not strictly needed *hic et nunc*. But when these data centres are small and can be spread across a geographic domain, then the clients (or generally, the users) can request a service just to the nearest available node. This paradigm of usage is typically called stream processing with respect to the batch, and in this thesis, that is the only focus. Indeed, we assume clients that continuously generate requests of execution of a service to a given (Fog or Edge) node, then, for each request, the node itself has to decide where the request has to be processed: if locally, namely within the same node to which the request has been made, to another neighbour node or even to the Cloud. In the last two cases, the request itself with its annexed payload, if any, will be forwarded.

The "State" Problem We can agree with the fact that enabling cooperation concretises with a decision problem. Indeed, a node has to decide where to execute the tasks received by the clients. At this point, we wonder how this decision can be taken and which conditions a node has to take into account to make a good decision. The crucial information that comes into play is the *state* of the node. At a very high level, a node can forward a request to another node if the state of the other node is better

than its own. This is a key fact because, only by doing this, we are levelling the goodness of the state, at least ideally. The state must be a parameter or a composition of parameters that reveal all what we need to know about a node. In practice, in the works that are going to be presented in this thesis, different strategies for modelling it have been followed: the state has been fixed as the number of tasks that a node is executing in a given time t, or even better, it has been considered as the effective request rate (in terms of requests per seconds - req/s) that a node has to execute. The modelling of the state essentially follows the type of modelling approach that has been chosen, if probabilistic (Chapter 2), Reinforcement Learning based (Chapter 4) or based on dynamical systems (Chapter 3). In general, we can assume that a node, in order to make the correct scheduling decision, it needs to know the state of the other nodes. This is the core problem that the thesis tries to solve. How can nodes make decisions if they do not know the state of the others? The scenario in which these nodes live is fully distributed, and therefore there is no central entity which holds the state of every node, but even in that case, the copy of the state that this hypothetical node always has a delay to be updated. The general assumption that we make in this thesis is the following. Suppose that node A requires the state of another node B at time t_0. Then node A will receive the answer from B at time t_1, and we call

$$\tau = t_1 - t_0 \tag{1.1}$$

Assumption 1 (Lack of the state)**.** *In real systems, $\tau > 0$. In other words, no node can know the state of another node without any delay.*

Indeed, for any given node, requesting the state of another node introduces delay, and the information is always "old" by τ units of time.

Motivation The motivation is now clear. The objective is to find a way of scheduling tasks in a fully distributed manner but taking into account that the state is very precious information which cannot be known if not explicitly asked for. Moreover, the act of retrieving the state introduces a delay making the same retrieved information always "old". Unfortunately, the truth is that for making the optimal decision, the state must be known, and therefore we can conclude that if Assumption 1 holds, then every node will always base its scheduling decision on imprecise information. The principal focus of this thesis is finding and studying methods and algorithms that are able to make decisions as much as possible closer to the optimal ones that could only be taken if an oracle would able to tell us the exact state of any other node with no delay. The main strategies studied in this thesis for designing this kind of solution are the following (in order of available information):

- not asking the state at all and forwarding tasks randomly and blindly;
- asking the state to random nodes and forward tasks only if a node with a better state is found;
- inferring the state of other nodes exploiting Reinforcement Learning strategies;
- asking the state to all the neighbour nodes and forward tasks only if a node with a better state is found;

A general rule of thumb is that the more state information is retrieved, the more the introduced latency and the more the information may be old. Even if the state probing is performed concurrently,

a slight increase in the latency may depend on the network traffic, which is serialised by the network adapter. Therefore, a good trade-off must be achieved in order to reach the best performance.

1.1 Research Contributions

We can summarise all the contributions that will be illustrated in this thesis. Each contribution will refer to a particular section, and they can be divided into three different categories.

Distributed algorithms for load balancing (and task offloading) Research contributions in this area regard the design of algorithms whose aim is managing the traffic in such a way the load across all of the nodes is balanced. In this thesis, we only focused on fully distributed and cooperative algorithms.

- in Section 2.2 the *power-of-n* choices is adapted for load balancing in the Fog Computing environment, in particular, the work (i) defines the $LL(F, T)$, Least Loaded among F nodes with threshold T, a threshold-based load balancing algorithm, and then shows (ii) a mathematical analysis of the same algorithm showing how $LL(1, K - 2)$, i.e., probing just one node when one or two servers out of the K are idle, reduces up to one order of magnitude the control and delay overhead with respect to a vanilla randomised load balancing implementation of the power-of-two random choices algorithm;
- in Section 2.3 the *power-of-n* choices is again exploited for (i) designing a multi-hop algorithm suitable for scenarios characterised by independent providers and heterogeneous load conditions, along with a variant based on a self-tuning mechanism, then (ii) a mathematical analysis and experimental evaluation of the algorithms is done on a realistic scenario, showing evidence of significant improvements compared to unbalanced nodes;
- in Section 2.4 a study of the impact of schedule lag, η, on the performance of *load-aware* balancing protocols based on randomisation is presented. The work studies how and to which extent making a decision based on stale information concerning the load state of the nodes weakens the effectiveness of the algorithm and how load balancing can be achieved when this delay in communicating state information is unavoidable. The work finds that it exists a "critical" value of η starting from which load information becomes meaningless, and a simpler blind forwarding algorithm performs better;
- in Section 3.2 a latency-levelling algorithm specifically designed for Fog and Edge Computing is designed, in particular, the section present: (i) a continuous-time model which describes the dynamics of the system by using a system of differential equations that reaches the stability when all the nodes experience the same service latency; (ii) a heuristic algorithm which tries to find a solution to the problem in a real environment by continuously adapting the migration ratios in rounds of fixed duration; (iii) simulation results of the proposed heuristic algorithm; (iv) results of the implementation of the proposed algorithm in a testbed of Raspberry Pis which shows the efficacy of the solution even in a real setting;
- Section 5.2 presents: (i(software libraries and methods for implementing mobile object recognition task and for offloading it to a possible Edge device by using publicly available open-source

frameworks and tools; (ii) measurements and analysis of the energy impact of a mobile neural network directly running in a mobile smartphone and the consumption gain that we can obtain when the neural network recognition task is offloaded to an Edge device; (iii) quantifies the beneficial effect of the task offloading.

- in Section 5.3 the load balancing problem is instead focused on energy consumption in a Green Edge Computing environment; the section presents (i) a characterisation of an off-grid green Edge computing model and (ii) an initial evaluation of the benefit of task green energy aware task offloading.

Distributed algorithms for scheduling Research contributions in this area regard the design of algorithms which are able to allocate clients' requests to the correct node in order to follow defined constraints. For example, tasks' completion time has to meet a deadline. As in the previous point, we only focus on fully distributed and cooperative algorithms.

- Section 4.2 presents (i) the design of a decentralised RL-based algorithm to be implemented in every Fog node that is able to choose the best scheduling decision according to the current load situation; then (ii) a study of a geographic setting which involves six Fog nodes deployed in the city of New York and in which the algorithm can be deployed, and finally the (iii) simulation results on a delay-based simulator prove the efficiency of the algorithm compared to the classic *power-of-n* strategy, moreover are also shown (iv) results from a pseudo-real deployment with the prototype framework "P2PFaaS" in a rack of 12 Raspberry Pis;
- Section 4.3 presents (i) an RL-based online scheduling algorithm for the computing continuum that is able to cope with node inhomogeneity and to satisfy user-defined processing frame rate requirement; then (ii) simulation results of the proposed algorithm in two main settings, one cluster or more clusters in the Edge layer, within a simulator that is focused on replicating fine-grained delays that a job may encounter during its execution path.

Testbeds and implementation design Research contributions in this area regard the study of techniques and solutions whose purpose is the development and implementation of load balancing and scheduling algorithms in real or pseudo-real environments.

- Section 6.2 illustrates (i) the definition of hardware and software requirements for a long-term, unattended and remote controllable solution for implementing a Raspberry Pi cluster, then it presents the (ii) design of a power supply board for using a desktop computer power supply (called ATX) for powering up to eight Raspberry Pi boards; moreover, the Section shows the (iii) design of a remote Ethernet switch system for remote controlling the power of the cluster to be associated with the power supply board and (iv) propose a way to define the testbed configuration and an experiment via JSON configuration files, towards a Testbed-as-a-Service paradigm; finally (v) the results of the benchmark of a distributed scheduling algorithm installed in the cluster are shown;
- Section 6.3 shows the internal design and the purpose of the open-source "P2PFaaS" framework that we started to develop during the MSc Thesis and that we continued to develop during the PhD for practically implementing the load balancing and scheduling algorithms tested only in

simulations.

1.2 Thesis Outline

The content outline of the thesis has been anticipated by the thesis contribution (Section 1.1). However, the overall scheme of the thesis is the following. Chapter 2 studies load balancing approaches based on a randomised approach. In particular, the *power-of-n* choices paradigm is exploited, a strategy which is based on random probing of neighbour nodes. All the strategies presented in the Chapter use probabilistic models of the system. Instead, Chapter 3 focuses on a solution which derives a load balancing algorithm based on a non-probabilistic model. Indeed, the differential model used for modelling the Fog or the Edge nodes is based on the dynamical systems theory. Chapter 4 shows a series of works in which the scheduling decision is made with Reinforcement Learning. The solution is applied both in a pure Fog environment and in a Edge/Fog-to-Cloud computing continuum with nodes organised in clusters. Chapter 5 presents strategies which are instead oriented to Green Edge Computed, and therefore they specifically target energy consumption, task offloading and solar energy contribution. Finally, Chapter 6 focuses on the infrastructural side of algorithms. In particular, a testbed for an Edge Computing solution will be shown with a final description of the open-source framework "P2PFaaS"[4] that we started to develop during my MSc Thesis and that we continued to improve during the PhD. The framework allows all the algorithms presented in theory and simulations to be implemented in a real environment. In particular, it has been installed and tested within a testbed of Raspberry Pi computers. Conclusions are finally drawn in Chapter 7.

1.3 Publications

A great part of the works presented in this thesis have been published in international conferences and journals, and for each section, the relative publications will be referenced. What follows is the list of the publications that I authored or co-authored during my PhD.

Publications accepted

(1) **G. Proietti Mattia** and R. Beraldi, "P2pfaas: A framework for faas peer-to-peer scheduling and load balancing in fog and edge computing," *SoftwareX*, vol. 21, p. 101 290, 2023, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2022.101290. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711022002084

(2) **G. Proietti Mattia**, M. Magnani, and R. Beraldi, "A latency-levelling load balancing algorithm for fog and edge computing," in *25th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'22)*, Montreal, Canada, Oct. 2022. DOI: 10.1145/3551659.3559048. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3551659.3559048

[4] https://gitlab.com/p2p-faas

(3) R. Beraldi and **G. Proietti Mattia**, "On off-grid green solar panel supplied edge computing," in *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS) (IEEE MASS 2022)*, Denver, USA, Oct. 2022

(4) G. Maiorano, **G. Proietti Mattia**, and R. Beraldi, "Local and remote fog based trade-offs for qoe in vr applications by using cloudxr and oculus air link," in *International Conference on Edge Computing and Applications (ICECAA 2022)*, Namakkal, India, Oct. 2022

(5) **G. Proietti Mattia** and R. Beraldi, "On real-time scheduling in fog computing: A reinforcement learning algorithm with application to smart cities," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022, pp. 187–193. DOI: 10.1109/PerComWorkshops53856.2022.9767498

(6) R. Beraldi, C. Canali, R. Lancellotti, and **G. Proietti Mattia**, "On the impact of stale information on distributed online load balancing protocols for edge computing," *Computer Networks*, p. 108 935, 2022, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2022.108935. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128622001207

(7) **G. Proietti Mattia** and R. Beraldi, "Leveraging reinforcement learning for online scheduling of real-time tasks in the edge/fog-to-cloud computing continuum," in *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, 2021, pp. 1–9. DOI: 10.1109/NCA53618.2021.9685413

(8) **G. Proietti Mattia** and R. Beraldi, "A study on real-time image processing applications with edge computing support for mobile devices," in *2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2021, pp. 1–7. DOI: 10.1109/DS-RT52167.2021.9576139

(9) T. Alawsi, **G. Proietti Mattia**, Z. Al-Bawi, and R. Beraldi, "Smartphone-based colorimetric sensor application for measuring biochemical material concentration," *Sensing and Bio-Sensing Research*, p. 100 404, 2021, ISSN: 2214-1804. DOI: https://doi.org/10.1016/j.sbsr.2021.100404. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S221418042100009X

(10) **G. Proietti Mattia** and R. Beraldi, "Towards testbed as-a-service: Design and implementation of an unattended soc cluster," in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021, pp. 1–8. DOI: 10.1109/ICCCN52240.2021.9522323. [Online]. Available: https://ieeexplore.ieee.org/document/9522323

(11) R. Beraldi and **G. Proietti Mattia**, "Power of random choices made efficient for fog computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020, ISSN: 2372-0018. DOI: 10.1109/TCC.2020.2968443

(12) R. Beraldi, C. Canali, R. Lancellotti, and **G. Proietti Mattia**, "A random walk based load balancing algorithm for fog computing," in *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, 2020, pp. 46–53. DOI: 10.1109/FMEC49853.2020.9144962

(13) R. Beraldi, C. Canali, R. Lancellotti, and **G. Proietti Mattia**, "Distributed load balancing for heterogeneous fog computing infrastructures in smart cities," *Pervasive and Mobile Computing*,

p. 101 221, 2020, ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2020.101221. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119220300791

(14) R. Beraldi, C. Canali, R. Lancellotti, and **G. Proietti Mattia**, "Randomized load balancing under loosely correlated state information in fog computing," in *23rd ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'20)*, Alicante, Spain, Nov. 2020. DOI: 10.1145/3416010.3423244

(15) R. Beraldi and **G. Proietti Mattia**, "A randomized low latency resource sharing algorithm for fog computing," in *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT) (DS-RT'19)*, Cosenza, Italy, Oct. 2019. DOI: 10.1109/DS-RT47707.2019.8958709. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8958709

Chapter 2

Randomized strategies

> Our universe is ruled by random whim,
> inhabited by people who laugh at logic.
>
> — Jeff Lindsay

The decision of using randomness for making scheduling decisions may sound slightly controversial. One may wonder how it is possible to obtain appreciable results if we choose, at random, nodes to which tasks have to be scheduled. However, under certain conditions and when some information about the state cannot be known, a random approach is proven to be preferable. Indeed, when the state is unknown, it may be unfeasible to probe each neighbour node to retrieve their state and then make the scheduling decision. Probing requires time, and a delay is added to the requests, which cannot be scheduled until all the probing requests are completed. The work presented in this chapter uses randomness for scheduling a task to a neighbour when the current node is not able to execute it. Therefore, instead of directly rejecting the request, it can make sense to forward them randomly.

There is a fundamental result from which the algorithms presented in this chapter are derived, and it is called the "*power-of-n* choices", which was extensively studied in literature [18] [19] and particularly fits our assumptions. The solution is applied to the so-called supermarket model, where we hypothesise that customers arrive and are served in a FIFO (First-In First-Out) protocol. Each customer must choose d servers out n uniformly at random, and it chooses the less loaded. It is proven that when we pass from $d = 1$ to $d = 2$ the improvement in performances is exponential, while when $d > 2$ the improvement is only by a constant factor. When applied to Fog and Edge Computing, we can actually map customers to requests that arrive at the nodes. Then, when a node has to decide to forward the task at random, if

it uses $d = 2$ and probes one node uniformly at random (the other node is the current one) then the same *power-of-n* result holds. Variants of this scheme have been studied with their implications even in a real Fog Computing environment, and they are presented in Section 2.2 and in Section 2.3.

Another important aspect that emerges during the random probing is the time delay that it implies. As anticipated in the introduction, it is obviously not possible to retrieve the state information from another node without a time delay. This means that when the scheduling decision is made and based on that information, then it will always be made on old information since the state of the probed node changed over time. From a merely qualitative and intuitive point of view, if the information that we retrieve is too old, then its significance to us could become negligible, and it would have been convenient to forward the task blindly without the probing. In Section 2.4, by using a probabilistic model, a quantification of this convenience threshold is given.

Concluding, the rest of the Chapter is organised as follows. Section 2.1 presents related works, Section 2.2 studies the implication of the *power-of-n* strategy when applied to a Fog Computing environment, Section 2.3 studies two variants of the random-based approaches called Adaptive Forwarding and Sequential Forwarding, finally, Section 2.4 focuses on the impact of the stale information when the scheduling decision is made. The algorithms and the results presented in Sections 2.2, 2.3 and 2.4 have been published respectively in [13], [15] and [8].

2.1 Related Work

Randomised approaches to load balancing The main randomised strategy used in the works presented in this chapter is based on the "power-of-d choices" result [19]. Load balancing algorithms based on this result adopt a unique scheduler that receives jobs to be dispatched to one of N equivalent workers. Scheduling decisions are performed by sampling the state of d workers and selecting the most convenient one according to their workloads. The main success of this strategy is that it avoids keeping the state of all the workers while being remarkably effective, which is particularly useful when N is high. A rich and sound literature has studied the property of these algorithms for $N \to \infty$. In this limit, [20] shows the asymptotic independence among queues, which allows for simplifying the analysis of these systems. All of the analyses performed in this chapter consider this ansatz.

The studies that characterise this randomisation algorithm in the context of load balancing can be divided into two bodies depending on the model of the workers that either cannot or can lose jobs. The first set of results, e.g., [19], [21], [22] model workers as M/M/1 queues and the selection criteria is to pick the Shortest among the d sampled Queues, $SQ(d)$. The most relevant finding is that the average job execution delay decay doubly exponentially in the limit as the number of servers goes to infinity, which is a substantial improvement over a classical queue case, where the queue size decays exponentially. A finer workload measurement is used in [23]. This extraordinary improvement has motivated the research on a different worker model that can somehow better fit applications to cloud computing, e.g., [24]. The second body of works, indeed, models the worker as a finite set of K servers, e.g., an M/M/K queue, and a job that cannot be scheduled it is blocked, [22], [24]–[26]. The selection criteria here is to pick the Least Loaded among d queues, $LL(d)$, where the server load is the number of busy servers. These works analyse the effect of randomisation on the blocking probability, which

is the counterpart of average delay. However, all models assume a single centralised dispatcher, and hence threshold-based control cannot be applied. The work in [27] is the most related to the strategies presented in this chapter. In the protocols presented in this chapter, if a job execution request arrives at an overloaded Fog node, the job is forwarded to a neighbouring node with some probability. Hence, contrary to the proposed protocol by the authors in which no power-of-choice is used. In [28] the power-of-d choices paradigm has been used to enable cooperation among different Fog providers. No thresholds are used, and the protocol is limited to unitary fanout. In [29] random choices are used for p2p load balancing protocols.

Load balancing for the Fog model has been studied in several papers. In [30] an algorithm called Multi-tenant Load Distribution Algorithm for Fog Environments (MtLDF) has been proposed to optimise load balancing in Fogs environments considering specific multi-tenancy requirements. However, the proposed load balancing scheme adopts a centralised Fog management layer that receives all the state information about the Fog nodes. In [31], the tasks that the nodes are called to complete are characterised according to their computational nature and are subsequently allocated to the appropriate host. Edge networks communicate through a brokering system with IoT systems in an asynchronous way via the Pub/Sub messaging pattern. However, a centralised workload balancer is used in the solution. In [32] an approach is presented to periodically distributing incoming tasks in the Edge computing network so that the number of tasks, which can be processed in the Edge computing network, is increased, and the quality-of-service (QoS) requirements of the tasks completed in the Edge computing network are satisfied. The model, however, assumes that a set of tasks to be assigned is available, i.e., the tasks are not processed online. In [33], a mechanism for load balancing policy is presented with a dynamic threshold, which is computed each time the scheduling is applied. In [34] and [35], the load balancing algorithm is based on a BFS search by also addressing the problem of the secure authentication between nodes. Other works are focused on the trade-off between energy consumption and latencies [36]. Finally, Yousefpour *et al.* [37] propose a system similar to the presented Sequential Forwarding algorithm (Section 2.3.1.1). However, the proposed solution still requires a centralised repository to store the load state of each Fog node. A variant of the proposed system relies on a specific communication pattern similar to a gossip protocol to send updates on the load state of each node. However, some of the approaches presented and based on blind forwarding provide good performance without complex coordination structures.

As a final remark, it is worth noting that almost all the referenced studies do not explicitly take into account the heterogeneous nature of a Fog computing infrastructure, while the analyses presented in this chapter explicitly focus on this aspect.

Management of Fog computing infrastructures The interest in managing Fog computing infrastructures has been addressed in literature in multiple ways. On the one hand, a corpus of literature aims to address the problem of connecting end devices (e.g., sensors) to Fog nodes and Fog nodes to cloud data centres. For example [38] proposes an optimisation model based on energy consumption to map processing tasks over Fog nodes and cloud data centres. A different approach is introduced in [39] where the focus is more oriented towards providing a good mapping between Fog nodes and sensors in a heterogeneous environment. However, both these papers follow an approach where the incoming data

flows and computing tasks are statically assigned over the infrastructure. The algorithms presented in this chapter are, instead, completely dynamic and can easily adapt to workload conditions that change over time. The experiments show that with, limited tuning, it is relatively easy to adapt to a wide range of application scenarios.

Impact of delay on load balancing performance The negative impact of stale information due to network-related delays on load balancing performance in distributed infrastructures was pointed out several years ago in the area of Web systems [40] and in a distributed setting in the seminal work [41]. In this Chapter, the focus is to analyse the effect of network delays in the novel area of Edge computing. In particular, the study presented in Section 2.4 focuses on load-aware probe-based approaches for load balancing among Edge nodes, considering their potential issues. Furthermore, the analysis is not limited to network delays. Indeed, the focus is on specific application requirements through the parameter η, which expresses the transfer-to-compute ratio. This allows understanding under which circumstances the probe-based approach can be preferable to the load-blind approach of the sequential forwarding algorithm.

Vertical Offloading vs Horizontal Load Balancing With the increasing heterogeneity of the computational capabilities of mobile devices and sensors used in large distributed applications, both the Edge and the cloud level can be exploited to share the load of the required data processing and satisfy the application requirements. The (total or partial) transfer of computationally intensive jobs from the local device/sensor to Edge and cloud nodes, called offloading, has been widely studied in the context of mobile cloud computing [42], [43], where is used to transfer the computational load to the resource-rich cloud infrastructure.

The cloud is often remotely located and far from end-users and sensors, so the data transfer delays can be long and unpredictable. To reduce the perceived latency, mobile Edge computing (MEC) has been proposed by several studies [44], [45] for offloading a part of the workload from mobile devices to the intermediate level of Edge nodes with sufficient computational resources. However, these studies typically do not consider horizontal cooperation strategies for load balancing among Edge nodes. Some studies focus on the Edge server placement issue [46] or include dynamic service migration to deal with erratic user mobility [47] but do not consider cooperation strategies at the Edge level.

Application Scenarios Many distributed applications, ranging from Industrial IoT to smart city contexts, rely on sensors and, in general, end devices to collect data that need to be processed for extracting features and information needed to provide the required final service [48]. Such applications may have very different requirements in terms of processing time and amount of data to be processed: these elements have a direct impact respectively on T_A (time required to calculate the mapping A between units of computations) and T_C (execution time) as defined in the introduction of this section. In this study, the parameter η is introduced to take into account the transfer-to-compute ratio characterising the application.

For example, a typical application that can take advantage of Edge computing support is video processing: by extracting at the Edge level only a few video features to be sent to the cloud, network

resources can be saved, and application latency decreased. The studies proposing Edge computing solutions for image processing [49], [50] usually focus only on the high computational load required (sometimes addressed by splitting among Edge nodes non-overlapping partitions of the video frames as in [50]) but do not consider the network-related contributions. However, depending on the specific characteristics of the application, it is not always correct to assume $\eta \ll 1$, meaning that the processing time is higher than the network contributions. In the case of high-performance image processing and high frame rate, the network latency and the jobs transfer time may become comparable or even higher than the job execution time. On the other hand, large distributed applications may also involve thin data to be transmitted and processed. This typically happens with crowdsensing applications based on data collected by geographically distributed sensors [51], but it may also be the case of Industry 4.0 or IoT-Based Manufacturing scenarios [52]: in all these applications, the data size may be around a few Kbytes (e.g., a small JSON file). However, thin data's execution time may vary significantly depending on the specific scenario. Indeed, for applications requiring simple computations of statistical indicators on the time series of the collected data, the execution time may be short and comparable with the network-related and transfer times, leading to a case characterised by a value of $\eta \approx 1$. On the other hand, in applications where machine learning techniques are applied [52] (for example, for complex forecasting, feature inference or patterns prediction), the execution time may significantly increase, leading to a scenario with $\eta < 1$. An added value of this study with respect to the state of the art is to consider which load balancing approach is preferable depending on specific scenarios and application requirements.

2.2 Specialising the "*power-of-n* choices" paradigm for Fog Computing

The ongoing advances in the ICT, from 5G to powerful open-source software libraries, e.g., open Computer Vision and TensorFlow, are creating the prerequisites for software applications positioned in the left upper corner of the latency bandwidth Cartesian product with use cases in different vertical domains, e.g., ranging from IoT, smart cities, AR/VR based applications with haptic interactions, as well as Tactile Internet, [53]. Fog computing is widely considered the key enabler for such applications as it makes backend capabilities physically close to the end-users and integrated with the usual cloud services. Fog computing is seen as an integrated intermediate layer along the thing-to-cloud path [19], [54]–[56]; readers may refer to [57] for an overview and tutorial on Fog computing.

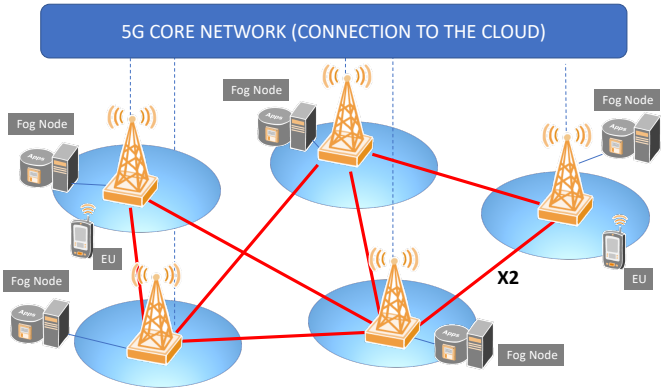


Figure 2.2.1: An illustration of the inter-node communication among 5G Fog nodes, that can support the load balancing discussed in this study.

Fog-enabled applications are a composition of units of computation, e.g., adhering to the Function as a Service (FaaS) delivery model, with the time-critical parts deployed on Fog nodes, [58]. OpenFog's Fog computing Reference Architecture [59], recently adopted as the IEEE-1934 standard, envisions Fog nodes to form a mesh to provide load balancing and minimisation of cloud communication. In particular, they may communicate laterally in a peer-to-peer fashion. For example, in the 5G architecture, a high number of devices are networked by a capillary network, and nearby Radio Access Network, equipped with Fog computing capabilities, (F-RAN) may communicate directly, through the so-called X2 or X2* interfaces, [60], [61], see Figure 2.2.1. This inter-node communication carries both control and user planes and allows for increasing the fraction of the service requirements that can be responded locally without interacting with the cloud computing centre via the fronthaul links. For example, a computation request generated from an End User (EU) connected to an F-RAN can be served by another nearby F-RANs, even via a virtual multi-hop connection, where the number of hops is limited to some units.

Also, in the industrial standard ETSI's Multi-access Edge Computing (MEC) reference architecture, the envision orchestration leverages on code relocation and traffic steering to dynamically move application code or data, [62].

Motivated by the above scenarios, in this section, we study the problem of how load balancing jobs among a set of Fog computation resources that provide the same computation service, for example, the object detection service described in [63]. Load balancing in Fog computing is, in fact, an open issue that may play an important role to enable Fog based applications, [57], [64]–[66].

In more detail, we focus on a Fog layer composed of N nearby Fog nodes, with K identical servers. The Fog nodes are willing to share their resources with the goal of reducing the fraction of jobs forwarded for execution to a distant cloud. Incentives to cooperate and the benefit of such sharing are discussed, for example, in [67]. In addition, jobs have no information related to their deadlines, computation requirements or priorities. It is worth noting that N can potentially be high. For example, in the 5G architecture, higher frequencies than 4G are not capable of travelling large distances. This requires placing 5G RAN every few hundred meters in order to utilise such higher frequency bands.

The proposed algorithm is an adaptation of the $LL(d)$ algorithm (Least Loaded among d random nodes) to the above Fog model. A Fog node normally executes jobs received from its direct users without any load balancing action. However, when its workload is higher than a threshold T, a node tries to delegate the execution of a job to another less loaded node among F randomly probed nodes [1]. We refer to this algorithm as $LL(F, T)$. This threshold regulation is a simple yet effective mechanism that reduces remote scheduling overhead remarkably, avoiding the inefficiency arising when autonomous schedulers compete on sharing a common set of resources, [41], [68], [69].

The contribution presented in this section can be summarised as follows.

- Definition of $LL(F, T)$, Least Loaded among F nodes with threshold T, a new scheduling algorithm the Fog computing model based on the power-of-random choices randomisation principle.
- Mathematical analysis of the algorithm showing how $LL(1, K-2)$, i.e., probing just one node when one or two servers out of the K are idle, reduces up to one order of magnitude the control and delay overhead with respect to a vanilla randomized load balancing implementation of the power-of-two random choices algorithm.

2.2.1 System Model

In this section, we describe the proposed solution to the load balancing problem for Fog computing. In general terms, the load balancing problem consists in determining how to allocate jobs generated by end-users to Fog nodes in an efficient way. More formally, we are given a set of N homogeneous Fog nodes with limited resource capacity, where each one is receiving a continuous flow of computation requests, or jobs. A job is blocked when it cannot be executed due to the lack of available resources. We call p_B the probability that this event occurs. A load balancing algorithm is a rule that assigns jobs to servers in a way that p_B is minimised. The efficiency of the rule is measured by the amount of control overhead generated.

A power-of-d random choices load balancing algorithm is a rule executed by a dispatcher to decide where the jobs received have to be forwarded and executed among a set of N equivalent workers. On

[1] We use a different symbol, F to indicate the number of probed nodes to make the difference with the original protocol clearer.

receiving a job, the dispatcher probes a small subset d of workers and then sends the job to the Least Loaded among them with ties broken at random. The key characteristic of the algorithm is $d \ll N$ and even $d = 2$ has shown to be very effective compared to probing all nodes. The algorithm, referred as $LL(d)$, requires an overhead as small as d control messages per job and scalable as this overhead doesn't depend on N.

A general Fog computing deploy model is a hierarchical multi-tier architecture where communication can also occur among nodes of the same tier, [59]. We focus on the first tier that provides access to end-users via N nodes. The application of $LL(d)$ as-is implies the existence of N dispatchers and N workers, and this has two main drawbacks. The first one is that in general concurrent dispatchers tend to be inefficient as their decisions are biased towards workers that appear lightly loaded, [41], [68]. In some extreme cases, this implementation may even deteriorate the initial performance, e.g., see [69]. The second implication is that the dispatching operations introduce a delay overhead for any job, even when nodes do not need to distribute the load, for example, because most of its current servers are idle, as discussed later. For these reasons, we propose the following algorithm that keeps the load balancing approach unchanged but allows reducing the aforementioned drawbacks. The design of the algorithm is based on the following assumptions:

- A1: The communication latency among nodes, although not negligible, is small compared with the job execution time. This assumption follows from the observation that for example, the time required for image recognition is in the order of tens of ms, while an X2 or X2* interface that connects Fog nodes - either directly or through a few relaying nodes, is likely to be not higher than a few ms, see Figure 2.2.1.
- A2: Probing occurs only among N nodes in the same tier composed of nearby Fog nodes, where N can be high. This assumption follows from the observation that the density of Fog nodes in a geographical can be high. For example, F-RAN can cover just hundreds of meters. If a job cannot be executed in the tier, the job is forwarded to the cloud. The protocol aims at reducing the probability that these events occur. Nodes do not have a waiting queue for incoming jobs, i.e., the workload is measured as the number of running jobs.
- A3: nodes are homogeneous, they have the same number of servers K and provide the same service at the same speed. The generalisation to the non-homogenous case is easy and left as future work. We will, however, report simulation results that cover this case.

2.2.1.1 $LL(F,T)$: Least Loaded with fan-out F and Threshold T

In this section, we describe the Least Loaded among F random nodes with threshold T load balancing protocol, referred to as $LL(F,T)$. The value F is called the protocol *fan-out*. The algorithm works as follows, see Figure 2.2.2.

When a node receives a job, if $k < T$, then the node executes the job immediately, otherwise, it probes the state of F different random nodes and computes the minimum among the returned values, say m. When $k < K$, if $m \geq k$, then the node executes the job locally; otherwise, it forwards the job to the node reporting m, with ties broken at random. If the receiving node, as well as the probed nodes, have no free servers, i.e., $k, m = K$, the job cannot be served by this Fog layer, and we count the job

Number of nodes	N
Number of servers	K
Arrival rate per node	λ
Number of nodes running k jobs	n_k
Number of nodes running at most k jobs	$\widetilde{n}_k = \sum_{i=1}^{k} n_i$
Stationary state probability for the infinity model	π_k
Tail of the stationary state probability	$\widetilde{\pi}_k = \sum_{i=k}^{K} \pi_i$

Table 2.1: Main symbols used in the analysis.

as being blocked. A blocked node can trigger different actions, like sending the job to an upper Fog layer (e.g., the cloud). We limit to consider this metric as an important performance measure of the algorithm with higher blocking probability being associated eventually to worse end-user performance.

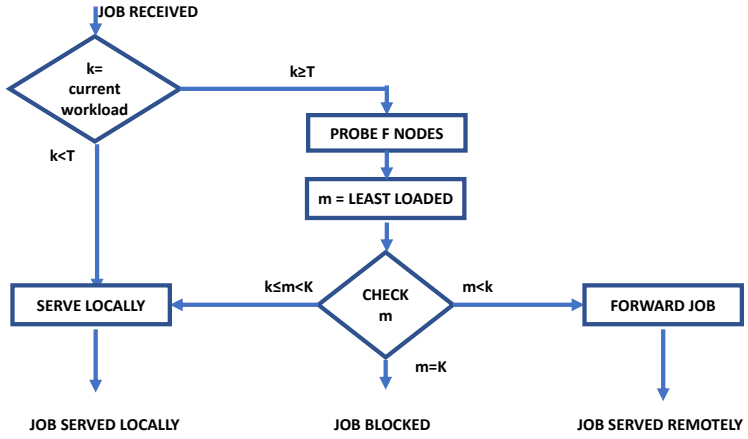


Figure 2.2.2: Least Loaded among F probed nodes with Threshold T control flow. K is the total number of servers available, k the number of busy servers at the receiving node. A job executed remotely means that the serving Fog node is not the node that received the job.

2.2.1.2 Protocol analysis

We study the performance of the $LL(F,T)$ protocol under a Poisson traffic with rate λ jobs per unit of time per node and exponential service time with mean one. Table 2.1 summarises the main symbols used in this study.

Protocol analysis for N finite This model considers a system of N homogeneous Fog nodes with K servers. The model is based on a standard Continuous Time Markov Chain (CTMC). The state of the

system is expressed through the vector:

$$\mathbf{n} = (n_0, n_1, \ldots, n_K) \quad 0 \leq n_i \leq N, \sum_i n_i = N$$

where n_i is the number of nodes with i running jobs. This direct method can only be applied to small values N, K, as the state space of the model explodes (see the [13] for a quantitative evaluation). However, it is anticipated that the characteristics of the load balancing protocol appear even with small values for N, K. The chain is solved numerically from its canonical formulation:

$$\pi Q = 0, \;\; \mathbf{1}\pi = 1$$

To deal with the memory issues, we've avoided direct matrix inversion and used a power method instead, as it allows for storing the spare matrix efficiently.

Let $\mathbf{e_k}$ be a $K+1$ sized vector of all 0s except 1 in position k. The entry $q(\mathbf{n}, \mathbf{n}')$ of the infinitesimal transition matrix Q is expressed as:

$$q(\mathbf{n}, \mathbf{n}') = \begin{cases} \lambda_i(\mathbf{n}, F) & \mathbf{n}' = \mathbf{n} - \mathbf{e_k} + \mathbf{e_{k+1}} \\ \mu_i(\mathbf{n}) & \mathbf{n}' = \mathbf{n} + \mathbf{e_{k-1}} - \mathbf{e_k}, \\ -\sum_i [\lambda_i(\mathbf{n}, F) + \mu_{i+1}(\mathbf{n})] & \mathbf{n} = \mathbf{n}' \\ 0 & otherwise \end{cases}$$

where $\mu_i(\mathbf{n}) = i \times \mu n_i$ (n_i is the i-th component of $\mathbf{n}$). Let us define the function:

$$\delta_{Nn}^F = \frac{n!}{(n-F)!} \frac{(N-F)!}{N!}$$

with the convention that $\delta_{Nn}^F = 0$ if $F < 0$. This number represents the probability to extract form an urn of N balls containing n balls of the same type, F balls of such a type without replacement.

The birth rate is conveniently divided into the sum of two flows. The first flow:

$$\lambda_{1k}(\mathbf{n}, F) = \lambda n_k \begin{cases} 1 & k < T \\ \delta_{N-1\widetilde{n}_k-1}^F & k \geq T \end{cases}$$

represents jobs that arrive directly to Fog nodes. For $k \geq T$ a job is served only if the state of all the F probed nodes is at least k. Let $\widetilde{n}_k$ the number of jobs with at least k jobs running. As the number of these nodes excluding the node itself is $\widetilde{n}_k - 1$, this event occurs with probability given by $\delta(\cdot)$ in the above expression. The second contribution is given by:

$$\lambda_{2k}(\mathbf{n}, F) = \lambda \begin{cases} \widetilde{n}_T(\delta_{N-1\widetilde{n}_k-1}^F - \delta_{N-1\widetilde{n}_{k+1}-1}^F) & k < T \\ \widetilde{n}_{k+1}(\delta_{N-1\widetilde{n}_k-1}^F - \delta_{N-1\widetilde{n}_{k+1}-1}^F) & k \geq T \end{cases}$$

where the term in the parenthesis is the probability that the minimum state of at least one probed node

is k.

Performance metrics N finite The following performance metrics are defined in terms of the stationary probabilities π_k of the CTMC process. As far as the delay is concerned, we approximate its evaluation by assuming that each round trip time is a uniform Random Variable $U = (0, 1]$ and correlations exit among probings.

- Blocking probability, p_b. This value corresponds to the probability of the event that a job received from a node cannot be executed, which occurs when the state of the receiving node, as well as all the F probed nodes, is K. As a job is blocked when the state of all the F different probed nodes is K, we have
$$p_b = \sum_{\pi_{\mathbf{n}}: n_K \geq F+1} \frac{n_K}{N} \cdots \frac{n_K - F}{N - F} \pi_{\mathbf{n}}$$
- Average delay per job, D.
This delay is the sum of the delay due to probing, job forwarding and result reply and it is given by:
$$D = \left[\frac{F}{F+1} \frac{1}{N} \sum_{\mathbf{n}: n_k \geq T} n_k \pi_{\mathbf{n}} \right] + \frac{1}{2} fwd$$
where fwd is the fraction of received jobs that are forwarded to another node. The first term is the mean of F uniform RV and $\frac{n_k}{N}$ is the probability that a job arrives to a node whose state is k. The fraction of job forwarded is given by:
$$fwd = \sum_{\pi_{\mathbf{n}}: \tilde{n}_k > 0} \frac{n_k}{N} \left(1 - \prod_{i=1}^{min\{\widetilde{n}_k, F\}} \frac{\widetilde{n}_k - i}{N - i} \right) \pi_{\mathbf{n}}$$
which represents the probability that a job arrives to any of the N nodes, sees the Fog node with k servers busy, and the state of all the probed nodes is at least k.
- Overhead per job, ovh.
As probing is triggered when a job arrives to any of the n_k out of N nodes whose state is $k \geq T$ and the probability such a job arrival event occurs is $\frac{n_k}{N}$, we have:
$$ovh = \frac{F}{N} \sum_{\mathbf{n}: n_k \geq T} n_k \pi_{\mathbf{n}}$$

Protocol analysis N infinite We now characterise the $LL(F, T)$ protocol in the limit of $N \to \infty$ Fog nodes. We make the *conjecture* that when N grows, the dynamics of a set of finite nodes tend to become independent one from each other. While a formal proof of such asymptotic independence is difficult, We remark that this approach has been taken in other works, e.g., [20], [24], [26] and it is the basis of the widely used mean-field theory. What is proven is that the model, arising from this assumption, considered per se has a single solution. If the conjecture is true, the model then provides correct results. The presented numerical results have similar shapes compared to the model with finite

N, thus making this conjecture reasonable.

Suppose then that queues describing the nodes are independent, and let us focus on a single tagged node. The state of this tagged node boils down to the number of its busy servers and changes according to a Birth-Death (BD) process with state-dependent birth transition rates, as defined next.

Let π_k the probability of the state being k, and $\widetilde{\pi}_k = \sum_{j=k}^{K} \pi_j$ the tail distribution of the state variable. The BD process describing the state of a node should satisfy the following set of K equations:

$$\lambda_k \pi_k = (k+1)\pi_{k+1} \quad k = 0 \dots K-1 \tag{2.1}$$

where $\sum_k \pi_k = 1$ and $\lambda_k = \lambda_{1k} + \lambda_{2k}$.

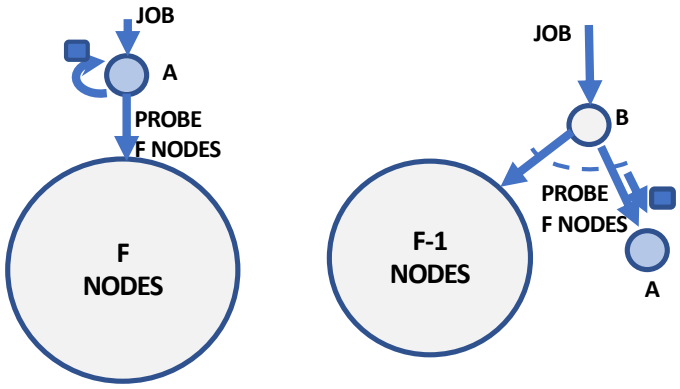


Figure 2.2.3: Traffic flows seen by a node A. A job is served by the node when (i) it is received directly from A (left); (ii) it is probed by some other node B, that then forwarded to it.

The tagged node executes jobs generated directly from its users (left side in Figure 2.2.3), when $k < T$, or $T \leq k < K$ and the state of all the F probed nodes is at least k. The birth rate associated to these events is

$$\lambda_{1k} = \lambda \times \begin{cases} 1 & k < T \\ \widetilde{\pi}_k^F & k \geq T \end{cases} \tag{2.2}$$

Node A may also execute a job on behalf of another node, say B in the right side of Figure 2.2.3. This occurs when B selects A and before that B probed other $F-1$ nodes. If the state of A is k, this occurs when the state of B is higher than k, k is also the minimum state of other i out of the $F-1$ other probed nodes, and B selected A among the $i+1$ least loaded nodes. Since B probes other nodes when the number of busy servers is at least T, these transitions occur with rate

$$\lambda_{2k} = \alpha(k,T) \sum_{i=0}^{F-1} \binom{F-1}{i} \pi_k^i \widetilde{\pi}_{k+1}^{F-i-1} \frac{1}{i+1} \tag{2.3}$$

where

$$\alpha(k,T) = \lambda \begin{cases} \widetilde{\pi}_T & k < T \\ \widetilde{\pi}_{k+1} & k \geq T, \end{cases}$$

Properties

Theorem 2.2.1 (Solution). *Equations Equ. (2.1) have a single solution.*

Lemma 2.2.1 (Upper bound). *For given $F, T, \lambda < K$, let*

$$\hat{\widetilde{\pi}}_k = \frac{\lambda^{(F+1)^{\frac{k-T-1}{F}}}}{\Pi_{i=0}^{k-T}(k-i)^{(F+1)^i}}$$

then if $\lfloor \lambda \rfloor \leq T$ when λ in not an integer and $\lambda < T$ otherwise, the following upper bound holds

$$\widetilde{\pi}_k \leq \hat{\widetilde{\pi}}_k \quad k = T, \ldots, K$$

Proof. see [13]. □

Lemma 2.2.2 (Equivalence). *$LL(F,1)$ is equivalent to $LL(d)$, where $d = F+1$.*

Proof. See [13]. □

Performance metrics The counterpart of the performance metrics for this model is defined in terms of the stationary probabilities π_k of the BD process describing the $LL(F,T)$ algorithm.

- Blocking probability, p_B.

  $$p_B = \pi_K^{F+1}$$

  In fact, this value corresponds to the probability of the event that the state of the node receiving a job as well as all the other F probed nodes is K. Note that $p_B \leq \hat{\pi}_K^{F+1}$

- Average delay overhead per job, D.

  $$D = \frac{F}{F+1} \widetilde{\pi}_T + \frac{1}{2} fwd$$

  where fwd is the fraction of received jobs that are forwarded to another node. As a node with state $k \geq T$ forwards a received job if the state of at least a probed node is lower that k, and this occurs $1 - \widetilde{\pi}_k^F$, it is:

  $$fwd = \sum_{T=k}^{K} \pi_k (1 - \widetilde{\pi}_k^F)$$

- Average number of generated probing messages per received job, h. As F messages are generated when the state of the job's receiving node is at least T, this value is hence given by:

$$ovh = F\widetilde{\pi}_T$$

2.2.2 Threshold limit on the power-of-n choices

It is known that the extraordinary efficacy of the power-of-d-choices algorithm is because the state dynamics of a node under $LL(d)$ is radically different from when the node works in isolation. The $LL(F,T)$ algorithm works between two extreme points. For $T = 1$ the algorithm is equivalent to $LL(d)$ with $d = F + 1$, and hence it completely exploits the power of choices effect, whereas with no cooperation, $T = \infty$, each node works in isolation. It is then worth to better understand how T determines the raising of this change, e.g., how and if it is smooth or not. We provide here some numerical results concerning this aspect.

A formal way to measure how T affects the state distribution is to compute the difference among the schemes as a *distance*:

$$dist(F,T) = \left(\sum_k |\pi_k - \pi'_k|^2\right)^{\frac{1}{2}}$$

where π_k (π'_k) is the state probability of $LL(F,T)$ and $LL(d)$ with $d = F + 1$, respectively.

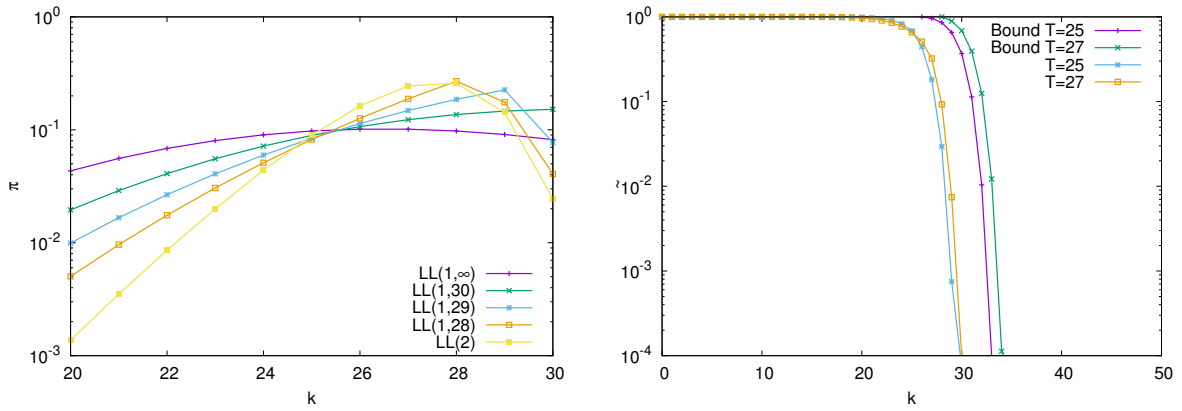
Table 2 reports the distance $dist(F,T)$ for different values F, T, and $K = 30, \lambda = 25$. The difference falls sharply as T becomes lower than λ. For example a distance of less than 0.005 is reached when $T = \lambda = 25$. Also, the fan-out F makes this change even stronger.

T	30	28	26	25	22
F=1	2.7e-01	1.7e-01	2.4e-02	5.0e-03	2.9e-05
F=2	3.9e-01	3.0e-01	2.1e-02	2.0e-03	1.0e-06
F=4	4.9e-01	3.8e-01	1.4e-02	4.3e-04	1.1e-08

Table 2.2: Distance $dist(F,T)$ for different fan-out F and thresholds T, $K = 30, \lambda = 25$ (top), and $\lambda = 29$ (bottom).

Though the previous distance provides an objective measurement, a better understanding can be gained by analysing the whole state probability distribution function. As the highest deviation from an isolated worker is registered with $d = 2$, [70], we focus on this case. Figure 2.2.4a shows the π's pdf for $K = 30, \lambda = 25$ and different T. The $LL(1,\infty)$ plot is the pdf of the M/M/K/K queue (no load balancing).

From Figure 2.2.4a, we observe how for $T < K$, the state probability after some k starts to decrease fairly sharply, whereas this is not true for node working in isolation or $T = K$. Figure 2.2.4b shows the exact and upper bound of $\widetilde{\pi}_k$, and confirms how the state probability falls sharply with K.



(a) Effect of the threshold on $\pi_k, F = 1$

(b) Upper bound and exact $\widetilde{\pi}_k$ for different thresholds $T, K = 50, \lambda = 25$

Figure 2.2.4

2.2.3 Performance results

This section reports some representative performance results of the $LL(F, T)$ protocol, using the infinite and finite models.

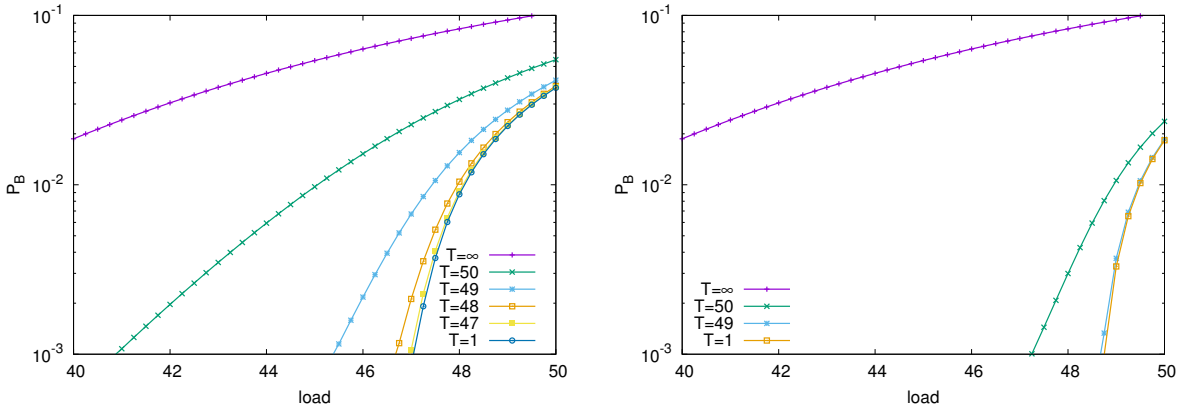
2.2.3.1 Infinity model

Figure 2.2.5a shows the blocking probability as a function of the load for $K = 50, F = 1$. The way the blocking probability changes with the load varies remarkably when load balancing is introduced. The performance can be seen under different angles. For example, suppose that job execution latency is such that they cannot be executed in the cloud[2] and that jobs' blocking probability, called target probability hereafter, should be $p_B \leq 10^{-3}$. Without load balancing, the node can serve up to a traffic load of $\lambda' \approx 33$ requests per unit of time (this value is not shown in the figure), whereas with load balancing with $T = 1$, which is equivalent to a classic Least Loaded policy, $LL(2)$, this traffic raises to $\lambda'' \approx 47$, corresponding to a traffic intensity of $\rho = \frac{\lambda}{K} = 0.74$. To ensure the same p_B the node should increase the number of servers to $K = 67$. We can envision a scenario where the traffic temporarily increases beyond λ' and the node should elastically accommodate this peak. Even assuming a virtualised environment that allows resource scaling, the node can benefit from load balancing during the start-up time of the new 17 resources, which may be not negligible (clearly the traffic of the other nodes should be less than λ'). The plot shows how the same blocking probability can be achieved with $T = 48$.

Another interpretation of this result is the following. Suppose that jobs are never lost as a node can delegate job execution to the cloud if congested. A node working in isolation can serve traffic at a rate of 47 jobs per unit of time, but a fraction of them, corresponding to the blocking probability, roughly 10 % according to Figure 2.2.5a, is executed in the cloud, thus expediency a longer latency. If load balancing is used, this fraction is reduced to just 0.1 %. Clearly, additional latency is to be considered that is due to load balancing penalty. By using a threshold, this latency can however reduced, as discussed next.

[2]Typical values of the round trip time to hit a cloud service can be as high as 100 ms, whereas Fog-to-Fog latency is likely to be as small as a few ms.

Figure 2.2.5b shows the blocking probability when the fan-out is increased to $F = 3$. Now it is enough to set $T = 49$ to get the same fraction of blocked jobs.



(a) $F = 1$ **(b)** $F = 3$

Figure 2.2.5: Blocking probability vs load

Figure 2.2.6a shows the frequency of probing messages generated as a function of the traffic and different T. The advantage of using a threshold is measured by the reduction of probe messages. For $T = 48$ and $\lambda = 47$, the messages rate decreases of a half with respect to an uncontrolled load balancing activity, and much more for lower traffic. For example, for $\lambda = 40$, the target p_B is reached with $T = 50$ at the cost of just a few probe messages. At this traffic, the node should still increase its servers (to $K = 59$) if working in isolation. Besides the obvious benefit of decreasing the control overhead, this reduction also reduces the risk of race conditions that may weaken the effectiveness of load balancing, as remarked in [68], [69].

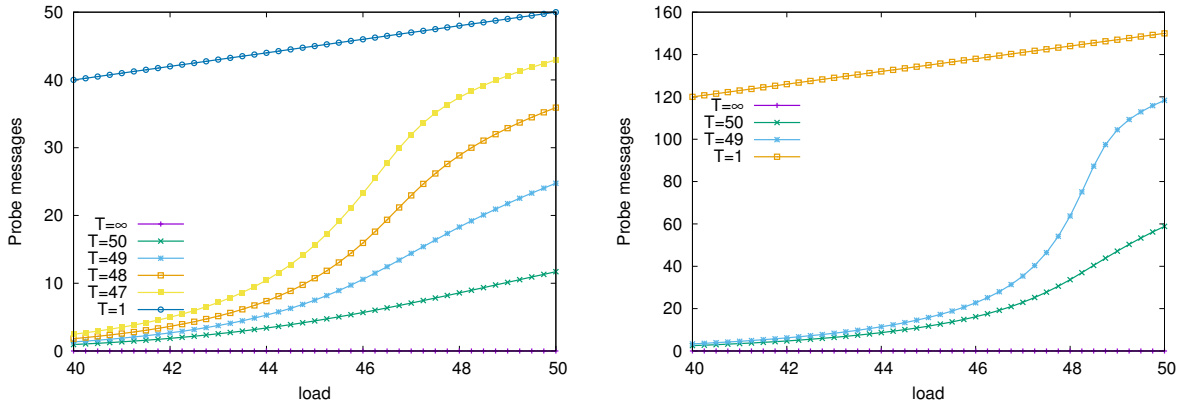
Figure 2.2.6b shows the frequency of probing messages for $F = 3$. Under the overhead point of view, the best fan-out is a matter of the working conditions. For example, if the load is less than $\lambda = 46$ and the target loss fraction is 10^{-3}, then $F = 1$ or $F = 3$ generate almost the same amount of control messages of less than 20 messages per unit of time, i.e. less than a half if threshold is not used. If the load is increased, $F = 3$ may be required to meet the target blocking probability.

Figure 2.2.7a shows the average control delay penalty introduced by the protocol as a function of the traffic. This delay also reduces remarkably using a threshold. For example, from 0.7 to 0.4 for $\lambda = 47$ and to 0.05 for $\lambda = 40$, i.e. of one order of magnitude. Note that for $T = 1$ the average delay is not one, because a job is not always forwarded (the delay is the sum of the average probing message, i.e., 0.5, plus the delay of job forwarding and reply, that occurs only when a job is forwarded.

The delay for $F = 3$ is shown in Figure 2.2.7b. Even under high load $\lambda = 50$ corresponding to $\rho = 1$, using a threshold reduces the control delay. This result corroborates the claim that to achieve the benefit of randomisation it not required to always probe other nodes.

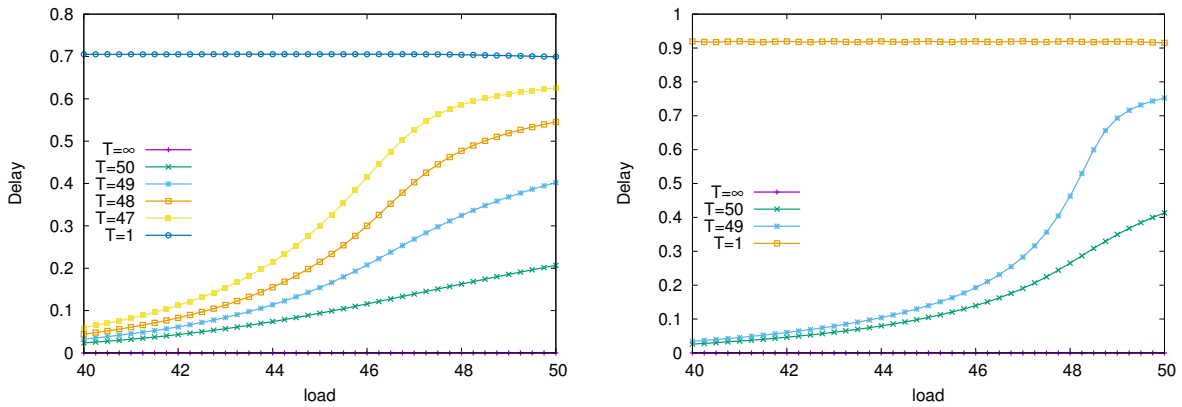
2.2.3.2 Finite model

This section reports the main three metrics, obtained with $N = 8, K = 14$. Figure 2.2.8a shows the blocking probability as a function of the traffic for $F = 1$. Load balancing allows reducing the blocking



(a) $F = 1$ **(b)** $F = 3$

Figure 2.2.6: Probe message frequency vs traffic

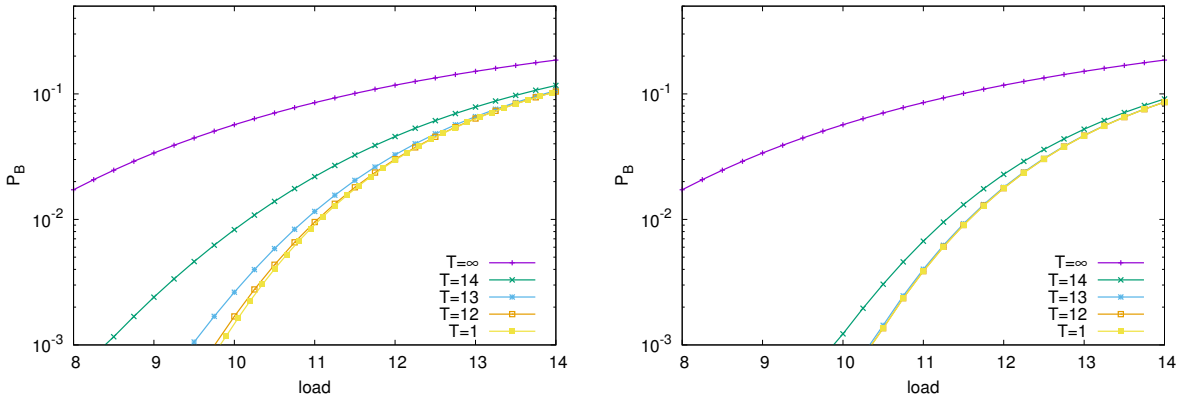


(a) $F = 1$ **(b)** $F = 3$

Figure 2.2.7: Average control delay vs traffic

probability considerably allowing to serve a traffic of up to approximately 10 requests per unit of time for the hypothetical target p_B, corresponding to a traffic intensity of 0.7. The same value is reached with $T = 12$.

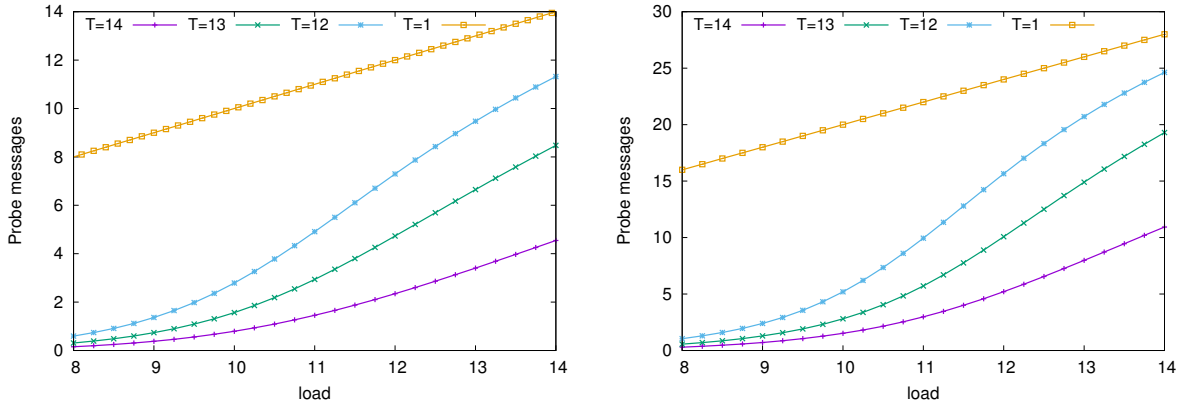
The effect of increasing the fanout is visible in Figure 2.2.8b. Clearly, the margin for improvement is now limited by the finite number of nodes. We can also see see that by setting $T = K - 2$, the protocol reaches the same performance of $LL(d)$.



(a) $F = 1$ **(b)** $F = 2$

Figure 2.2.8: Blocking probability vs traffic

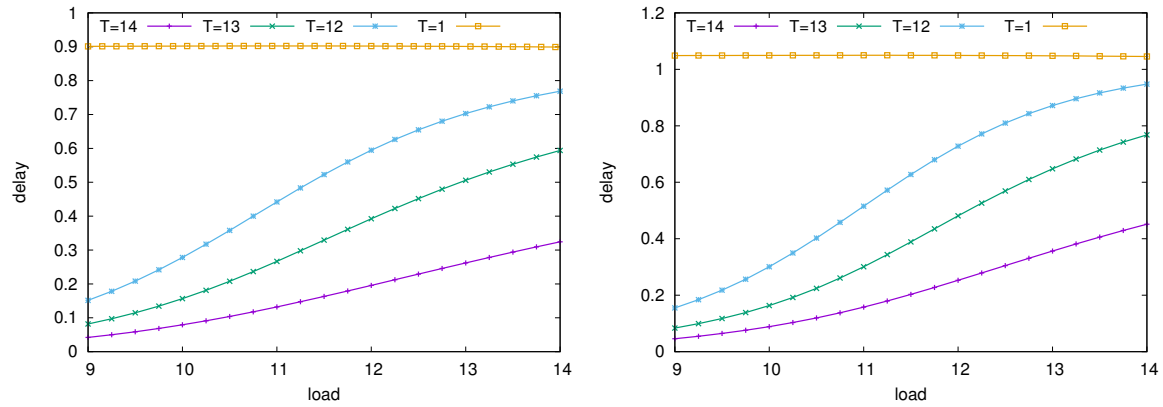
Figure 2.2.9a and Figure 2.2.10.(a) show the benefit of threshold on the control overhead and delay. As for the infinity model, this advantage is even higher for lower traffic.



(a) $F = 1$ **(b)** $F = 2$

Figure 2.2.9: Probe message frequency vs traffic

Figure 2.2.9b and Figure 2.2.10.(b) shows the same performance index when $F = 2$. Similarly to the results obtained from the infinity model, increasing F allows to further reduce the blocking probability, though the improvement is limited by the finite number of nodes.



(a) $F = 1$ **(b)** $F = 2$

Figure 2.2.10: Average control delay vs traffic, finite model

2.2.4 Simulation

In this section, we report the results of a simulated model that considers additional details missed in the mathematical study, concerning the effect of delay and heterogeneity among Fog nodes.

We've used a custom discrete-event simulator, written in Python. The system under observation is composed of N_1 fast Fog nodes and N_2 slower nodes that provide the same service to a set of end users in a given restricted area. Globally, the traffic generated by users covered by the same Fog node is a Poisson process, whereas the average execution of the provided service requires s_1 ms by fast nodes and s_2 by slow nodes[3]. The Poisson assumption can capture a realistic scenario of moving end users entering and leaving areas covered by Fog nodes and requesting an application service, i.e., object recognition for VR/AR applications, as described for example in [63]. The parameters used for simulations are reported in the following Table:

Total Number of Nodes	32
K	14
Traffic rate λ [jobs/s]	35,105,210
Job duration, $\frac{1}{\mu}$ [ms]	300,100,50
Traffic intensity per node $\rho = \frac{\lambda}{K\mu}$	0.75
Task duration on a fast server [ms]	90%
Job Length [MB]	1
Device-to-Fog Delay [ms]	Uniform [5,5.5]
Fog-to-Fog Delay [ms]	Uniform [5,10]
Device-to-Fog Bandwidth [Mbps]	100
Fog-to-Fog Bandwidth [Mbps]	54

Each simulation lasts until at least 3000 loss events are detected. The plots report the average among 5 independent repetitions of a same simulation.

[3]Due to the insensitiveness to the service distribution of loss models, only the average matters.

We have compared the proposed protocol with a centralised round-robin load balancing algorithm, where all Fog nodes first send their jobs to the centralised scheduler (assumed with infinity capacity), then that applies the Round-Robin rule (RR).

Figure 2.2.11 shows the blocking probability as a function of the threshold (left) and the delay for the same traffic intensity $\rho = 0.75$ and different execution times of a task. When the service time of a task is much higher than the latency of control messages, 300 ms corresponding to control delay of about 3% of the service time, the blocking probability follows what has been predicted by the model: as the threshold decreases, it falls sharply to its minimum value and remain unchanged. However, as the execution time becomes comparable with the control delay, after the minimum, the blocking probability increases again. This is especially evident for service time 50 ms that corresponds to control delay 15%. The reason is that exactly because of the control message transmission delay, the state of a remote node at probing time can differ from its state when a job is actually received. It may then happen that the workload of a received Fog node may be higher than the workload of the sending node, hence weakening the effectiveness of the load balancing mechanism. Also, we have noticed from inspecting simulation traces that in some cases, when the state of a node is close to K a job is dropped by the probed node because, differently from what it has reported, the node has no longer idle servers. The left side of Figure 2.2.11 shows that the blocking probability of the RR balancer is higher than the proposed protocol. With the optimal threshold the proposed protocol drops about 0.25% of message, whereas under RR about 2.5%, i.e. one order of magnitude higher. The blocking probability of RR did not changed with the service time (recall that the traffic intensity is kept fixed in the experiment to 0.75).

Figure 2.2.11 also shows the delay (on the right), measured as the time from when a job is generated by an IoT device until the device gets the reply. This value was found to be slightly higher for low thresholds, which is due to the additional job transfer time and probing overheads. The delay of the RR protocol is always higher than LL, hence we can obtain a net advantage since more jobs are served without any delay penalty.

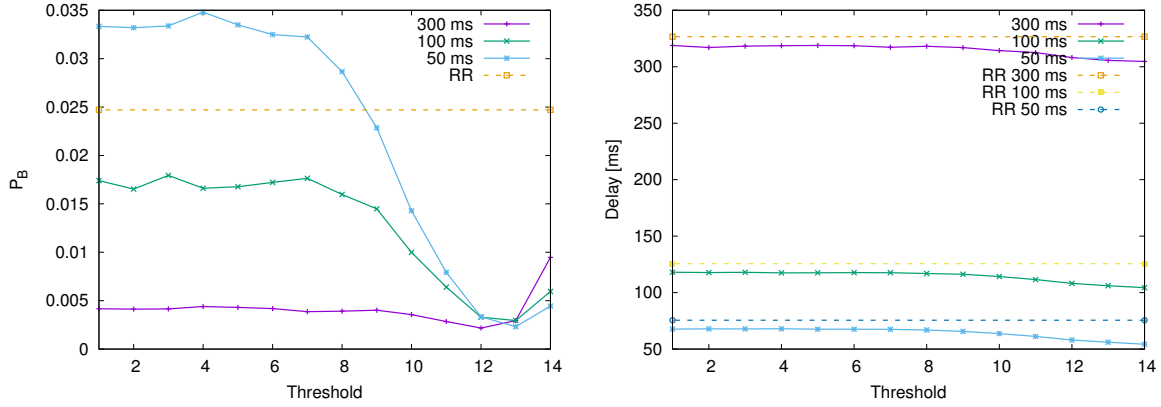


Figure 2.2.11: Impact of the execution time on the blocking probability and delay.

The second set of experiments measured the effect of server heterogeneity. In this experiment half of the nodes execute a job in 300ms (slow node) and the other half in 250 ms (fast node), i.e. half of the nodes are approximately 20% faster. Users are connected to a slow or fast node. The load of a node is

$\lambda = 35$ req/s. Figure 2.2.12 reports the blocking probability as a function of the threshold seen by users that send their job requests to a fast or slow node. The figure also shows the total job's response time, i.e. the time elapsed from when a device sends a job execution request until it gets the reply. We can see that the effect of the threshold is still effective in case of server heterogeneity, namely a threshold of $T = 12$ provides similar results of the homogeneous case. Since the load balancing allocates jobs to random servers, the response time for jobs coming from users connected to a fast node is higher than 250 ms, i.e., the execution time when executed on fast nodes, time because it may occur that a job is executed on a slower server. The advantage lays in the higher number of served jobs. For the same reason, the response time seen by users connected to a slow node becomes lower than when jobs are not forwarded to any faster node.

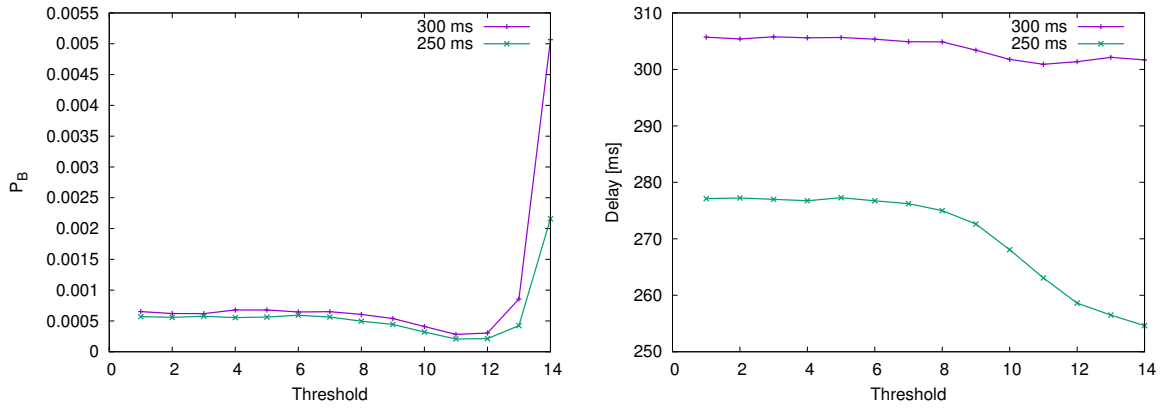


Figure 2.2.12: Performance for the heterogenous case.

2.2.5 Implementation

An experimental test of the proposed scheduling algorithm has been conducted on a framework, dubbed *P2PFaaS*, that we conceived and designed according to the findings described in the previous sections. Each Fog node runs an instance of such a framework, see Figure 2.2.13. The framework relies on the Function-as-a-Service model (FaaS), [71] according to which the service provided by a Fog node is exposed as a stateless function. The exported functionality is an image detection service that is provided by all the Fog nodes, [4]. Function invocation takes an image as input and it returns the coordinate of a rectangle containing a face in the passed image. The invocation of such a function is bounded to a specific image and corresponds to the unit of execution, called job or task hereafter. Job transfer corresponds to transferring the image to be processed to another Fog node.

The framework, that is completely written in *Go*, is composed by a discovery service, which allows nodes to know each other, and a scheduler service in which the core of the scheduling policy resides. In particular, the scheduler service can forward image detection requests to other nodes, do probing or schedule the function execution to the current node. The framework implements only the scheduling

[4]The function that implements this face detection is the Pigo Face Detector (`https://github.com/esimov/pigo/`) function and implements the Pixel Intensity Comparison-based Object detection that is a modification of the standard Viola-Jones method

logic since the actual function execution is delegated to OpenFaaS [5]. Both OpenFaaS and the proposed framework rely on Docker and Docker Swarm, for which every node represents a swarm/cluster with only one node. This one-to-one mapping is done in order to avoid to use the Docker Swarm pre-built scheduler, which always assigns a new job to the least loaded node among the cluster it manages. To avoid conflicting decisions, OpenFaaS auto-scaling is disabled and the maximum number of concurrent functions is set to $K = 10$.

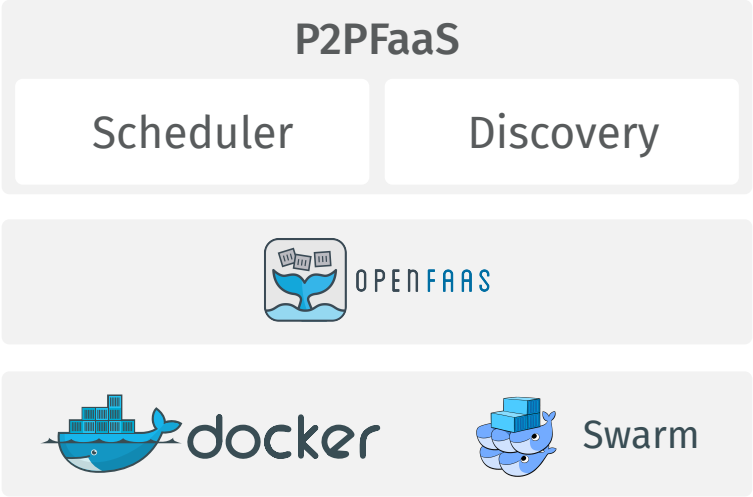


Figure 2.2.13: P2PFaaS concept diagram, illustrating the complete stack of services

A client that needs to perform a face detection task, sends an HTTP request to a Fog node. When the node receives the request, the scheduling policy is applied: the current number of running functions is checked and if it is below the threshold T the request is immediately delegated to OpenFaaS, which executes the face detection function. In the other case, when the current load is equal to or above the threshold T, the scheduler service picks F node IDs at random and probes their load via parallel HTTP requests. When all the replies have been collected, the scheduler decides where to schedule the request, and if it is forwarded to another node, it performs another HTTP request. When this node completes the execution the result is returned via HTTP response to the origin node which returns it to the client. Again the client waits for the job completion and it executes only one HTTP request, all what happens behind it is totally transparent to the client.

An important optimisation that has been introduced since the initial concept version of the framework regards probing. Indeed, after some tests, it resulted that serialising a JSON for replying to a probe, with node load information, is too demanding to be performed since it requires a considerable amount of CPU time. For circumventing this problem, load information is now passed via HTTP headers and this allowed to drop the average probing time from 40ms to 10ms.

[5] `https://openfaas.com`

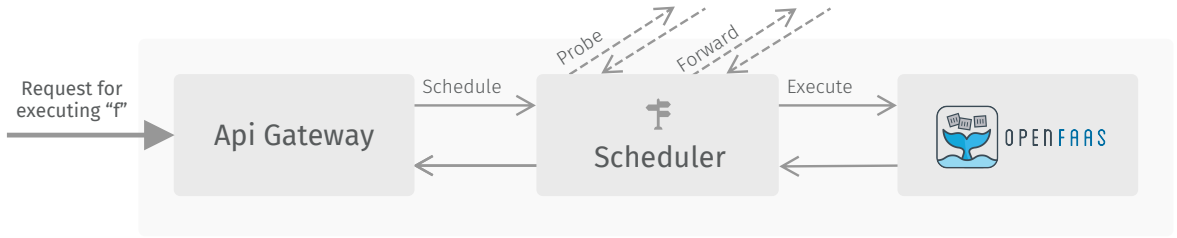


Figure 2.2.14: Scheme of execution of a function in the P2PFaaS framework

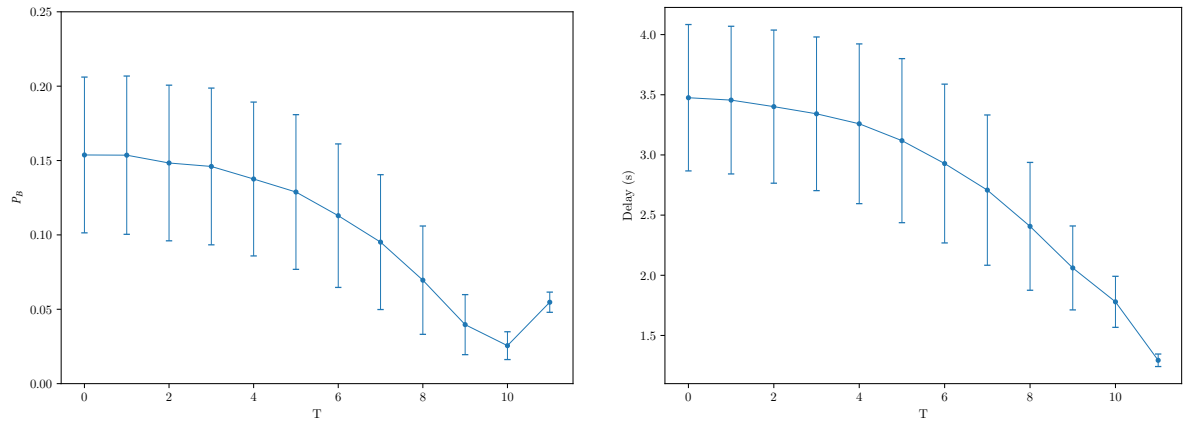
2.2.5.1 Results

We have conducted several tests by exploiting a cluster of two servers equipped with Intel Xeon @ 2.80 GHz, which are used to instantiate 8 VMs with assigned 1 core, 3GB of RAM and with Debian installed. Every machine has been equipped with Docker, OpenFaaS and the proposed framework with $K = 10, F = 1$, and they have been set up as master nodes of single-node Docker Swarms. Then a ninth VM, the "benchmarker", has been instantiated in order to generate the traffic of the requests and collect all the data. All VMs are connected via fast ethernet within the same local network. A series of Python scripts generates parallel traffic of image recognition requests to every machine, then they collect the number of dropped requests, the average execution time of a function and a series of other parameters that regard probing times, forwarding times, number of http errors and many others. The average execution time of a single image recognition is 300ms. Each experiment consisted of sending 20.000 detection requests at rate $\lambda = 3.00$, thus having $\rho = 0.9$.

The experiment has been repeated 7 times, due to its duration ($\approx 24\text{hrs}$), and in the following figures, results are shown by using a confidence interval with $\alpha = 0.1$ and with sample mean error of $\pm\, t_{\frac{\alpha}{2},n-1}\sqrt{\frac{S^2}{n}}$ (where S^2 is the sample variance).

Figure 2.2.15a shows the estimated blocking probability (ratio of dropped requests to the total number of requests generated). This experiment shows a minimum for $T = 8$. As predicted by the theoretical model (see Figure 2.2.8a), the blocking probability drops sharply as T decreases; however, rather than remaining almost constant at that value it starts to increase when T is further reduced. The reason is that when T is lowered, the workload due to job scheduling at each Fog node increases since the number of probes per job increases. In the limit of $T = 1$, unless the Fog node is idle, $every$ job arrival triggers a probe-reply cycle. While the length of such control messages is overall negligible, their processing is not and it has the net effect of reducing the CPU cycles allocated to the image detection or, equivalently, to increase the duration job execution. This aspect is not captured by the model. The reduced CPU execution speed clearly increases the average execution time of served job, as reported in Figure 2.2.15b.

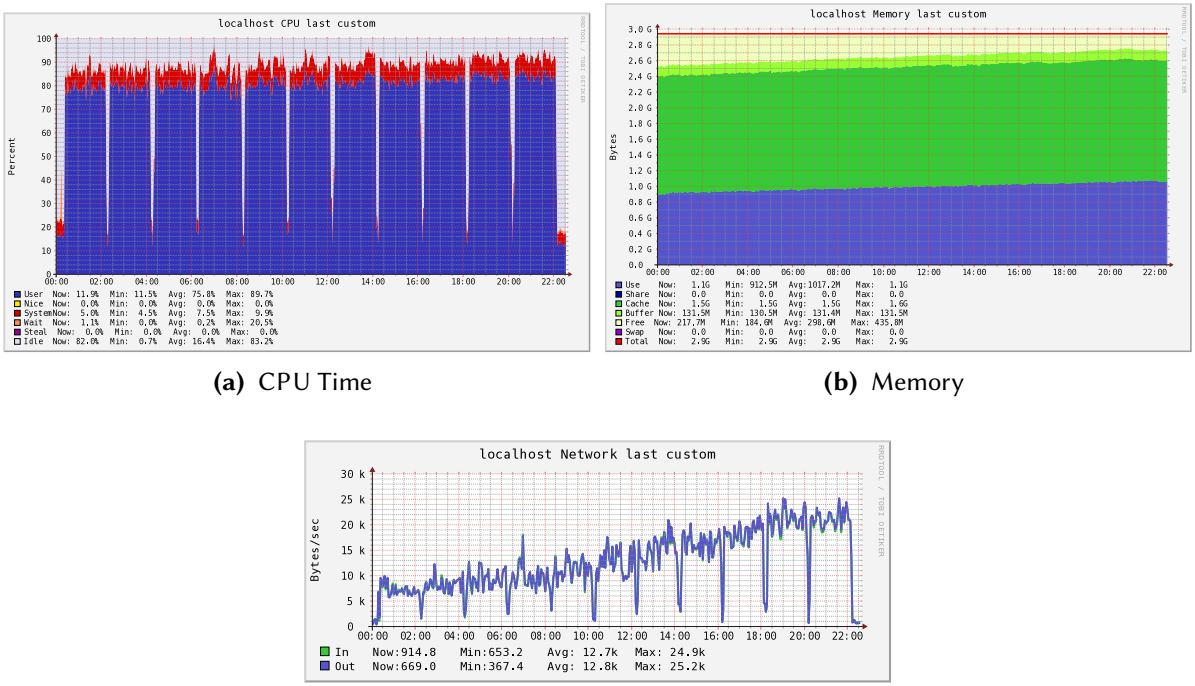
Finally Figure 2.2.16 reports the output of the RRDTool performance profiler used during the trials, showing the CPU breakdown, total memory usage and control traffic of a Fog node. Each pause in the trace corresponds to decreasing the threshold of one unit, starting from $T = 10$. We can see how the CPU usage slightly increases as T decreases and is approximately 0.9, which is consistent with the nominal generated traffic intensity, $\frac{\lambda}{\mu}$. Such an increase is due to an increase in the control message processing, as outlined above. Neither the memory nor the network is saturated, although they both



(a) Blocking probability as a function of the threshold **(b)** Average end to end delay as a function of the threshold

Figure 2.2.15

increase as the protocol becomes more proactive.



(a) CPU Time

(b) Memory

(c) Network Usage

Figure 2.2.16: Performance profile during a test. The threshold is varied approximately every 2 hours

2.3 Performance comparison between blind and random forwarding approaches

The reference model for this study is provided in Fig. 2.3.1, with Fog nodes densely distributed in a given geographic area that provides a service of data processing for a plethora of sensors, e.g., IoT sensors or image processing in a smart city application, using a 5G Fog Radio Access Network (F-RAN) architecture [72]. See [73] and [63] for a more in-depth description of computer vision-based services and smart cities infrastructures. It is worth to note that, even if in this study the reference scenario is focused on a smart city application, the main findings of the proposed research have general validity and can be applied to other application fields with similar characteristics.

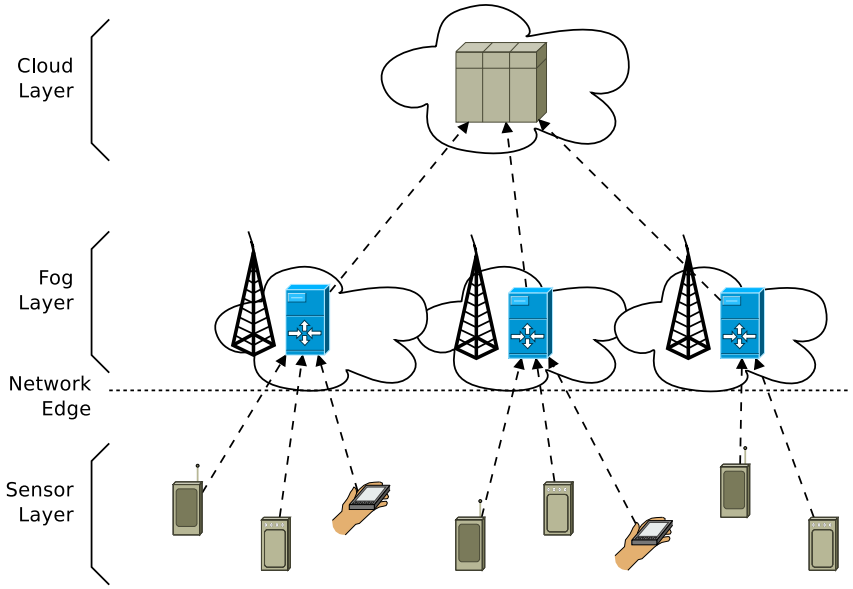


Figure 2.3.1: A typical Fog computing deploy model.

Among the several challenges introduced by the Fog computing (see [74] for a general discussion), in order to address the above upcoming scenario, we focus on a specific research topic currently under investigation: the design of an algorithm for resource sharing among uncoordinated and heterogeneous Fog nodes in order to improve the response time of smart cities applications. Although resource sharing is a classical and well-studied topic in the computer science community, this model of Fog computing does not fit all the assumptions of the studies available in the literature. In particular, the following elements are new to the Fog deployment: (i) the heterogeneity among the elements of the infrastructure; (ii) the execution time of a job that is comparable to the time required to transfer the job from the node of origin to another node; (iii) the absence of a centralised entity that acts as a load balance.

This study introduces two load balancing algorithms, namely *sequential forwarding* and *adaptive forwarding*, designed to take these peculiarities into account. In particular, in this study we place a major emphasis on the heterogeneity aspect of the problem, considering that Fog nodes are characterised by different computing power and may receive different workload intensities. The workload consists of jobs that are continuously generated from end devices (on-line load balancing). The basic idea of the

proposed algorithms is the followingwe assume that the Fog computing layer provides an elementary service to end-users, e.g., consisting in object detection inside a video frame [63]. As jobs reach the Fog nodes, the nodes estimate the expected waiting time (based on the number of jobs already being processed by the node). If the waiting time exceeds a threshold Θ (that may change depending on the Fog node and may be self-tuning), the Fog node forwards *blindly* at random the job to another Fog node, that executes the same decision algorithm. By this, we mean that the node doesn't keep or probe any information about the current state of the other nodes, but rather picks one of the nodes it is aware of, uniformly at random. Decisions are memory-less, except for the number of forwarding (steps) already done, which is carried in the message. The steps are upper bounded by a parameter M. At the M-th forwarding, the receiving Fog node will process the job without further attempts, unless its processing queue is full, in which case the job will be dropped.

Any of the two algorithms fit the three challenges of Fog computing because (i) it takes explicitly into account uneven load distributions and heterogeneous node characteristics and configurations parameters, e.g., lightly loaded nodes do not forward their jobs often (where the load is normalised to the actual node execution speed) (ii) it places a significant effort in limiting the number of (potentially expensive) transmissions between nodes by adapting Θ, (iii) it is completely distributed since forwarding decisions are local, uncorrelated and autonomous.

The contribution of this study can be summarised as follows:

- definition of a lightweight randomized on-line distributed load balancing algorithm suitable for scenarios characterised by *independent* providers and *heterogeneous* load conditions, along with a variant based on a *self-tuning* mechanism;
- mathematical analysis and experimental evaluation of the algorithms on a realistic scenario, showing evidence of significant improvements compared to unbalanced nodes. Throughout the sensitivity analysis, it is found that the loss rate of the proposed algorithms is in most cases 15 to 19 times lower than the case where no cooperation is used. In a similar way, the response time is reduced by 19% to 11% in most scenarios. In the realistic setup the proposed algorithms provide an even more impressive performance gain with a reduction in the loss rate from 13% to 0.2% and a response time nearly halved;
- through the tests we point out how a self-tuning mechanism can provide robust performance requiring limited tuning of the algorithms' parameters and we provide some insight into how the characteristics of a heterogeneous infrastructure impact on the algorithms' parameters.

2.3.1 Sequential forwarding algorithms

We now discuss two proposed algorithms for load balancing, which are variants of the same central idea. This key idea is to allow Fog nodes to make autonomous and uncoordinated decisions about serving or offloading a job. This decision is promptly taken on a per-job basis without the need of any coordination, e.g., a centralised entity or information about the state of other nodes. This is a distinctive feature of the proposed design that in fits a general Fog computing model, where Fog nodes can be heterogeneous and can undergo different local management rules. In order to explain the load balancing algorithms, we refer to the architecture in Fig. 2.3.1, where a set of sensors send jobs to a layer of Fog

nodes. Each sensor communicates with one Fog node. The sensor to Fog node mapping can be based on geographic distance as in [38] or can exploit a more complex algorithm [39]. The workload of each Fog node is typically heterogeneous, for the twofold reasons of time-dependent fluctuations in the workload patterns or due to uneven distribution of sensors among the Fog nodes [75]. A similar heterogeneity can occur also in the characteristics of the Fog nodes as the deployment of the infrastructure starts with a small prototype implementation that is then expanded over time with more modern Fog nodes. Furthermore, Fog nodes may belong to different providers that may adopt their own management rules or technology, e.g., containerisation implementations based on Docker Swarm or Kubernetes from one hand, or based on VMs on the other hand. For the sake of this proposed approach we assume that the Fog nodes expose some standard high-level interface through which they may forward jobs, e.g., HTTP endpoints, masking the actual network layer solution[6]. All nodes know the communication endpoints of the other nodes.

The proposed contribution consists of two algorithms aiming to define *when* a job should be forwarded to a neighbour, and *to which* neighbour the job should be forwarded. We start the presentation with a *Sequential Forwarding* algorithm; nextly an evolution of this algorithm is described, namely *Adaptive Forwarding* algorithm. Finally, we discuss a baseline algorithm, namely *No LB*, that is the case where no load balancing occurs among the Fog nodes.

2.3.1.1 Sequential Forwarding algorithm

The *Sequential Forwarding* algorithm uses a threshold Θ_n for each Fog node n to decide if an incoming job should be forwarded to a random neighbour or not. The threshold operates on the system load, which is the number of jobs queued in the Fog node (or being executed). The system load represents an estimate of the waiting time for the incoming job. An additional parameter of the algorithm is the maximum number of steps M to guarantee a limit on the delay associated with the load balancing phase.

Algorithm 1 Sequential Forwarding Algorithm

```
Require: M, Θ_n, Job
  if Job.Steps() < M then
    if System.Load() ≤ Θ_n then
      ProcessLocally(Job)
    else
      Neigh ← Random(System.neighbours())
      Job.IncrementSteps()
      Forward(Job, Neigh)
    end if
  else
    ProcessLocally(Job)
  end if
```

Algorithm 1 presents the proposed load balancing mechanism. When a job arrives, if the job has not yet reached the M-th step, the system load (that is the number of jobs already scheduled for processing

[6]For example, the X2 interface allows direct communications among 5G nodes.

in the Fog node) is considered. If the value does not exceed the threshold Θ_n, the job is accepted and scheduled for local processing. Otherwise, it is forwarded to a randomly-selected neighbourwe point out two main features of the proposed algorithm that are (i) the *blind* and *memoryless* nature of the algorithm so that no probing for the neighbour status nor reservation (to make sure that the job finds the resource available [69]) is required; and (ii) the ability of the algorithm to adapt to heterogeneous scenarios and to operate in a completely distributed way thanks to the per-node threshold Θ_n. If the job has already been forwarded M times, it is scheduled for local processing. This makes the algorithm extremely simple to implement and facilitates its adoption among different providers.

We also detail the local processing of the job, as this task is also responsible for the drop of the job if the queue is full, as pointed out in Algorithm 2. It is worth to note that dropping a job in the case the queue becomes too long may be an extreme measure undesirable for some applications. In this analysis we prefer to focus on a simplified scenario that is easy to model and to implement rather than considering multiple queuing and dropping behavior depending on the application. Such evolution of the algorithms can be an additional extension of the present research that is left as future work.

Algorithm 2 Local processing: *ProcessLocally*()

Require: Job
  if System.*Queue*() $<$ System.*MaxQueue*() **then**
    Enqueue(Job)
  else
    Drop(Job)
  end if

2.3.1.2 Adaptive Sequential Forwarding algorithm

The Sequential Forwarding proposed in Sec. 2.3.1.1 has two separate parameters, Θ_n and M, that show an inherent inter-dependence: indeed, if Θ_n is low, we may have a high number of forwarding, so M may play a pivotal role. This makes the algorithm tuning complex, especially in heterogeneous scenarios, where the thresholds may be different across the infrastructure. To address this problem we introduce an adaptive version of the algorithm.

The Adaptive Sequential Forwarding algorithm (*Adaptive Forwarding* for short), is an evolution of the Sequential Forwarding proposed in Sec. 2.3.1.1 that introduces some self-tuning ability.

Algorithm 3 Adaptive Forwarding Algorithm

Require: M, Job
  $Q \leftarrow$ System.*MaxQueue*()
  $\Theta \leftarrow \lceil$Job.*Steps*() * $Q/M \rceil$
  SequentialForwarding(M, Θ, Job)

Algorithm 3 shows the behavior of the Adaptive Forwarding Algorithm. The threshold Θ_n is computed in the same way for every node and grows linearly with the number of steps. If a job has never been forwarded (or has been forwarded just a few times), the job is not processed locally unless

the local load is very low. On the other hand, if a job has already been forwarded several times we assume a more relaxed attitude towards the search of a Fog node with a low load.

The algorithm starts requesting the maximum queue length of the Fog node; next the algorithm computes the threshold Θ_n that grows linearly with the number of times the job has been forwarded. In particular, we tune the growth of the threshold in such a way that, after M steps, the threshold Θ_n is equal to the maximum queue length of a Fog node, thus guaranteeing that the job will be accepted unless this last visited node has no room in its queue.

2.3.1.3 Baseline algorithm

In the performance evaluation, we consider also the *No LB* algorithm, that is the case where no load balancing occurs, as a baseline.

Considering the previously described algorithms, and given Q as the maximum length of queue (as in Algorithm 3), the behavior corresponds to the case where $\Theta_n > Q$ or to the case where $M < 1$we expect this algorithm to suffer from a high loss rate (that is jobs dropped because the queue is full), unbalanced load (especially in scenarios with heterogeneous load distribution among the Fog nodes) and, generally, poor performance.

2.3.2 Model

We start studying the algorithm under an ideal deployment composed by an infinite number of nodes and for $M = 1$. The case $M > 1$ can be incorporated in the proposed framework, but for the sake of clarity, this extension is left as future work. To capture heterogeneity, we assume that two types of Fog nodes exist, type A and type B.

We define as α (β) the probability that a class A node (class B node) forwards a job to a node of the same class. Any Fog node is abstracted as a FIFO queue with the same bounded number of places Q_A (Q_B), included the server. Class A (B) nodes get a nominal Poisson flow of jobs at rate λ_A (λ_B) jobs/s from directly connected users, while the service time of a job is exponentially distributed with an average processing rate of μ_A (μ_B).

The system under investigation is composed of two tagged nodes $a \in A, b \in B$ plus a set of N_A class A nodes and a set of $N_B = N_A = N$ class B nodes. We derive the main performance metric of the load balancer in the limit of $N \to \infty$. The reason for this limiting study is that it assumes independence among nodes, a desirable property that holds for simple homogeneous FIFO queues [76]. Let then assume that all the nodes of the system are independent from each other. The effect of the N nodes on a and b is equal to the probability of node a (b) being in a given state, since it represents the fraction of the N_A (N_B) nodes in the same state.

Without loss of generality, we now focus on node a. Due to the symmetry, the results for node b are the same, except for the fact of swapping labels A with B and α with β. In the following derivations, we assume that a node is in a state i when it has i tasks in its queue.

Node a may receive jobs forwarded by other nodes of the same class or from the other class with rate:

$$\lambda_{F_A} = \alpha\lambda_A\tilde{\pi}_{A\Theta_A} + (1-\beta)\lambda_B\tilde{\pi}_{B\Theta_B} \tag{2.4}$$

where $\tilde{\pi}_{Ai} = \sum_{j=i}^{Q_A} \pi_{Ai}$ ($\tilde{\pi}_{Bi} = \sum_{j=i}^{Q_B} \pi_{Bj}$) is the probability that a node of class A (B) is in a state higher or equal than i, indeed π_{Ai} (π_{Bi}) is the steady state probability of a node of class A (B) to be in state i. In fact, due to independence among states, $N_A \lambda_A \tilde{\pi}_{A\Theta_A}$ is the rate at which N_A class A nodes forwards job. And one of this job hits a with probability $\frac{\alpha}{N_A+1}$: the job selects nodes of the same class with probability α and picks exactly a with probability $\frac{1}{N_A+1}$. For this reason, the first term in the above expression represents the flow of jobs seen by a and coming from nodes of the same class, in the limit of $N \to \infty$. The second term has a similar interpretation.

The transition rate from the state i to $i+1$ is:

$$\lambda_{Ai} = \begin{cases} \lambda_A + \lambda_{F_A} & i \leq \Theta_A \\ \lambda_{F_A} & i > \Theta_A \end{cases} \tag{2.5}$$

The steady state probability distribution satisfies the following standard linear set of equations:

$$Q_A \pi_A = [0, \ldots, 1]^T \tag{2.6}$$

where:

$$Q_A = \begin{pmatrix} -\lambda_{A_0} & \mu_A & 0 & \ldots & 0 \\ \lambda_{A_0} & -(\lambda_{A_1} + \mu_A) & \mu_A & \ldots & 0 \\ & \ddots & & & \\ 0 & \lambda_{A_{i-1}} & -(\lambda_{A_i} + \mu_A) & \mu_A & \ldots & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{2.7}$$

The solution is found numerically as follows. The linear system is first solved using the matrix Q_A^0 with $\lambda_{Ai} = \lambda_A$. An analogous system of equations is solved using Q_B^0, where $\lambda_{Bi} = \lambda_B$. These solutions exist because they define two independent $M/M/1/Q$ Markov Chains. From these solutions, Eq. (2.4) (and the analogues for B) are used to compute the traffic flows for two new pair of matrix, say Q_A^1 and Q_B^1. The algorithm continues until $max\{||Q_A^n - Q_A^{n-1}||\} < \epsilon$ and $max\{||Q_B^n - Q_B^{n-1}||\} < \epsilon$.

2.3.2.1 Metrics

In this section, we derive the main performance indicator of the algorithm. Due to the symmetry we keep the node a point of view. Any metric concerning b is the a's one where B, A and α, β are swapped.

Blocking probability The first metric of interest is the probability that a job is blocked p_B, namely the fraction of jobs that cannot be served by any Fog node. As these jobs are dropped by the Fog nodes, this metric is also referred to as loss rate or drop rate. The value of this metric is given by:

$$p_B = \frac{p_{B_A} + p_{B_B}}{2} \tag{2.8}$$

where the two contributions are the probability that a job received by a class A (class B) node is blocked, which are given by:

$$p_{B_A} = \tilde{\pi}_A [\alpha \pi_{A_{Q_A}} + (1-\beta) \pi_{B_{Q_B}}] \tag{2.9}$$

Response Time The second relevant metric is the average response time, T, namely the time elapsed from when a job is received by a node until the serving nod ends to execute the jobwe do not consider the delay due to a possible reply to the originator of the job. This quantity is due to the queueing waiting time (W), execution time ($\frac{1}{\mu}$), and job forwarding delay. Concerning the fist contribution, the average queue length of the node a is:

$$\overline{Q}_A = \sum_{k=1}^{Q_A} k\pi_{A_k} \tag{2.10}$$

The net flow of jobs entering in a is the sum of jobs from:

- users connected to a, at rate $\lambda_A(1-\tilde{\pi}_{A\Theta_A})$
- class A nodes with state above Θ_A selecting a, at rate $\tilde{\pi}_{A\Theta_A}\lambda_A\alpha(1-\pi_{Q_A})$
- class B nodes with state above Θ_B selecting a, at rate $\tilde{\pi}_{B\Theta_B}\lambda_B(1-\beta)(1-\pi_{Q_A})$.

Hence, applying the Little's result the queue's waiting time at class A nodes is:

$$W_A = \frac{\overline{Q}_A}{(1-\pi_{Q_A})(\lambda_A(1-\tilde{\pi}_{A\Theta_A}) + \lambda_A\tilde{\pi}_{A\Theta_A}\alpha + \lambda_B\tilde{\pi}_{B\Theta_B}(1-\beta))} \tag{2.11}$$

Since a not blocked job arriving to a class A node is served either by a class A node (hence experiencing S_A) or a class B node (hence experiencing S_B), the average response time of jobs received by class A nodes is a weighed average of the two service delays plus the average time spent to forward a job:

$$T_A = \frac{P_{S_{AA}}\left(\frac{1}{\mu_A}+W_A\right) + P_{S_{AB}}\left(\frac{1}{\mu_B}+W_B\right)}{P_{S_{AA}}+P_{S_{AB}}} + \tilde{\pi}_{A\Theta_A}\delta \tag{2.12}$$

where $P_{S_{AA}}$ ($P_{S_{AB}}$) is the probability that the job is served by a class A (class B) node, $\tilde{\pi}_{A\Theta_A}$ the probability a job is forwarded, and δ the average forwarding time. The probability $P_{S_{AA}}$ takes into account that the fact that the job can be forwarded and served by another class A node or directly served by a:

$$P_{S_{AA}} = \tilde{\pi}_{A\Theta_A}\alpha(1-\pi_{A_{Q_A}}) + (1-\tilde{\pi}_{A\Theta_A}) \tag{2.13}$$

while $P_{S_{AB}}$ is:

$$P_{S_{AB}} = \tilde{\pi}_{A\Theta_A}(1-\alpha)(1-\pi_{B_{Q_B}}) \tag{2.14}$$

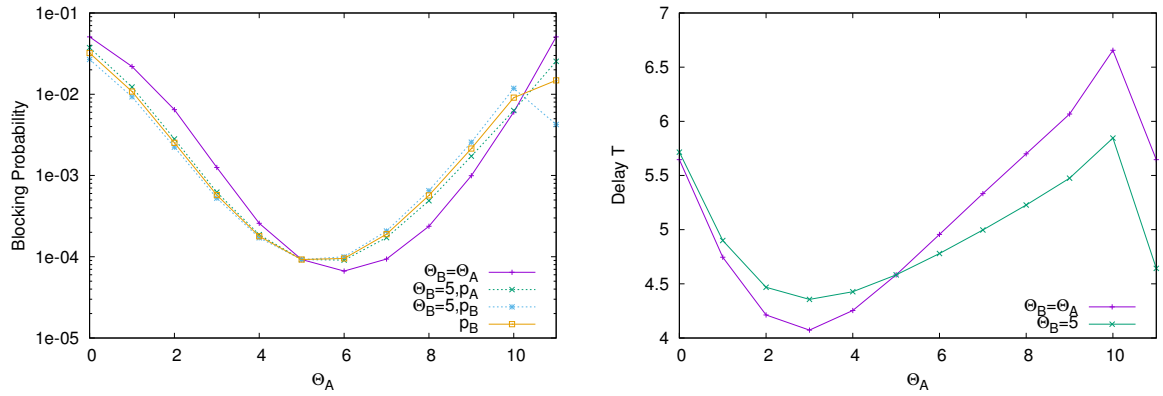
Finally:

$$T = \frac{T_A + T_B}{2} \tag{2.15}$$

2.3.2.2 Result

In this section,we report some representative results obtained from the proposed model, when $\alpha = \beta = 0.5$. Fig. 2.3.2 reports the blocking probability and the response time as a function of Θ_A for the homogeneous case, i.e., when nodes have the same speed. The optimal threshold that minimises the blocking probability is when $\Theta_A = \Theta_B = 6$ (as shown in Fig. 2.3.2a), i.e., roughly half of the total queue

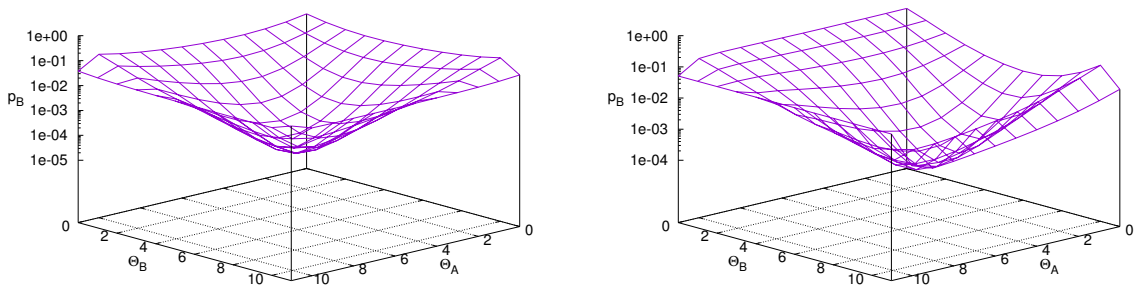
length, which is also an intuitive result: if the current length is too small, there is also a small chance for the job of landing on a less loaded queue, whereas if the length is too high load balancing doesn't arise since the job cannot return to the original node. The lowest response time is however obtained for the different threshold value $\Theta_A = \Theta_B = 3$, as shown in Fig. 2.3.2b. The reason is that nodes now drop more jobs and hence the queue length is shorter. Reducing the threshold further will also progressively eliminate the load balancing effect and hence the average queue length increases again. Both figures also show a representative case of thresholds tuned differently, in particular when $\Theta_B = 5$ while Θ_A is free to vary.



(a) Blocking probability **(b)** Response time

Figure 2.3.2: Performance metrics vs Θ_A, same traffic and service times.

Figure 2.3.3a shows the blocking probability for the full combination of thresholds for the homogeneous case and Figure 2.3.3b for when the speed of class B node is twice the A's one. We can see how there is still an advantage despite the heterogeneity among nodes, and that the best threshold combination is now slightly different.



(a) $\mu_B = \mu_A$ **(b)** $\mu_B = 2\mu_A$

Figure 2.3.3: Loss rate vs. thresholds, traffic intensity $\rho = 0.9$

To better explore the range of applicability of the proposed protocol, Tab. 2.3 reports the highest

λ_A	λ_B	μ_A	μ_B	ρ_A	ρ_B	Θ_A^*	Θ_B^*	p_{B_0}	p_B^*	T_0	T^*
0.85	0.85	1.00	1.00	0.85	0.85	6	6	3.55e-02	5.12e-06	5.22	4.52
0.90	0.90	1.00	1.00	0.90	0.90	6	6	5.08e-02	6.67e-05	5.65	4.96
0.95	0.95	1.00	1.00	0.95	0.95	6	6	6.94e-02	1.18e-03	6.08	5.65
0.99	0.99	1.00	1.00	0.99	0.99	7	7	8.64e-02	1.06e-02	6.42	7.02
0.85	0.90	1.00	1.00	0.85	0.90	6	6	4.31e-02	1.85e-05	5.43	4.72
0.95	0.90	1.00	1.00	0.95	0.90	6	6	6.01e-02	2.71e-04	5.86	5.25
0.99	0.90	1.00	1.00	0.99	0.90	6	6	6.86e-02	9.05e-04	6.03	5.56
0.90	0.90	**0.90**	1.00	1.00	0.90	6	7	7.09e-02	1.12e-03	6.43	6.13
0.90	0.90	**1.10**	1.00	0.82	0.90	6	6	3.91e-02	8.66e-06	5.07	4.38
0.90	0.90	**1.20**	1.00	0.75	0.90	6	5	3.28e-02	1.74e-06	4.66	3.81
1.80	0.90	**2.00**	1.00	0.90	0.90	7	5	5.08e-02	1.12e-04	4.23	3.65
2.70	0.90	**3.00**	1.00	0.90	0.90	8	5	5.08e-02	2.61e-04	3.76	3.33
3.60	0.90	**4.00**	1.00	0.90	0.90	8	4	5.08e-02	4.97e-04	3.53	2.96

Table 2.3: Optimal thresholds that minimise the blocking probability.

possible reduction of the blocking probability under a variety of conditions, obtained by setting the thresholds to the values Θ_A^*, Θ_B^* that minimise p_B. The results are divided into four groups. In the first one, the offered traffic changes for all nodes in the same way and nodes are also equal in terms of execution speed. i.e., the system is homogeneous. The result confirms that the best threshold is almost half of the queue length. Load balancing can reduce up to four orders of magnitude the blocking probability for moderate traffic with respect to the case in which there is no cooperation, reported in column p_{B_0}, and it allows reducing at the same time the average delay (see T^* and T_0) for all cases but the high load case. As the load increases to 0.99 in fact, while there is still an improvement in terms of p_B the average response time is higher compared to no cooperation just because more jobs are queued. Moreover, the table reports the result of three heterogeneous cases, where nodes are different because type A nodes: (i) get a different traffic flow, (ii) have a different execution time, (iii) have different traffic and execution time but their traffic intensity $\rho = \frac{\lambda}{\mu}$ is constant. The changed parameter is given in bold. For all scenarios, the algorithm is able to reduce the blocking probability of at most two orders of magnitude. The response time T^* is also always lower than the no cooperative case.

Once the analysis provided a generally positive assessment concerning the expected behavior of the protocol, we are now ready to consider more realistic cases, with a finite number of nodes and higher M.

2.3.3 Simulation Results

In the present section, we evaluate the performance of the proposed algorithms relying on a simulation approach. Throughout these tests we will consider the Sequential Forwarding algorithm (*Seq Fwd* described in Sec. 2.3.1.1), the Adaptive Forwarding alternative (*Adapt Fwd*, Sec. 2.3.1.2) and the case where no load balancing occurs (*No LB*). We start providing a summary of the tests carried out, presenting the reference experimental scenarios. Next, we consider a simplified scenario where we have two classes

of Fog nodes, namely A and B, characterised by different configurations and, possibly, by different computing power. In this scenario we first validate the simulation results against the numerical model presented in Sec. 2.3.2 and, next, we discuss the main findings of the simulation-based performance evaluation. After this preliminary study, we carry out a thorough sensitivity analysis with respect to the main parameters that may determine a scenario heterogeneity, regarding different computational power of the Fog nodes and different distribution of A and B Fog nodes populations. Finally, we focus on a realistic geographic setup for a smart city and we evaluate in detail both the Sequential Forwarding and the Adaptive Forwarding algorithms.

2.3.3.1 Scenarios definition

The first scenario used in the experiments is named *simplified scenario*. We model the Fog nodes as M/M/1/Q queuing systems, with an exponential distribution of both incoming jobs from the sensors and job service in the Fog nodeswe consider two populations of Fog nodes, namely A and B characterised by a different processing power such that $\mu_A \geq \mu_B$, with $\mu_B = 1.0$ jobs/sec. Let N_A be the number of nodes of class A and N_B be the number of node of type B. Concerning the workload, we consider that every Fog node receives the same workload intensity $\lambda_A = \lambda_B = \lambda$. To make sure that the global load on the Fog infrastructure is $\rho = 0.9$, we consider that $\rho = (N_A + N_B)\lambda/(N_A\mu_A + N_B\mu_B)$ and we derive λ from this formula. For each Fog node, maximum the queue length is set to $Q = 10$. Additional parameters related to adaptive queue size or the possibility to drop only some classes of jobs are not considered in the experiments. Indeed, taking into account all these options would result in combinatorial explosion of the parameter space. This would make the analysis hard to perform and present in the space of a single research paper. Every sensor sends jobs to just one Fog node, and the sensor-to-Fog mapping is statically assigned. The delay experienced every time a job is forwarded corresponds to the network latency between each pair of Fog nodes that is $\delta = 0.9$s (that is the network delayδ is comparable with the average service time $1/\mu$). In this scenario the simulation considers a set of 50 Fog nodes, that should be enough to capture the main characteristics of the algorithm performance. Each simulation is repeated 5 times and the results are averaged over the runs.

The second scenario, called *realistic scenario*, is based on a smart city case study based in Modena, a city in northern Italy with roughly 180'000 inhabitants. The Fog infrastructure aims to support a smart city application that provides environmental sensing and vehicular traffic monitoring. Sensors located along the main city streets collect data on air quality (e.g., atmospheric pollutants such as suspended particulate) and vehicular traffic (e.g., number and speed of vehicles). The goal is to provide in real time a detailed model on urban traffic and air pollution. The Fog layer is composed of Fog nodes placed in facilities belonging to the municipality that exchange information with the sensors and among themselves using long-range wireless links (such as IEEE 802.11ah/802.11af [77]). Each sensor communicates with the nearest Fog node, as in [38], and we assume the delay among Fog nodes to be proportional to the distance between them. Concerning the processing capability of the Fog nodes, we developed a prototype software that counts the number of vehicles in a frame taken from a camera connected to the sensor. Based on these experiments, we modeled the processing time using a Gaussian probability distribution with an average $1/\mu = 10$ms (and with a standard deviation of 1ms). The network delay is proportional to the distance between nodes, but, throughout the infrastructure, is

normalised to have an average delay of $\delta = 10$ms, that is comparable with the processing time. As in the previous scenario, we also introduce a population of A nodes that are faster and are characterised by a processing rate that is double compared to the standard B-class nodes ($\mu_A = 2\mu_B$). We select 10% of the nodes as being of A-class, and the selected nodes are the ones receiving the highest amount of jobs from the sensors. The process of producing images from the sensors is modeled using an exponential distribution. The topology is generated from real geographic data using the PAFFI framework [75], with 100 sensors and 20 Fog nodes. The geographic placement of sensors, results in heterogeneous workload distributions among the nodes, ranging from 250 jobs/sec to some Fog nodes that are almost idle. The average load over the whole infrastructure is such that the average utilisation $\overline{\rho} = \overline{\lambda}/\overline{\mu} = 0.9$.

We summarise the main parameters of the realistic scenario in Fig. 2.3.4. Each Fog node is represented by means of two circles: the thin circle represents its incoming load, while the thick circle is the processing power. If the thin circle is outside the thick one, the node is at risk of overload. Otherwise, if the thin circle is inside the thick one, no overload should occur. Moreover, in this figure we represent A and B Fog nodes classes using two different colors (purple and green, respectively).

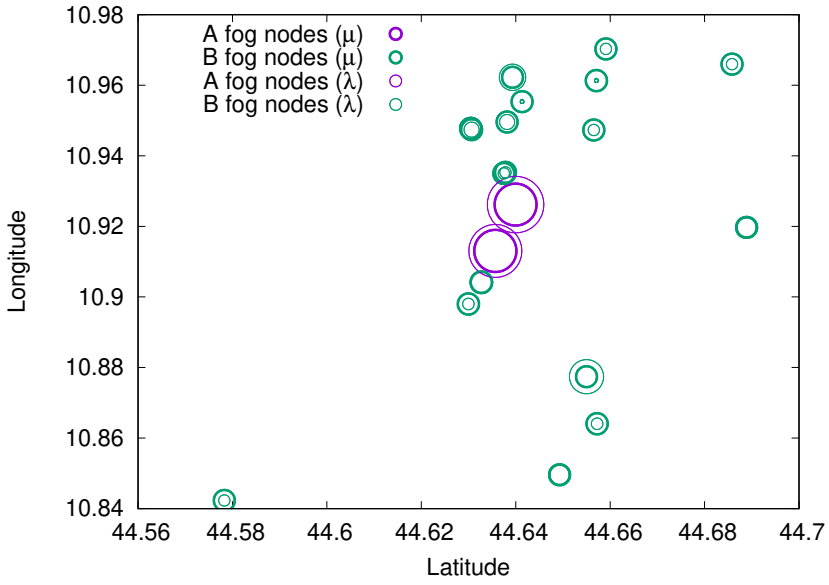


Figure 2.3.4: Realistic scenario - map representation

From a software tools point of view, the simulation is based on the Omnet++ framework[7], with additional modules developed *ad-hoc* to support the two proposed load balancing algorithms.

Throughout the performance evaluation, the main considered performance metrics are:

- *Loss rate*, that is the probability of a job being dropped because the queue of the selected Fog node is full. This condition is described for the model in Sec. 2.3.2.1.
- *Response time*, that is the time occurring between the moment the job is received from the first Fog node, to the moment the processing ends on the final Fog node. The response time model is described in Sec. 2.3.2.1. In the experiments, we provide a breakdown of the response time

[7] https://omnetpp.org/

components: that are *service time* (T_{Srv}, the time spent being processed), *balancer time* (T_{Bal}, the time spent being forwarded among the Fog nodes), and *queuing time* (T_{Queue}, the time spent in the Fog node ready queue waiting to be processed).

Table 2.4: Parameters and metrics for the proposed experimental scenarios.

Scenario parameters	
N_A, N_B	Percentage of type A nodes (B) [%]
λ_A, λ_B	incoming job rate in type A nodes (B) [jobs/s]
μ_A, μ_B	processing rate of type A nodes (B) [jobs/s]
Θ_A, Θ_B	Threshold on type A nodes (B)
M	Maximum number of steps
Q	Maximum queue length
Metrics	
T_{Resp}	Response time. Time elapsed between receiving a job from a sensor and completing its processing. $T_{Resp} = T_{Bal} + T_{Queue} + T_{Srv}$ [s]
T_{Bal}	Time spent in the load balancing phase [s]
T_{Queue}	Time spent waiting in queue [s]
T_{Srv}	Time spent being processed ($= 1/\mu$) [s]
Drop rate	Fraction of job dropped (values in range $[0, 1]$)

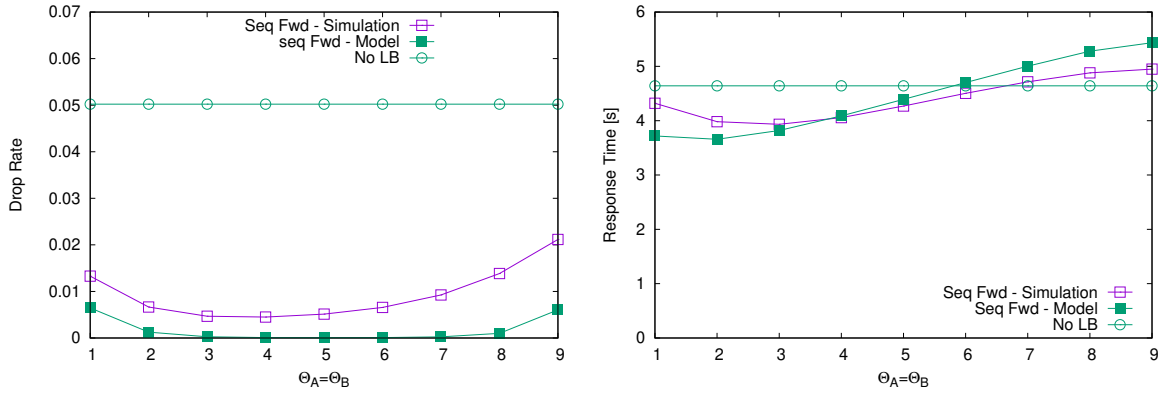
As a reference for the reader we summarise symbols and metrics in Table 2.4, together with their units of measure.

2.3.3.2 Simulation validation

The first step in the proposed analysis is a cross-validation between the results obtained with the simulator and the results of the theoretical model described in Sec. 2.3.2. Specifically, we can compare the performance of the sequential forwarding algorithm in the simplified scenario with restrictive hypotheses that are: $N_A = N_B$ (that is we have the same amount of A-class and B-class nodes such that the probabilities α and β are the same), $\mu_A = \mu_B$, $\Theta_A = \Theta_B$. Furthermore, due to the limitation of the model, we set the maximum number of hops $M = 1$.

Fig. 2.3.5 compares the drop rate and the response times as a function of $\Theta_A = \Theta_B$we show both the sequential forwarding algorithm and the case where no load balancing occurs.

Focusing on Fig. 2.3.5a on the drop rate (that corresponds with the blocking probability in the theoretical model of Sec. 2.3.2.1), we observe that the performance of the No LB case is poor, with a loss rate close to 5%. For the proposed algorithm both the simulation and the model confirm a similar behavior resulting in a U-shaped curve where the values of $\Theta_A = \Theta_B$ very high or very low provide higher drop rates. Indeed, when the threshold is low, most jobs are forwarded to a random neighbour that may be overloaded. In a similar way, when the threshold is high, the jobs are unlikely to be forwarded and are processed locally even is the node is at risk of overload. Finally, comparing the results of the model and the simulator, we observe that both approaches capture the main characteristics of the algorithm. The discrepancy in the numeric value is likely due to the relatively low number of nodes used in the simulation.



(a) Drop rate **(b)** Response Time

Figure 2.3.5: Comparison with model ($M = 1$)

Fig. 2.3.5b shows the response time of a job using both the simulator and the model, compared with the case where no load balancing occurs. Again, we observe that the simulator and the model present a similar behavior, with a range of threshold values ($\Theta_A = \Theta_B < 7$) where the proposed algorithm outperforms the case where no load balancing occurs (please note that the relatively good performance of the non-cooperative approach are due to the high loss rate that reduces the amount of jobs that are served).

2.3.3.3 Evaluation in the simplified scenario

We now provide a more detailed analysis of the proposed algorithms carried out with the simulator.

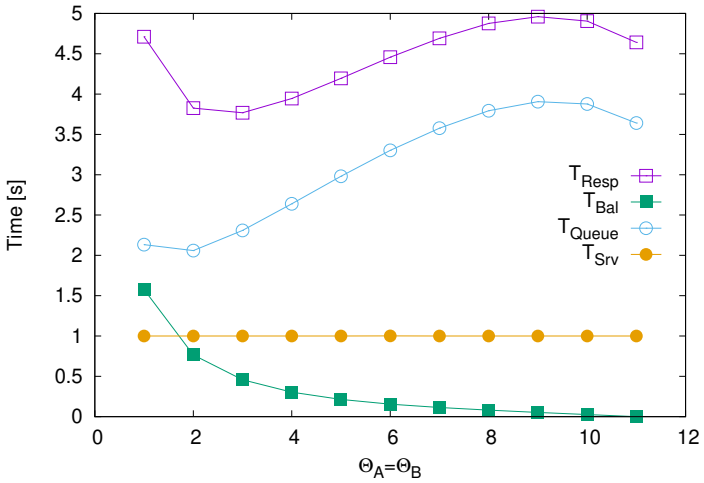
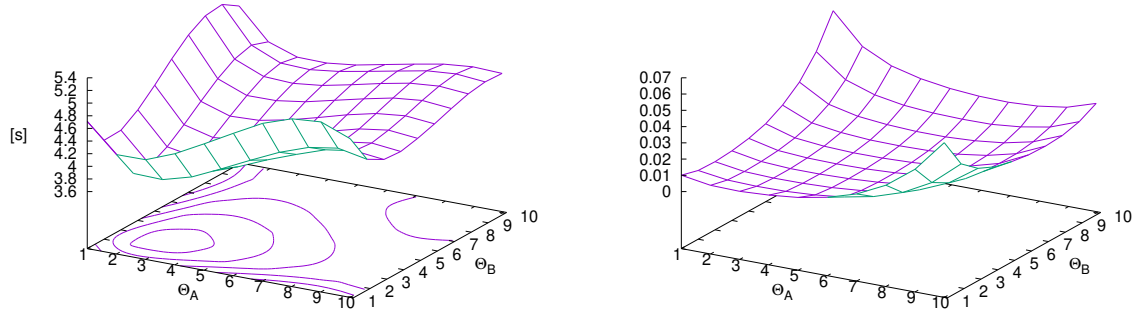


Figure 2.3.6: Breakdown of response time

Fig. 2.3.6 shows the response time for the same conditions considered in Sec. 2.3.3.2, focusing on the sequential forwarding algorithm with $M = 10$ (as we are no longer comparing the results with the model, we no longer need to limit the number of hops)we also provide a breakdown of the response

time (T_{Resp}, purple line with empty squares) in its main component. As expected the service time (T_{Srv}, yellow line with filled circles) corresponds to $1/\mu_A = 1/\mu_B$ and does not depend on the threshold. The balancer time (T_{Bal}, green line with filled squares) decreases as the threshold grows to make the load balancing less aggressive; however, as the load balancer becomes less aggressive, we accept to process jobs on nodes with a longer queue, thus explaining the increase in the queuing time (T_{Queue}, blue line with empty circles). The combination of these contributions determines a local minimum of T_{Resp} for $\Theta_A = \Theta_B = 3$. It is worth to note the similarities among Fig. 2.3.6 and Fig. 2.3.2 of the model, although M is different in the two cases. Comparing the Sequential Forward algorithm with the NoLB alternative the results are clearly in favor of the proposed approach, with a response time reduced by 19% (3.77s *vs.* 4.64s) and a drop rate reduced by a factor of nearly 17 (0.3% *vs.* 5%)

Having discussed the behaviour of the sequential forwarding algorithm in this simple experimental setup, we now introduce the impact of having parameters that differ for the two classes of Fog nodes.



(a) Response Time **(b)** Drop rate

Figure 2.3.7: Response time and Drop rate, $N_A = 50\%$, $N_B = 50\%$, $\mu_A = 1.0$, $\mu_B = 1.0$

In Fig. 2.3.7 we still focus on a case where $N_A = N_B$, $\mu_A = \mu_B$ to understand the impact of the different threshold values Θ_A and Θ_B over the infrastructure. In particular, we consider the drop rate and the response time as the main performance metrics.

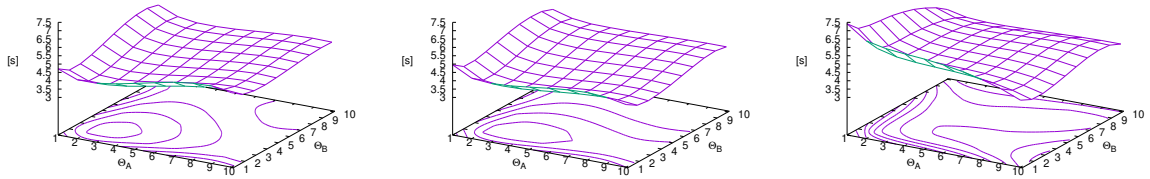
Fig. 2.3.7a shows the response time as a function of Θ_A and Θ_B. If we cut the surface for $\Theta_A = \Theta_B$, we obtain the curve in Fig. 2.3.6. Considering the whole space, we observe that, when a threshold (for example Θ_A) is very low, we experience an increase of response time due to the higher number of hops experienced by the jobs passing trough the node of that class (in the example, A). As the threshold increases (again, let us consider the case where Θ_A grows), we accept to process jobs on A-class nodes with a higher load. This increases the queuing time, and, as a result, leads to higher response times. The figure shows the presence of an optimal configuration for the response time when $\Theta_A = \Theta_B = 3$.

Fig. 2.3.7b shows the drop rate. Again, we observe that, as $\Theta_A = \Theta_B$ we have the U-shaped curve similar to the one in Fig. 2.3.5a. On the other hand, as we explore a parameter space where, for example $\Theta_A \ll \Theta_B$, the low threshold in the A-class nodes determines a higher load in the B-class nodes (a similar behavior is shown for Fig. 2.3.2 in Sec. 2.3.2) resulting in a global increase of the drop rate. The same effect occurs for $\Theta_B \gg \Theta_A$. As for the results in Fig 2.3.7a, we observe a configuration

($\Theta_A = \Theta_B = 6$) that minimises the drop rate. The results are consistent with the findings of Tab. 2.3 obtained using the theoretical model.

2.3.3.4 Sensitivity to μ_A

Having provided some insight on the behavior of the Sequential Forwarding algorithm, we now consider how its performance is affected by a change in the processing power of the A-class nodes. For this analysis, we consider a population with 50% A-class nodes and 50% B-class nodes. Throughout the experiments $\mu_B = 1.0$ jobs/s while $\mu_A \in [1.0, 2.0]$ jobs/s. Again we point out that, unlike the experiments in Sec. 2.3.2, we have $\lambda_A = \lambda_B = \lambda$ for both A-class and B-class Fog nodes and λ grows with the average computing power of the infrastructure so that the average utilisation of the infrastructure is $\rho = 0.9$. This means that, for same configurations we have a potential overload of half of the infrastructure, creating a major challenge for the load balancing algorithms.



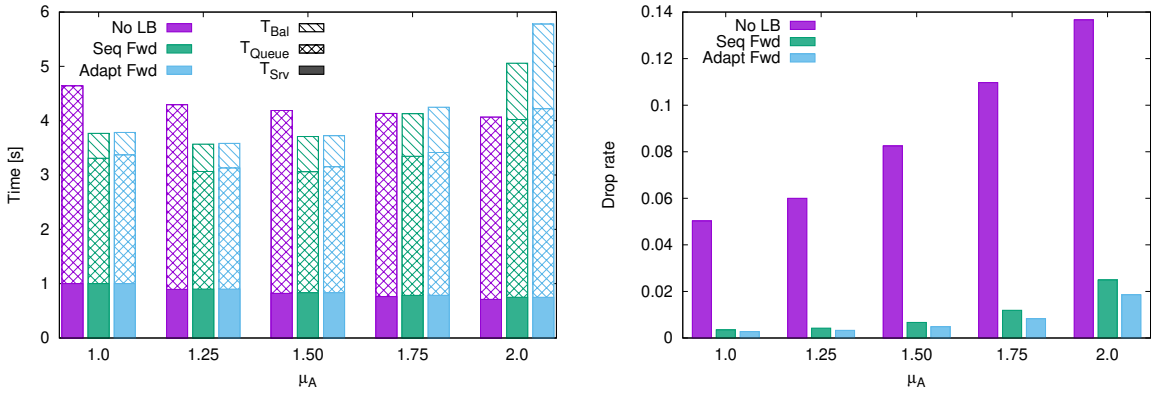
(a) $\mu_A = 1.0$ **(b)** $\mu_A = 1.5$ **(c)** $\mu_A = 2.0$

Figure 2.3.8: Response time for different μ_A

Figure 2.3.8 shows the average response time as a function of Θ_A, Θ_B and describes how such metric changes with μ_A for three values of μ_A. We observe that as the A-class nodes become more powerful, the surface becomes asymmetric stretching towards increasing values of Θ_A. Contour lines in Fig. 2.3.8a and 2.3.8b help observe this effect. However, as μ_A approaches 2.0 the shape of the curve changes significantly. This is caused by the incoming load λ exceeding the μ_B. As a consequence a large fraction of the infrastructure is at risk of overload and the low response times for high threshold values (e.g., $\Theta_A = 10, \Theta_B = 3$) corresponds to trashing conditions of the system with a significant drop rate (in some cases higher than 10%).

We now provide a comparison of the alternatives. In particular, we focus on the *No LB* case, on the Sequential Forwarding and on the Adaptive Forwarding algorithms. For the Sequential Forwarding algorithm, we tune Θ_A and Θ_B (for every value of μ_A) to provide the best response time without causing unacceptably high drop rates. For the adaptive algorithm, that aims at requiring little tuning, we consider the same value of M for every μ_A (the value $M = 6$ was found in preliminary experiments as a good trade-off between a smooth increase in the threshold and the ability to adapt to heterogeneous conditions reducing the number of hops).

For each column in Fig. 2.3.9a we provide a breakdown of the response time divided in service time (T_{Srv}, solid color, bottom part of the histogram), queuing time (T_{Queue}, crossed pattern, middle part of the histogram) and balancing time (T_{Bal} oblique pattern, top of the histogram). In the no LB case,



(a) Response Time **(b)** Drop rate

Figure 2.3.9: Response time and Drop rate for different μ_A

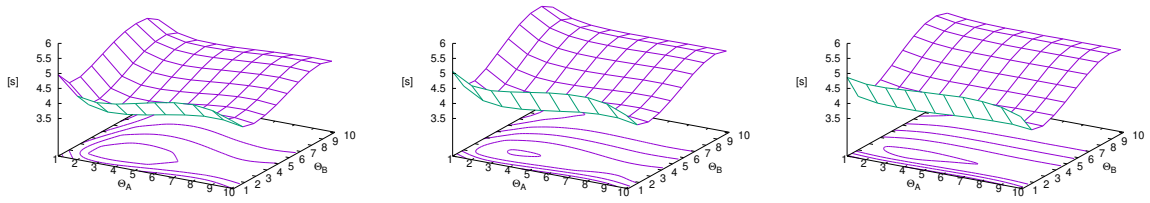
the balancing time is obviously absent. Looking at the response time histograms we observe a general performance degradation for the two proposed algorithms as μ_A grows. This is due to the scenario that increases lambda as μ_A increases. For high values of lambda, half of the infrastructure is at risk of overload with the twofold effect of the need to forward more jobs (resulting in a growth of T_{Bal}) and of increasing the waiting time as the queue tends to be longer in the nodes at risk of overload (explaining the increase in T_{Queue}). The No LB case seems to provide better performance as the load increases, but this is an effect of the high drop rate shown in Fig. 2.3.9b that increases with the load. Indeed, for the NoLB approach the drop rate increases from 5% to 13.7%. For the cooperative algorithms, the drop rate remains below 1% as long as μ_A remains less or equal to 1.75, and in the worst case ($\mu_A = 2.0$) it reaches 2.5 for the Sequential Forward algorithm and remains below 2% for the Adaptive Sequential one. Hence the drop rate for the proposed algorithms remains from 19 to 5.5 times lower compared to the NoLB alternative.

Comparing the Sequential Forwarding and the Adaptive Forwarding algorithms, we observe that the two alternatives provide similar performance, with the adaptive solution offering slightly better performance in terms of drop rate and the sequential forwarding achieving slightly lower response times. However, it is worth to note that the adaptive algorithm provides a major advantage as it can reach a performance level similar to the Sequential Forwarding alternative but does not require a complex tuning of the threshold.

2.3.3.5 Sensitivity to N_A and N_B

The second sensitivity analysis carried out in the experiments concerns the ratio between the A-class and the B-class nodes. In this case, we keep the values of μ_A and μ_B fixed ($\mu_A = 1.5, \mu_B = 1.0$ jobs/s).

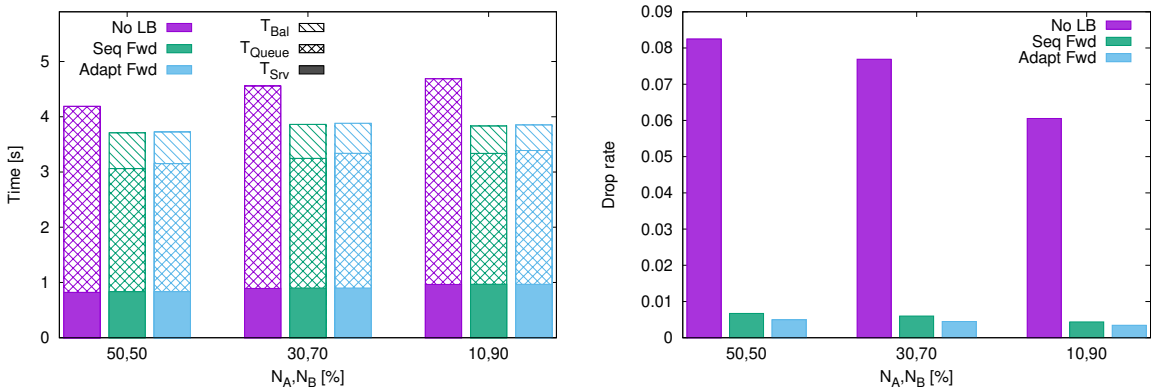
As in the previous sensitivity analysis, Fig. 2.3.10 shows the response time for different percentages of A-class and B-class nodes (represented with the N_A and N_B parameters). We observe that the overall shape of the surface remains similar. However, as we reduce the number of A-class nodes, we observe that the contour lines of the figures become more and more similar to parallel lines in the direction of a single value of Θ_B. This means that the weight of the Θ_A parameter becomes less



(a) $N_A = 50\%, N_B = 50\%$ **(b)** $N_A = 30\%, N_B = 70\%$ **(c)** $N_A = 10\%, N_B = 90\%$

Figure 2.3.10: Response time for different N_A, N_B

and less significant compared to Θ_B. On one hand, this result is expected because, as we reduce the number of fast A-class nodes, the slower and more numerous B class nodes become the real bottleneck of the infrastructure. On the other hand, this result provides an important lesson to learn: adding a few powerful nodes in a slow infrastructure is unlikely to solve any performance problem unless an adequate tuning of the large fraction of the remaining nodes is carried out.



(a) Response Time **(b)** Drop rate

Figure 2.3.11: Response time and Drop rate for different N_A, N_B

Figure 2.3.11 summarises the main findings of the sensitivity analysis with respect to the N_A, N_B parameters. As in the previous analysis, we present a breakdown of the response time with service time (T_{Srv}), queuing time (T_{Queue}) and balancing time (T_{Bal}) in Fig. 2.3.11a. Again, for the Sequential Forwarding algorithm, the results are referred to the best scenario for every (Θ_A, Θ_B) couple considered, while for the Adaptive Forwarding algorithm we relay on the initial tuning. Fig. 2.3.11b shows the drop rate for the various considered alternatives. From a comparison of the three considered options, the No LB is clearly the worse solution, with higher drop rate and higher response time (in this case the infrastructure does not reach a level of trashing such that the drop rate reduces significantly the response time). Indeed, the response time remains from 11% to 19% higher compared to the proposed algorithms while the drop rate of the NoLB approach remains roughly 15 times higher compared to the proposed approach.

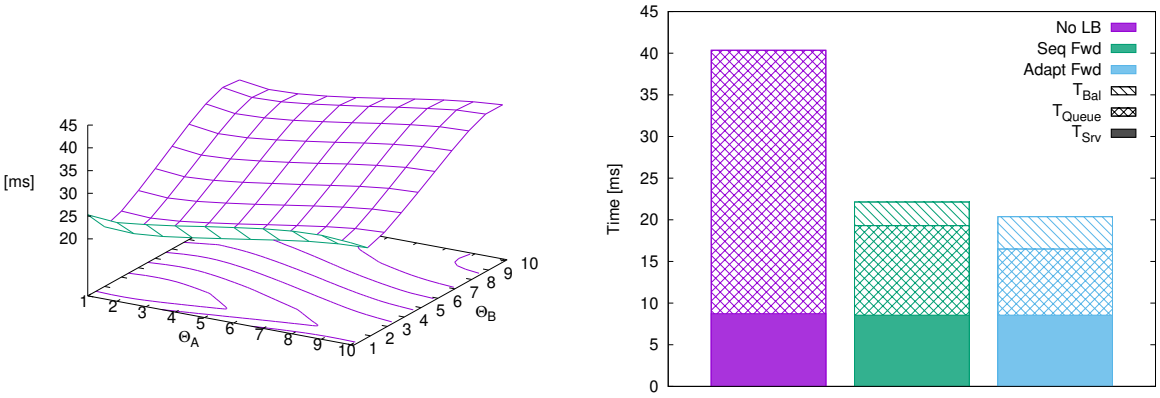
On the other hand, the two other load balancing algorithms provide similar performance in terms

of drop rate and response time, confirming the main finding of Sec. 2.3.3.4.

2.3.3.6 Evaluation in the realistic scenario

We now focus on the realistic scenario, that is a case where loads are unevenly distributed over the Fog nodes, the network link delays are uneven and where the processing time is no longer described as an exponential (that is the Fog nodes are described as M/G/1/Q queuing network elements). We recall that in this scenario, the setup is based on a geographic placement of nodes based on real locations.

Fig. 2.3.12 provides a performance evaluation for the different considered algorithms in terms of response time. Specifically, Fig. 2.3.12a shows the response time for the Sequential Forwarding algorithm as a function of the threshold Θ_A and Θ_B. We confirm the main findings of Sec. 2.3.3.5 about the major impact of the threshold Θ_B that affects the performance of the more numerous slower nodes. However, the high incoming load in the A-class nodes (we recall that in this scenario $\lambda_A \gg \lambda_B$ due to the criteria used to select the A-class nodes) makes the impact of the parameter Θ_A less negligible compared to the analysis in 2.3.3.5. Using the results in Fig. 2.3.12a to tune the Sequential Forwarding algorithm, in Fig. 2.3.12b we can compare the response times of the three considered alternatives, that is the No LB case, and the Sequential Forwarding and Adaptive Forwarding algorithms. As in the previous analyses, for the adaptive algorithm, we consider $M = 6$ that is the value identified previously as a value providing good and stable performance. It is worth noting that we do not provide a figure concerning the drop rate because the two algorithms that provide load balancing have a drop rate very low (less than 0.2%), while the No LB alternative is characterised by a drop rate in the order of 13%.



(a) Response time of sequential forwarding algorithm

(b) Response time comparison

Figure 2.3.12: Performance evaluation in the realistic scenario

The main outcome of this comparison confirms the main finding of the previous experiments: the two load balancing algorithms provide similar performance both in terms of response time and drop rate (response time between 19 and 21 ms, with a drop rate in both cases below 0.2%) and far outperform the NoLB approach that is characterised by a response time nearly double compared to the proposed algorithms. However, we point out once again that the good performance of the sequential forwarding algorithm is the result of careful tuning of the algorithm parameters, while the adaptive alternative provides good and stable performance with minimal effort.

2.4 Study on the impact of the stale information on the scheduling decision

Edge computing resource management is an essential open issue in the research agenda [74], [78], which boils down to determining a mapping A between units of computations C, submitted to the Edge layer by users, and the available Edge nodes E:

$$A : C \rightarrow E$$

in a way that some performance attributes are optimised. Depending on the strategy followed to determine A, the solutions can be divided into two categories: offline and online. Let T_A be the time required to calculate $A(\cdot)$, and T_C the service time to compute C.

In offline management strategies, several units C (in this context also called *jobs*) are collected and grouped in batches. Moreover, A is a solution to a well-defined optimisation problem, where optimisation actions include job execution migration or offloading from an Edge node to another one or from the Edge layer to the cloud one. The prerequisite for these algorithms is that the response time is not critical. A variation to the scheme is to divide the time into intervals, observe the performance during a time interval, and migrate the expected jobs of the next time interval towards other nodes, based on an estimation of the benefit of such migration. Here the prerequisite is that the load is stationary over a suitable period of time.

On the other hand, online resource management is used when the response time is critical and hence waiting to form a batch is not possible or the load is not stationary. Online approaches apply A on-the-fly to each new submitted job C. In this case, T_A includes the time to gather information from other nodes, eventually needed to make a forwarding decision, e.g. via probing, and the job transfer time[8]. The ratio among T_A and T_C may limit the applicability of online approaches. We will refer to this ratio as the schedule lag and denote it as $\eta = \frac{T_A}{T_C}$. The schedule lag measures the time interval between decision making and decision actuating times.

The value η depends on the application characteristics and on the structure of the Edge computing system. For example, let us consider an image processing application where the job C corresponds to detecting and recognising objects in video frames. An application with a frame rate of 60 FPS implies that $T_C < 16.67\ ms$. Assuming a high-speed connection among Edge nodes, for example of 1 Gbps, and a frame size of 2 MB, the frame transfer time is about $20\ ms$, so that $\eta > 1$. Image compression techniques can reduce this time, but still, it is likely to have $\eta \approx 1$. More complex image processing may require higher service times leading to $\eta < 1$. Another example is sensing applications that process thin data, like those collected by sensors. Here data are likely to be a small JSON file of a few Kbytes and requires very short service times, likely providing $\eta \geq 1$.

This section considers a family of distributed online protocols based on randomisation algorithms in which the function A is computed by each node when it receives a new job C. The function A returns either a node id of a less loaded node picked at random among a small subset or none, which means

[8] For the sake of simplicity we count it as a component of T_A, even if strictly speaking the transfer time is needed to reach the Edge node and not to determine it.

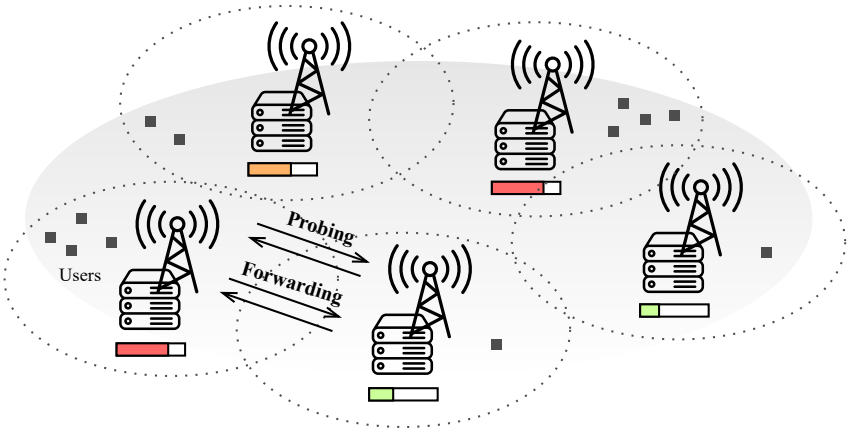


Figure 2.4.1: The cooperation in the Edge computing system based on load-aware random probing: the overloaded Edge node (bottom left of the figure) probes a random neighbour and then forwards a job to it.

executing the job locally. The implementation of A is done via random probing [18], [28], as shown in Figure 2.4.1.

The main contribution of this section is a study of the impact of schedule lag η on the performance of such load *load-aware* balancing protocols based on randomisation. The work studies how and to which extent making a decision based on stale information concerning the load state of the nodes, weakens the effectiveness the algorithm and how load balancing can be achieved when this delay in communicating state information is unavoidable. The work shows that it exists a "critical" value of η starting from which load information has no value and a simpler blind forwarding algorithm performs better.

Figure 2.4.2 sketches this claim, in which the critical value is the dot line where $\eta = 1$. When $\eta > 1$ (upper triangle) there is a high risk for *load-aware* algorithms of schedule decisions based on stale (out-of-date) load information and consequent poor performance. On the other hand, in scenarios characterised by values of $\eta < 1$, the risk related to stale load information is low, as represented by the triangle at the bottom and a load-aware algorithm works at its best.

The work presents a thorough analysis of the impact of stale information on the effectiveness of load balancing protocols, with the final aim to support the system designers in deciding, based on the scenario, which algorithm is most suitable. In the proposed analysis, we consider typical application scenarios based on smart cities. We assume to have intelligent traffic lights that can monitor traffic (cars, pedestrians and bicycles) possibly equipped with cameras and sensors. The traffic information can be used for multiple purposes, from supporting autonomous driving to identifying suspicious behaviour from people.

A qualifying point of the proposed analysis concerns the methodology used in the present study. We combine a theoretical model and numerical solutions with a simulation approach. The theoretical model derives the correlation between two states as a function of η and quantifies when load-aware probe-based algorithms, which pull load information from the other node, become, in fact, useless. This result is confirmed by detailed simulations that find a similar conclusion. Moreover, a Sequential Forwarding algorithm [15], that follows a *load-blind* approach where the decision is based only on local

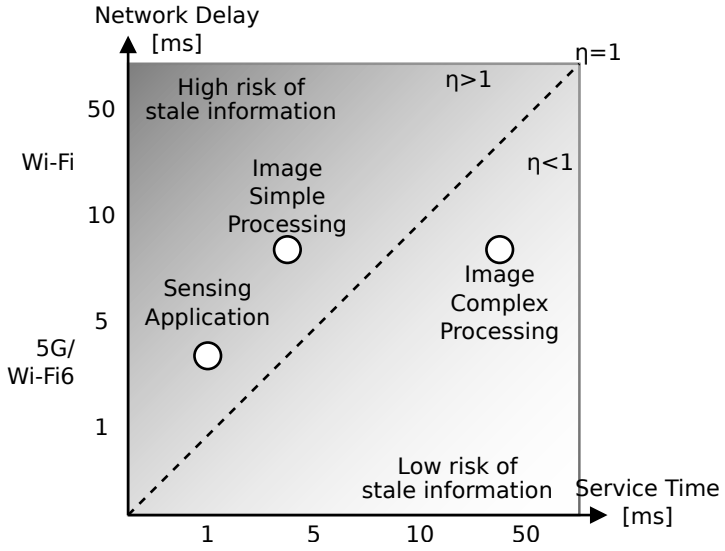


Figure 2.4.2: The figure qualitatively shows the risk of stale load information depending on η.

load information, is also analysed.

Using simulations, we explore a wide set of parameters considering multiple scenarios: a *uniform mesh scenario* where incoming load is evenly distributed among nodes organised in a regular mesh, and a *geographic scenario* derived from the topology of a medium-sized city in Italy.

2.4.1 Algorithm definition

We now introduce the two algorithms used in our study to investigate the effect of stale information on different approaches to load balancing. Specifically, we first introduce the *Probe-based* algorithm [16], representative of *load-aware* approaches, then we describe a load-blind algorithm, namely *Sequential forwarding*, presented in literature in [15].

2.4.1.1 Probe-based algorithm

The *probe-based* algorithm relies on a threshold to determine whether, upon receiving a new job, a probe for a less loaded neighbour is to be started. The threshold Θ is applied to the system load, that represents the number of jobs queued in the Edge node (or being executed). This metric is used as an estimation of the waiting time for the incoming job. If the load exceeds the threshold, a probe is started, with the the Edge node issuing query messages to a randomly selected neighbour.

Algorithm 4 presents the formalisation of the probe-based algorithm. When a job from a sensor is received, the Edge node uses the threshold Θ and the local load to decide if a probe for the neighbour load should be issued (jobs forwarded from other Edge nodes are processed locally without additional evaluation). If probing is required, the Edge node issues a query message to the neighbour and waits for the response. The neighbour provides its load status within the response, so the Edge node can decide if the job has to be forwarded to the neighbour or if the job is to be processed locally (if the neighbour has a higher load than the local node). It is worth noting that, in the case of high network delay, the

Algorithm 4 Probe-based Algorithm

```
Require: Θ, Job
  if Job.IsForwarded() or System.Load() ≤ Θ then
    ProcessLocally(Job)
  else
    Neigh ← Random(System.neighbours())
    NeighLoad ← Probeneighbour(Neigh)
    if System.Load() > NeighLoad then
      Forward(Job, Neigh)
    else
      ProcessLocally(Job)
    end if
  end if
```

load returned by a neighbour may be a *stale* information far different from the load the forwarded job will actually encounter. This may result in inaccurate forwarding decisions, already pointed out in the area of Web servers [40].

Algorithm 5 Local processing: *ProcessLocally*()

```
Require: Job
  if System.Queue() < System.MaxQueue() then
    Enqueue(Job)
  else
    Drop(Job)
  end if
```

Algorithm 5 details the case where a job is processed locally (for example, due to the call to the *ProcessLocally*() procedure in Algorithm 4). In this case, the job should be placed in the ready queue of the server. However, if the queue is already full (since it has a finite size), the job is dropped, resulting in a loss.

2.4.1.2 Sequential Forwarding algorithm

The *Sequential Forwarding* algorithm [15] uses the threshold Θ to decide if an incoming job must be forwarded to a random neighbour or locally processed. The algorithm relies on an additional parameter M, the maximum number of steps to guarantee a limit on the delay associated with the load balancing phase. Algorithm 6 presents the formalisation of the algorithm. When a job arrives, if the job has not yet reached the M-th step, the system load is considered: if the value does not exceed the threshold Θ, the job is accepted and scheduled for local processing; otherwise, it is forwarded to a randomly-selected neighbour. If the job has already been forwarded M times, it is scheduled for local processing. This algorithm, which is load-blind, is extremely simple to implement. In this study, we set the parameter $M = 5$, which is a value proved in preliminary experiments to provide low response time and low drop rate. A detailed description of this parameter and its impact on the algorithm performance has been provided in [14].

Algorithm 6 Sequential Forwarding Algorithm

```
Require: M, Θ, Job
  if Job.Steps() ≥ M then
    ProcessLocally(Job)
  else
    if System.Load() ≤ Θ then
      ProcessLocally(Job)
    else
      Neigh ← Random(System.neighbours())
      Job.IncrementSteps()
      Forward(Job, Neigh)
    end if
  end if
```

The sequential forwarding process is detailed in Algorithm 6. We assume that the data structure describing the job is enriched with metadata to keep track of the number of times the job is forwarded.

2.4.2 System models

In this section,we develop two models for the probe-based protocol that uses stale (out-of-date) information and for the sequential forwarding protocol. Table 2.5 summarises the main term definitions.

2.4.2.1 A model for probe-based protocols with stale information

The probe-based algorithm relies on load state information gathered from the other nodes. We define the *schedule lag* the time interval elapsed from when a node reports its state until the node receives a job due to a scheduling decision based on that value. As the lag increases the scheduling decisions becomes sub-optimal since they are based on stale information. Intuitively this occurs because the job finds the probed node in a state which is progressively unrelated to the reported state.

We now compute this value considering the generic interaction pattern of the probe based protocol, see Figure 2.4.3. Let t_P be the time when the node A receives a job, T_P the time required to decide where to schedule a job and T_F the time required to forward. In addition, τ_P and τ_J denote respectively the transmission time of a probe or a job. The schedule lag value τ is the sum of two contributions due to queuing and transmission times.

In the proposed model, if the current state of the node is $k > \Theta$ the node stores the job into a Probe Queue (PQ) with k annotated and it selects another node B, picked at random. It waits from t_P to t_1 in the queue before the probe message for that job is actually sent over the wire to B. At time t_{PR}, B receives the probe message, samples its load i that sends back to A through its Probe Reply Queue (PRQ). The total delay for this operation is T_{PR}, which is the first source of delay of the lag. At time t_F, node A receives the reply message and decides either to serve the job locally (if $k < i$) or to forward it to B through its Forwarding Queue (FQ). Job forwarding is the second source of the schedule lag because the job enters the service queue at time t_s when it is fully received. The value of schedule lag is then $\tau = T_P + T_F$. Note that the service time of PQ lasts from t_P to t_F and it is equal to $T_{PR} + \tau_P$.

Symbol	Explanation
λ	Job arrival rate.
μ	Service completion rate.
ρ	Traffic intensity ($\frac{\lambda}{\mu}$).
p_p	*Probing probability*: probability to probe a node.
τ_P	*Probe Sending time* Time required to send a probe message over the wire.
T_P	*Probe time*: total time required to decide where to serve a job, for sequential forwarding $T_P = 0$.
T_{PR}	*Probe Reply Time*: Time required to receive the state information from a node.
σ^2_{PR}	Variance of Probe reply time.
p_F	*Forwarding probability*: probability that a node forwards a job.
τ_J	*Job Sending time* time required to send a job over the wire.
T_F	*Forward Time*: total time required to forward a job.
T_{Bal}	*Balancer Time*: time required to move a job in a service queue either the local queue or a remote queue ($T_{Bal} = T_P + T_F$).
τ	*Schedule lag*: time difference between the time when a node reports its state until the node receives a job whose scheduling is based on that value.
η	Ratio between schedule lag and service time.
Θ	Activation threshold.
π_i	Probability that the state of the service queue is i.
$\tilde{\pi}_i$	Probability that the state of the service queue is at least i.
$\tilde{\pi}_i{}'$	Probability that the state of the service queue is at least $max\{\Theta, i\}$.
Performance metrics	
P_B	*Drop rate*: probability to drop a job
T_{Resp}	*Response Time*: Average time elapsing from when a job is received from and Edge node until its service ends

Table 2.5: Summary of symbol definitions used in the model.

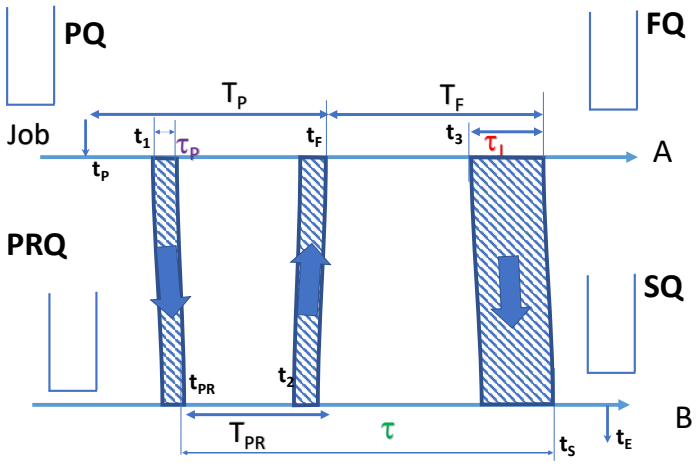


Figure 2.4.3: Time diagram model of a probe based algorithm.

This analysis doesn't consider the propagation delay of signals because nodes are physically closed enough to neglect this contribution.

The schedule lag value is critical because it determines the probability that by the time that a job is received the state of B changes from i to j, which are exactly the events that cause sub-optimally. The goal of the proposed model is to express such probability of as a function of τ and from here how its value affects the performance of the algorithm.

Model description Based on the previous description, we now define a model for a probe based algorithm running on a large number of nodes. The model is based on three queues belonging to the node A and a queue belonging to node probed node B:

1. Probe Queue (PQ) has an unbounded capacity and stores jobs awaiting for a probe reply message; the average service time of the queue is $T_{PR} + \tau_P$ and its variance is σ^2_{PR} (see below). The propagation delay is not considered. This queue is modelled as an $M/G/1$ queue.
2. Probe-Reply Queue (PRQ) has as unbounded capacity and constant service time τ_P equal to the transmission time of the probe reply message, which is modeled as an $M/D/1$ queue.
3. Forward Queue (FQ) has an unbound capacity for jobs to send to other nodes, whose service time is τ_J equals to the job transmission time. This queue is modelled as an $M/D/1$ queue.
4. Service queue (SQ) with finite capacity K for jobs being served by the node. The service time of this queue is exponentially distributed with mean $T_S = \frac{1}{\mu}$). This time corresponds to the actual time required to process the job. The dynamic of the queue is modelled as a birth-death process.

Nodes receive jobs according to a Poisson flow with rate λ. We assume that these queues are independent from each other, [18] and denote by π_i the steady-state probability of the service queue length being i.

Probe Queue (PQ). The probe queue is modelled as an M/G/1 queue. A job enters the queue only when the length of the service queue (SQ) is higher than Θ, and it leaves the queue when the probe reply is received. The probability that the node probes another node is clearly:

$$p_p = \sum_{i>\Theta} \pi_i \tag{2.16}$$

The *mean* service time of the queue is $T_{PR} + \tau_P$, so that the traffic intensity for this queue is:

$$\rho_P = \lambda p_p (\tau_P + T_{PR})$$

The variance of this service time is σ^2_{PR} (see later). The mean time spent by message in this queue is computed from the Pollaczek-Khinchin mean formula, [79]:

$$T_P = \frac{1 + C_s^2}{2} \frac{\rho_P}{1 - \rho_P} (\tau_P + T_{PR}) \tag{2.17}$$

where $C_s^2 = \frac{\sigma^2_{PR}}{T_P^2}$ is the squared coefficient of the service time.
Probe-Reply Queue (PRQ). This queue is modelled as an M/D/1 queue. Because the transmission time of a probe message is τ_P, the traffic intensity towards this queue is:

$$\rho_{PR} = \lambda p_p \tau_P$$

The mean time spent by message in this queue is derived from the well known M/D/1 formula:

$$T_{PR} = \frac{\tau_P}{2} \frac{2 - \rho_{PR}}{(1 - \rho_{PR})} \tag{2.18}$$

The variance of the waiting time for this queue is: $\sigma^2_{PR} = \frac{\rho_{PR}\tau_p^2}{3(1-\rho_{PR})} + \frac{\rho_{PR}^2\tau_p^2}{4(1-\rho_{PR})^2}$, [79].
Forward Queue (FQ). The forward queue is also modeled as an M/D/1 queue. A job enters the queue only when the service queue length is higher Θ *and* the state reported by the probed node is lower than the current state, so that the probability that a job enters the queue is:

$$p_F = \sum_{i>\Theta} \sum_{j<i} \pi_i \pi_j \tag{2.19}$$

Similarly to the previous queue, the delay due to job forwarding is:

$$T_F = \frac{\tau_J}{2} \frac{2 - \rho_J}{(1 - \rho_J)} \tag{2.20}$$

where $\rho_J = \lambda p_F \tau_J$.
Service Queue. To study this queue we take a different approach considering $N \to \infty$ nodes, and using a deterministic fluid flow model. This approach has been successfully applied to other studies on load balancing, [18].

Let $P_{ij}(0, t)$ be the fraction of service queues in the system that at time $t = 0$ have length i and at

time t have length j, $q_{ij}(t)$ the rate at time t at which the length of a queue changes from i to j , and $q_{jj}(t)$ the rate at time t at which it changes from j. The dynamic of these nodes is described through the set of equations, [41]:

$$\frac{dP_{ij}(0,t)}{dt} = -P_{ij}(0,t)q_{jj}(t) + \sum_{k \neq j} P_{ik}(0,t)q_{kj}(t)$$

The equations measure the rate at which the population of nodes change their state. We now specialise the above equations for jobs arriving to nodes according to independent Poisson processes with rate λ, exponentially distributed amount of service time with mean $\frac{1}{\mu}$ and finite queue capacity K. The only rates that are not zero are: $q_{jj} = \lambda_j + \mu$, $q_{jj+1} = \lambda_j$, $q_{jj-1} = \mu$. Hence:

$$\frac{dP_{ij}(0,t)}{dt} = -P_{ij}(0,t)[\lambda_j + \mu] + P_{ij+1}(0,t)\mu + P_{ij-1}(0,t)\lambda_{j-1} \quad 1 \leq j < K$$

$$\frac{dP_{i0}(0,t)}{dt} = -P_{i0}(0,t)\lambda_0 + P_{i1}(0,t)\mu$$

$$\frac{dP_{iK}(0,t)}{dt} = -P_{iK}(0,t)\mu + P_{iK-1}(0,t)\lambda_{K-1}$$

This set of equations can be given in the following matrix form due to Chapman-Kolmogorov, e.g. see [80]:

$$\frac{d\mathbf{P}(0,t)}{dt} = \mathbf{Q}\mathbf{P}(0,t)$$

where $\mathbf{Q}$ is the following $(K+1) \times (K+1)$ (infinitesimal generator) matrix:

$$\mathbf{Q} = \begin{bmatrix} -\lambda_0 & \lambda_0 & 0 & \dots & 0 \\ \mu & -\mu - \lambda_1 & \lambda_1 & \dots & 0 \\ 0 & \mu & -\mu - \lambda_2 & \dots & 0 \\ & & \ddots & & \\ 0 & 0 & 0 & \dots & -\mu \end{bmatrix}$$

Formally, the solution of this equation with initial condition $\mathbf{P}(0,0) = \mathbf{I}$ is:

$$\mathbf{P}(0,t) = e^{\mathbf{Q}t}$$

The challenging part to apply this equation is that the arrival rates λ_i that appear in the matrix depend on the time t. As a workaround we use a constant value for the rates to obtain an approximation of the real values of the matrix. Let focus of the node B probed at random by A (see Figure 2.4.3) and suppose that the time zero corresponds to the time when B samples its state, say i, i.e.. $t_{PR} = 0$. For the sake of simplicity we omit the symbol zero from the element of the matrix $\mathbf{P}$. The probability that after a time lag τ the state of B changes from i to j is $P_{ij}(\tau)$ because this is also the fraction of nodes that change their state from i to j and $P_{ij}(\tau)$ can be interpreted at the probability that B belongs to

this fraction. The rate at which jobs find node B in state j, given that it announced i, is then:

$$\lambda_j^F(\tau) = \frac{1}{\pi_j}\lambda \sum_{i=0}^{K} \pi_i \tilde{\pi}'_{i+1} P_{ij}(\tau) \tag{2.21}$$

Equation (2.21) reflects the probabilities of the following events: (i) node B sends a reply message to the probe message reporting state i (which occurs with probability π_i), (ii) the state of A was $k \geq i+1$ (occurring with probability $\tilde{\pi}'_{i+1}$) and (iii) during τ time units the state of B changed from i to j. These probabilities are conditioned to the event of B being in state j. The rates in the matrix $\mathbf{Q}$ are then:

$$\lambda_j(\tau) = \begin{cases} \lambda_j^F(\tau) + \lambda & \text{if } j \leq \Theta \\ \lambda_j^F(\tau) + \lambda\tilde{\pi}_j & \text{otherwise} \end{cases} \tag{2.22}$$

Indeed, node B receives jobs from nodes like A (first term), plus all the jobs coming from its users (when the state is $j \leq \Theta$), or from users if the state of the probed node was worst than j ($j > \Theta$). Without loss of generality, from now on, we assume death rate $\mu = 1$.

Model solution. To find the steady-state of the above set of queues we use a fixed point algorithm divided into two steps. The first step calculates the $\mathbf{Q}$ of the service queue for a given fixed τ, while the second step calculates the waiting times of all the three queues based on the steady-state of the service queue. Initially, $\tau = \tau_p + \tau_J$ that represents the minimum delay needed to schedule a job from node A to node B.
STEP 1. Given a value τ, first the matrix $\mathbf{Q}_0$ where $\lambda_i = \lambda$ is created. Then, using this matrix, the vector π of the steady-state probabilities is computed. From these values, the rates of Equation (2.22) are computed, and they are used to define another matrix $\mathbf{Q}_1$. This procedure is repeated until the numerical convergence to a matrix $\mathbf{Q}^*$.
STEP 2. This step uses $\mathbf{Q}^*$ to compute the steady-state probabilities of the service queue and from here, the waiting times associated with the probe, probe reply and forward queues from Equation (2.17) Equation (2.20) and Equation (2.18), which allows determining a new value, say τ'. A new STEP 1 is then executed to find a new matrix $\mathbf{Q}^*(\tau')$. This second step also monitors the distance between two successive matrix passed from STEP 1 and halts the computation if the value is lower than ϵ.

Model convergence In the numerical examples, the above procedure converged in less than 500 steps for $\epsilon = 10^{-5}$. Figure 2.4.4a compares the result from the model and discrete event simulations.

2.4.2.2 A model for the sequential forwarding protocols

The model for sequential forwarding is easier since a job is forwarded blinding to another node and the schedule lag has τ has no effects. Recall that jobs are served by a node only if the current service queue length is at most Θ or the job was already forwarded M times. The rate of jobs that are forwarded only one time is $\lambda\tilde{\pi}_{\Theta+1}$, those that are forwarded two times is $\lambda\tilde{\pi}^2_{\Theta+1}$, etc. The total rate of jobs arriving to

a node from other nodes is then:

$$\lambda_{SF} = \lambda \sum_{m=1}^{M} \tilde{\pi}_{\Theta+1}^{m} = \lambda\Big(\frac{1-\tilde{\pi}_{\Theta+1}^{M+1}}{1-\tilde{\pi}_{\Theta+1}} - 1\Big)$$

Since a job can find the landing queue at any state, the arrival rate of a generic queue is:

$$\lambda_j = \begin{cases} \lambda + \lambda_{SF} & \text{if } j \leq \Theta \\ \lambda_{SF} & \text{otherwise} \end{cases}$$

As before, to model job forwarding, we assume that a job is moved to a forwarding queue, modelled as an M/D/1 queue with $\rho_{SF} = \lambda_{SF}\tau_J$ and service time τ_J. Accordingly, the average waiting time is:

$$T_{SF} = \frac{\rho_{SF}}{2(1-\rho_{SF})} \tag{2.23}$$

2.4.2.3 Performance Metrics

Dropping rate The drop rate is defined as the probability to drop a job. For the probe-based algorithm, it is given by:

$$P_B = \pi_K^2 + \sum_{i=\Theta}^{K-1} \tilde{\pi}_{i+1}\pi_i P_{iK}(\tau) \tag{2.24}$$

because a job is dropped if: (i) the receiving node is full but the job cannot be forwarded since the receiving node reports it is full as well (first term), or (ii) the job is forwarded, but during the time τ the target node becomes full (the job finds the node in state K) and drops the job.

For sequential forwarding:

$$P_B = \tilde{\pi}_{\Theta+1}^{M}\pi_K \tag{2.25}$$

which reflects the fact that a job is forwarded towards a congested node, i.e. whose state is K.

Response time The *Response time* T_{Resp} of a job is defined as the time elapsed from when a job is received from a node (time t_p of Figure 2.4.3) until its service ends, t_E. This delay can be conveniently expressed as the sum of two contributions: the balancer time T_{Bal} due to move a job into a queue and the proper service time:

$$T_{Resp} = T_{Bal} + T_S$$

From Figure 2.4.3 the balancer time is the weighted sum $T_{Bal} = p_P T_P + p_F T_F$, see Equation (2.16), Equation (2.19), Equation (2.20) and Equation (2.18). The second term is determined by applying the Little's result to the service queue:

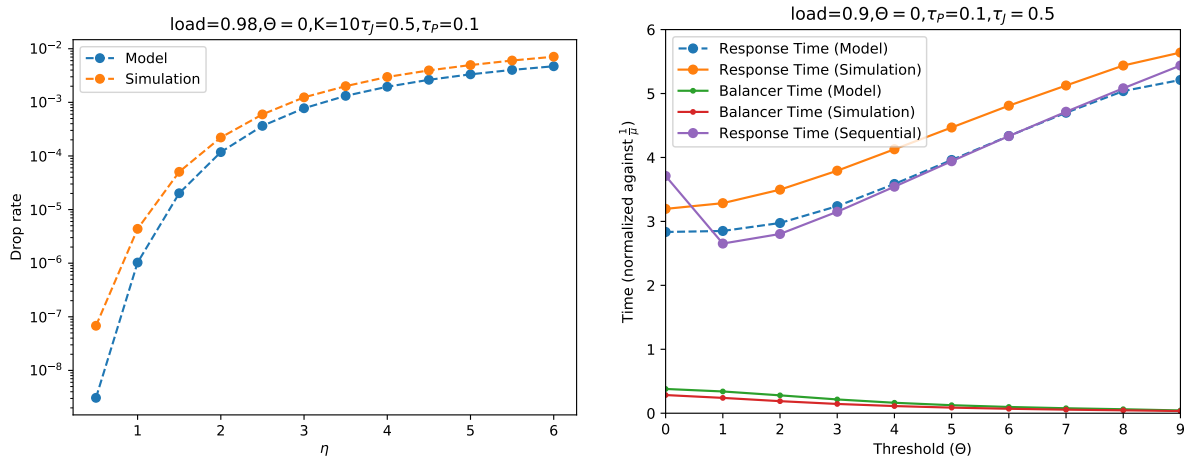
$$T_S = \frac{\sum_i i\pi_i}{\lambda(1-p_B)}$$

For sequential forwarding:

$$T_{Resp} = T_S + \sum_{m=0}^{M-1} T_{SF}(1-\tilde{\pi}_{\Theta+1})\tilde{\pi}_{\Theta+1}^{m} = T_S + T_{SF}(1-\tilde{\pi}_{\Theta+1}{}^{M})$$

where T_S is derived from the Little's result using Equation (2.25), while T_{SF} is computed in Equation (2.23).

Figure 2.4.4b shows the response time for the probe-based algorithm and sequential forwarding algorithms that reveal a good match between model prediction and simulations. A deep explanation of this shape is given in the simulation section.



(a) Probe based model validation

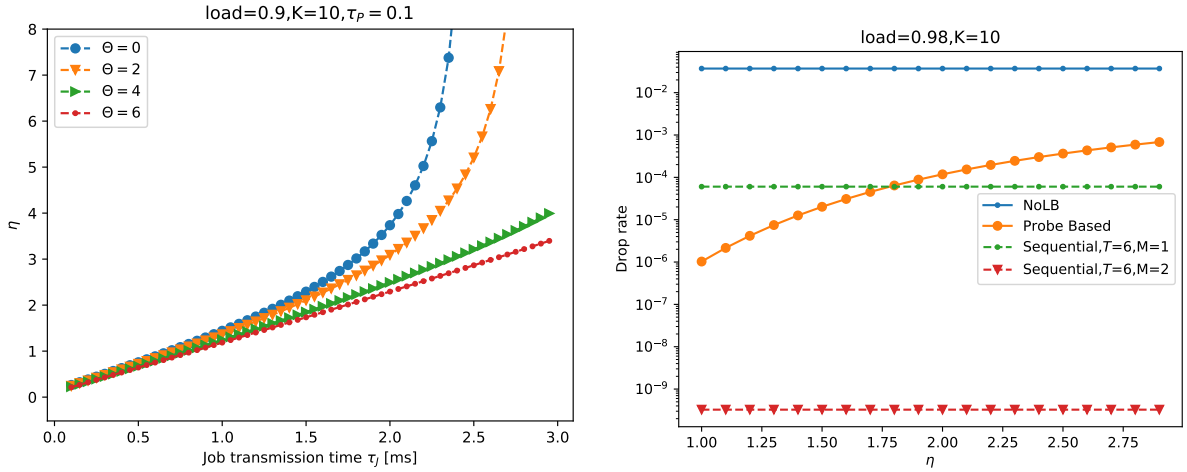
(b) Response and Balancer time vs threshold Θ

Figure 2.4.4

2.4.2.4 Performance detriment and critical value of the schedule lag

With the model in hand we can now address the main question: how long the schedule lag can be before the detriment of the algorithm becomes prominent? Before that, it is convenient to use a new quantity η which is the schedule lag normalised respect to the service time, $\eta = \tau\mu$ and that permits to set $\mu = 1$ in the numerical solutions. It it is also important to realised that the main source for a high η is the need to transfer long jobs. Figure 2.4.5a shows in fact how η increases with the size of the job that is forwarded for the probe-based protocol. A similar shape is found for the sequential forwarding algorithm. Clearly, because a lower threshold implies a less number of jobs that are forwarded by the Forward Queue, η decreases with the threshold.

Figure 2.4.5b shows the drop rate for the probe based protocol and for the sequential forwarding with $M = 1, 2$. The drop rate of sequential forwarding is not affected by η, while it increases for the probe-based protocol. The value $\eta^* \approx 1.8$ is a cross point, after which the probe-based algorithm performs worst than the simpler sequential forwarding, i.e. its dropping rate is higher. The following table provides some example of critical values of the schedule lag for $M = 1$.



(a) Normalised schedule lag vs job size.

(b) Drop probability vs η

Figure 2.4.5

	K=6	K=8	K=10
0.9	1.90	3.4	4.9
0.95	0.69	2.0	3.5
0.98	0.13	0.54	1.8

Table 2.6: Examples of critical schedule lag

The Response time for the same setting is reported in Figure 2.4.6 and clearly it increases with M. In the simulation section we will discuss a way to limit this issue.

2.4.3 Simulation results

To provide an additional performance evaluation of the proposed load balancing algorithm, taking into account additional parameters and scenarios, we rely on a discrete event simulator based on the Omnet++ framework[9]. The load balancing algorithms are implemented in an additional module specifically developed. A specific additional module implements a dummy load balancing, namely *NoLB*, that processes locally every received request.

In the proposed analysis, we take advantage of the insight provided by the model described in Sec. 2.4.2 and we aim to capture the impact of the network delay on the load balancing effectiveness, pointing out under which conditions each considered load balancing option is preferable.

2.4.3.1 Experimental setup

In the experiments, we consider the probe-based and the sequential forwarding algorithms. Furthermore, we consider also the *NoLB* dummy algorithm that does not perform any load balancing.

Throughout the experiments, we consider that in the probe-based algorithm, the query/response for probing incurs into a delay that we quantify as T_P, while the job forwarding introduces an additional

[9] https://omnetpp.org/

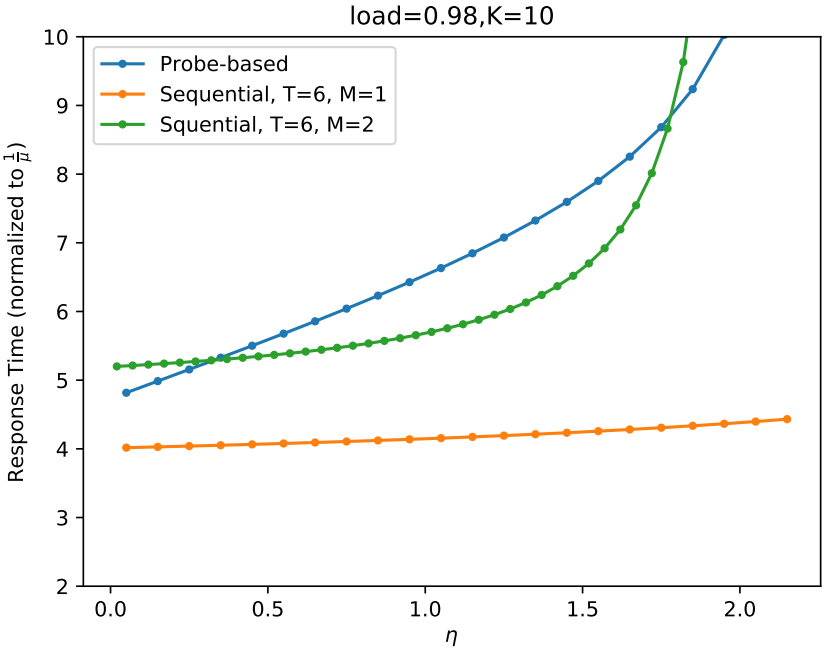


Figure 2.4.6: Total delay vs job transmission time for the sequential forwarding and the probe-based algorithms.

delay T_F, as shown in Figure 2.4.3. This delay accounts for the schedule lag discussed in the model. On the other hand, the sequential forwarding algorithm requires a delay that is just T_F; but the forwarding can occur up to M times looking for a randomly-selected neighbour that is not overloaded. In the NoLB, no forwarding and no probing occurs.

In all the experiments, we model the nodes as servers with an M/G/1/K queue. The incoming load is represented as a stream of incoming jobs with an inter-arrival time exponentially distributed. The service time is modelled according to a log-normal distribution, with a standard deviation that is comparable with the average value.

Finally, we consider that each node has a queue of finite size K. In the experiments, we set to $K = 9$. In this the system capacity is 9 jobs in the queue plus one job being executed way, that is consistent with the theoretical model of Sec. 2.4.2 (The system capacity is 9 jobs in the queue plus one job being executed). This assumption is consistent with application scenarios characterised by soft real-time requirements where long delays are not acceptable. The specific setting $K = 9$ is the result of an initial analysis where we explore the impact of the queue length on the algorithms' characteristics.

In the proposed analysis we consider two scenarios, namely *uniform mesh*, and *geographic*.

The first scenario, namely *uniform mesh* consider a mesh of uniform nodes that have the same incoming load λ, and the same service rate μ. The inter-arrival time of jobs follows a Poisson distribution, while the service time is based on a log-normal distribution with a standard deviation comparable with the average value. In the experiments, we focus on an overall utilisation $\rho = 0.9$ of the infrastructure to capture the case where load balancing becomes a critical component of the infrastructure. We do not explicitly report all the timings of the experiments as we consider more general to provide normalised results with respect to the average service time $1/\mu$. We assume that the network introduces a delay

normally distributed for both probing and job forwarding. Each probing phase (query and response) is characterised by an average delay equal to T_P, while the average job forwarding delay is T_F. Throughout the performance evaluation, we consider different values for these delays. In particular, we provide a comprehensive sensitivity analysis on the impact of parameter ηwe explore setups from $\eta \approx 0.1$, where schedule lag is significantly lower than the service time (for example, if the data to process is just an array of scalar values but significant mathematical analyses must be performed on these data), up to a case where $\eta \approx 10$ (for example if trivial computation must be carried out on large multimedia data). The intermediate case where schedule lag and service time are comparable is of particular interest in the area of Edge computing and IoT because it is a common situation when data are transferred on long-range, low-power wireless links for Edge-to-Edge communication [77] and must be processed on low-end devices. Another analysis we carried out is evaluating the impact of the probe time compared to the job forwarding time. To this aim, we perform a sensitivity analysis to the parameter $\zeta = T_F/T_P$ where the job forwarding time ranges from $1\times$ to $10\times$ the probe time. This latter analysis is particularly interesting to understand under which circumstances the overhead probe-based approach becomes overwhelming, compared to the faster sequential-forwarding alternative.

In the *geographic scenario*, we focus on a more complex setup derived from a realistic topology based on an ongoing project of traffic sensing in Modena, a city in northern Italy of roughly 180'000 inhabitants. The sensors are located in the main city streets and collect information about the traffic (for example, taking pictures of the street when movement is sensed to count how many cars are passing). Fog nodes are placed in facilities belonging to the municipality and exchange data using long-range wireless links (such as IEEE 802.11ah/802.11af [77]) to interact both with the sensors and among themselves. The scenario description is generated using the PAFFI framework [75]. In these links, the available bandwidth decreases with the distance. Hence, we assume the delay of each link to be directly proportional to the distance between the two communication endpoints. In this scenario we consider also the impact of network congestion, considering that probe packets and jobs must queue before being sent to the neighbour node. Each sensor is connected to the closest Fog node as in [38] so that the incoming load on each Fog node is highly heterogeneous, ranging from cases where the incoming load is more than $3\times$ the processing capacity to cases where a Fog node is nearly idle.

We summarise the main experimental parameters and performance metrics in Tab. 2.7 that expands the symbol list introduced in Tab. 2.5.

2.4.3.2 Uniform mesh scenario

We start the proposed analysis focusing on the uniform mesh scenario.

Figure 2.4.7 shows the response time of the probe-based and sequential forwarding algorithm for several values of the threshold Θ. The response time breakdown is provided to provide an insight into its components. The results are related to the scenario where $\eta = 1.1$, $\zeta = 10$, for the probe-based algorithms, while for the sequential forwarding, we have $\eta = 1.0$ (the job forwarding delay T_F is the same, but in sequential forwarding, we do not have the probing contribution T_P). However, even if the example is referred to a specific scenario, the main findings have general validity and confirm previous observation for the impact of load balancing [15]: as the threshold increase, we observe a reduction of the time spent in the load balancing phase, due to a less frequent activation of the algorithm, at the

Symbol	Range	Explanation
		Scenario parameters
Θ	$[1, 10]$	Load Balancing activation threshold.
η_J	$[0.1, 10]$	service time. similar to η, without including probing
ζ	$[1, 10]$	$\eta_J = T_F\mu$; used to compare the algorithms. impact of the cooperation delay compared to job forwarding delay ($\zeta = T_F/T_P$).
		Performance metrics
P_B		*Drop rate*: probability of a job being discarded because the queue of the selected Fog node is full.
T_{Resp}		*Response time* normalised against $1/\mu$.
T_{Srv}		*Service time*: time spent by jobs being processed; normalised to 1.
T_{Bal}		*Balancer time*: time taken for the load balancing jobs.
T_{Queue}		*Queuing time* that is the time spent in the Fog node ready queue waiting to be processed; normalised against $1/\mu$.

Table 2.7: Summary of simulation parameters and metrics

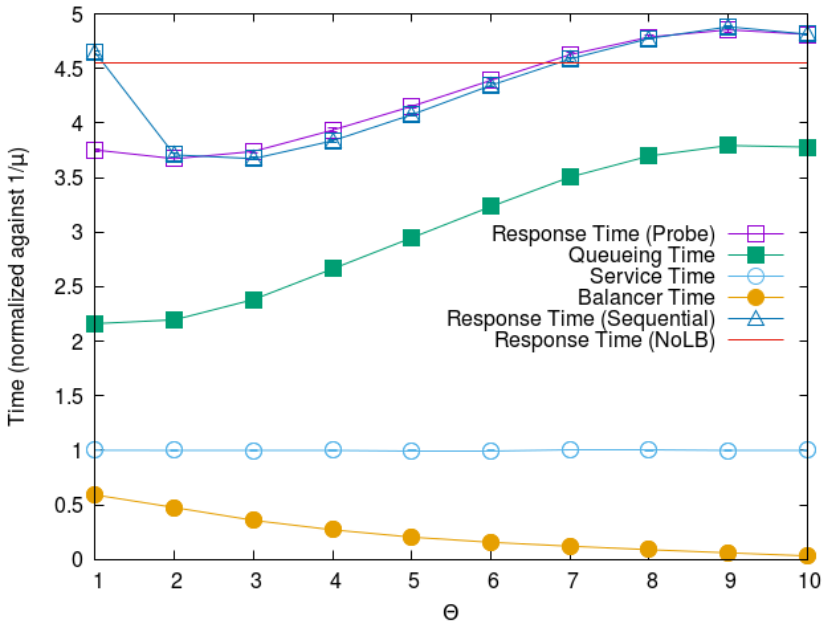
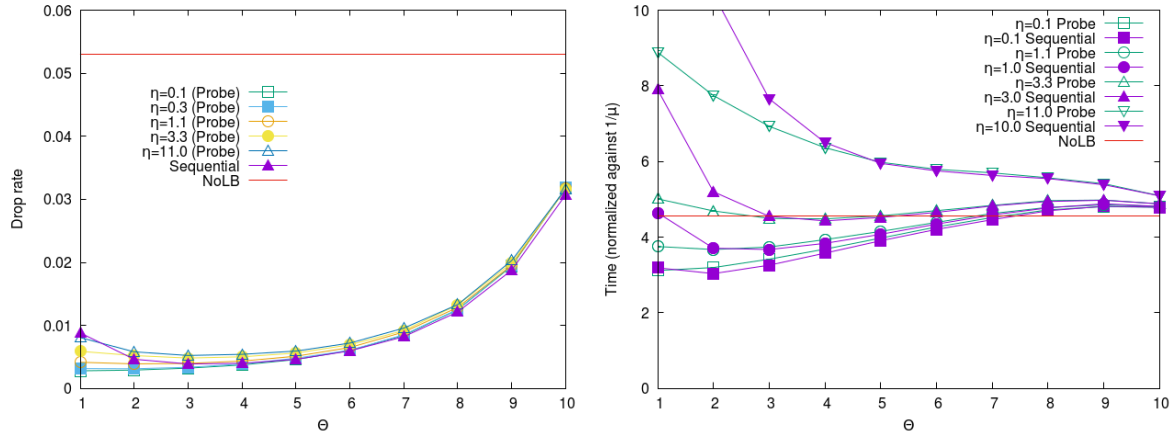


Figure 2.4.7: Response time *vs.* Θ

expense of a higher queuing time, due to potential queue build-up. We also observe that the sequential forwarding algorithm response time has a similar shape but is characterised by a higher variance with respect to the threshold Θ, suggesting the need for careful tuning of this parameter. Finally, in the case where no load balancing occurs *NoLB*, we observe that the response time is generally higher compared

with the alternatives. The response time is lower only for very high threshold values, where the load balancing algorithm seldom intervenes. This effect has already been observed in literature [15]



(a) Drop rate **(b)** Response time

Figure 2.4.8: Sensitivity to η ($\zeta = 10$)

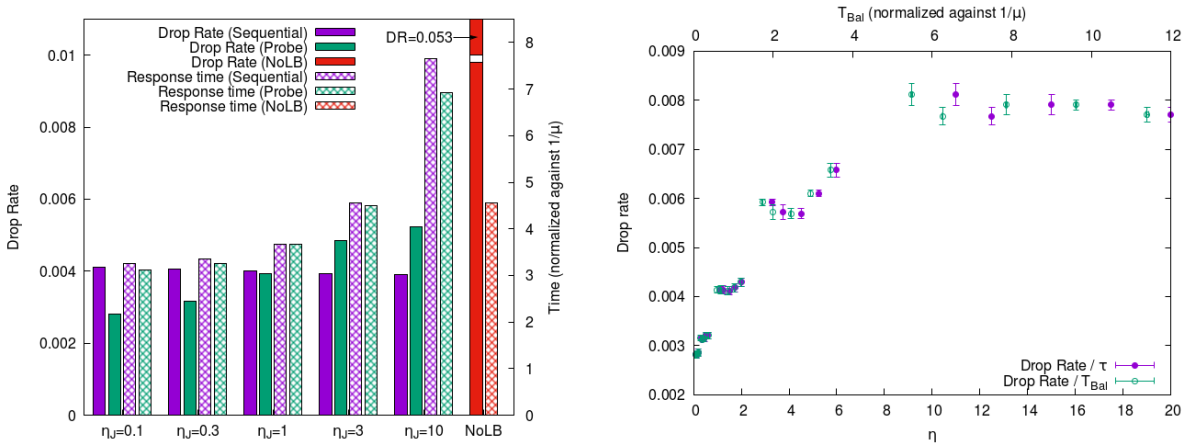
Figure2.4.8 provides a sensitivity analysis concerning the η parameter (the ratio between the schedule lag and the service time). In particular, Figure 2.4.8a shows the drop rate, while Figure 2.4.8b provides an analysis of the response time as a function of Θ for different values of η. In this analysis, we present results just for the case where $\zeta = 10$, for space reasons.

Focusing on Figure 2.4.8a we observe that, especially for low values of Θ and for low values of η (for example, in the curve marked with white squares), the probe-based algorithm outperforms the sequential forwarding alternative (reported as the curve with filled triangles). As Θ increases, the difference between the algorithms is reduced because the load balancing is activated less frequently and is, therefore, less effective. However, as η grows, the curve of the drop rate shifts from a monotone growing shape when $\eta = 0.1$ (meaning that the impact of delay is negligible) to a concave cup-shaped curve. This latter shape, which characterises the sequential forwarding algorithm, occurs when a job is sent to a randomly selected neighbour and is consistent with other results in literature [15]. This means that as the schedule lag grows, the load returned by the probing phase is less correlated with the load found on the node when the job is forwarded – ideally up to the point when the load encountered is completely unrelated to the probing result, reducing the probing to a random forwarding. This effect, already discussed in Sec. 2.4.2, is consistent with findings in other fields, such as the case of load balancing in Web servers [40]. The graph also shows the much higher drop rate that characterises the *NoLB* alternative: the fraction of dropped job is more than 5%, while the other load balancing solutions reach a drop rate typically below 1%.

Focusing on Figure 2.4.8b, we observe that the response time of the probe-based algorithm is generally lower compared with the sequential forwarding algorithm with the same threshold value, especially for low values of Θ when the load balancing is activated more frequently (again, when Θ grows, the difference between the two algorithms decreases). The case when η_J is very high (e.g. $\eta_J = 10$ – the curve with triangles) shows that load balancing is providing no actual benefit because the time to transfer the data is higher than the time to process them even with the queues are full

(the longest wait is, on average, K/μ that becomes comparable with T_F). The *NoLB* solution (red line) guarantees a response time that is lower compared to the load balancing alternatives as long as $\eta_J > 1$, thanks also to the high drop rate. However, when the network delay is not so overwhelming, the benefit of load balancing is evident also from the response time point of view.

Figure 2.4.9a summarises the analysis on the impact of η over the load balancing performance. In the graph, we present a group of histograms for each value of η_J (in this graph we focus on η_J rather than on η because the former parameter has the same value for both algorithms for each application setup). For each considered value of η_J, we present the drop rate of the sequential forwarding and probe-based algorithms and the response time of the two algorithms measured for the threshold where the drop rate is minimum. We observe that for the sequential forwarding algorithm, the minimum drop rate remains stable with respect to the network delay, as expected. On the other hand, the drop rate of the probe-based alternative increases. When the schedule lag becomes higher than the service time, the lowest achievable drop rate of the probe-base algorithm is worse than the sequential forwarding alternative. Considering the response time, we observe that the two algorithms have comparable performance unless the network delay is very high, in which case, the multiple load balancing hops in the sequential forwarding algorithm determine a clear performance penalty for this algorithm. The poor performance of the *NoLB* alternative are clearly visible. However, to make the figure more readable, the column of the drop rate is truncated. Indeed, the *NoLB* option is characterised by a drop rate that is more than $10\times$ compared with the load balancing algorithms.

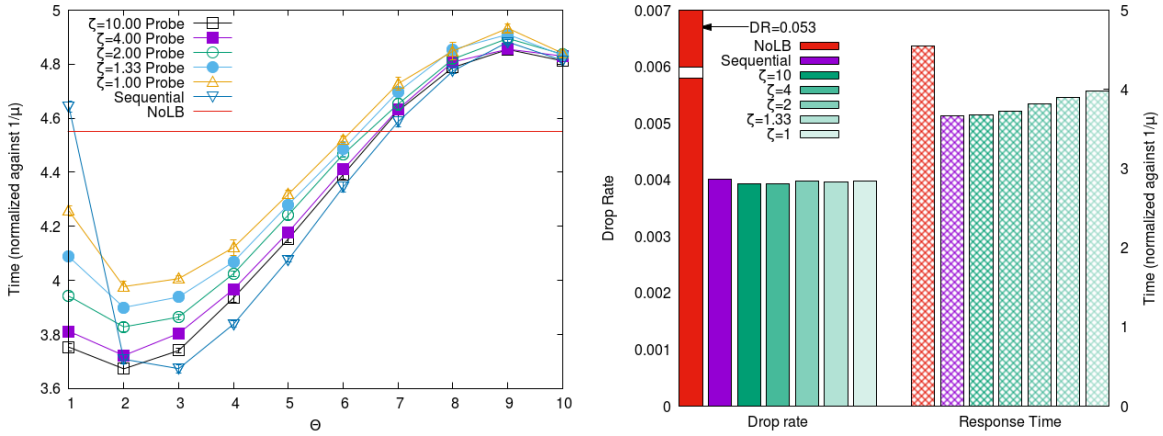


(a) Algorithm comparison with respect to η_J **(b)** Schedule lag *vs.* Drop rate

Figure 2.4.9

Figure 2.4.9b provides further proof of the impact of the delay associated with the load balancing on the drop rate of the algorithm. For these analyses, we focus on the case where $\Theta = 1$, because this is the situation where the impact of load balancing is more evident. In particular, we consider two different measures of the balancing-related delay: the first is η; the second is the time spent in the load balancer T_{Bal} that is highly correlated with the schedule lag. Both measures provide consistent results, demonstrating that, as the delay increases, the drop rate grows, as suggested by the model in Sec. 2.4.2.

As the last sensitivity analysis for the uniform mesh scenario, we consider the impact of the ratio ζ between the time for job forwarding and the probing. In this analysis, we keep constant η_J, while η



(a) Response time **(b)** Algorithm comparison

Figure 2.4.10: Sensitivity to ζ ($\eta = 1$)

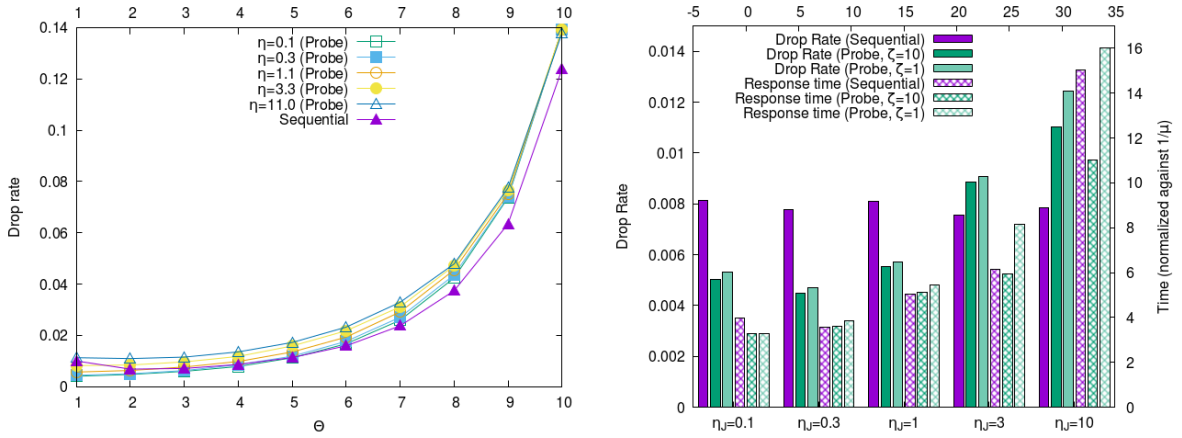
changes as we modify ζ. If we analyse the impact of ζ on the response time (Figure 2.4.10a) we observe that, as the probing time grows due to the reduction of ζ, the response time increases as well (for example from 0.038s to 0.043s, with an increment of 12% for $\Theta = 1$). Furthermore, as Θ grows, the reduction in the number of probes issued reduces the impact of the ζ parameter.

For a better comparison between the two considered algorithms, we present a histogram-based representation in Figure 2.4.10b, where the sequential forwarding is compared with several setups of the probe-based algorithm with different ζ. From the column on the left side of the graph, we observe that the drop rate remains unaffected by the slight increase in the network delay due to the probing overhead. On the other hand, in the right part of the graph, we show that, as the impact of the probing delay increases (ζ is reduced), the response time associated with the minimum drop rate grows accordingly. From this comparison, we can conclude that, when the probing time is comparable with the job forwarding time ($\zeta = 1$), which is common when the application is sending a limited amount of data to the Edge nodes, we can expect a performance drop in response time in the order of 8% compared to a case where the time to transfer the data is $10\times$ compared to the probe. This effect should be further considered when selecting the most appropriate protocol for load balancingwe also report the performance of the *NoLB* solution, confirming its poor performance in terms of both drop rate and response time.

2.4.3.3 Geographic scenario

We now focus on the final scenario, where the role of load balancing is crucial. In this scenario some node receives an incoming load more that $3\times$ w.r.t. their processing capacity, while other nodes are nearly idle. Without load balancing the high overload in part of the infrastructure can lead to a drop rate up to 30% (again we omit in analysis the results for the case where no load balancing occurs due to the overload conditions). This is the opposite situation compared to the mesh uniform scenario where load balancing must cope just with small temporary load fluctuations and, even in the worst conditions, the drop rate is below 5%.

Figure 2.4.11a provides the sensitivity analysis with respect to the η_J parameter. The analysis is similar to the one in Figure 2.4.8a. Even the presence of network effect, with the risk of congestion does not affect the general conclusions. However, we point out the main differences with the previously discussed mesh scenarios. It is interesting to observe that, in this highly skewed workload, reducing the intervention of the load balancing (for example, for $\Theta \geq 8$) determines a significant increase of the drop rate that rapidly grows beyond 5%. It is worth noting that only a small subset of the nodes experiences overloaded. Hence, only these nodes will issue probes very often. This reduces the loss of correlation in the state reported by probing and keeps the drop rate stable even when the threshold is very low (e.g., $\Theta = 1$), explaining the monotonic shape of the drop rate curve. This is a significant difference with respect to the previously considered scenarios, suggesting that, in the case of localised hot-spots, the problem of stale load information previously observed is much less critical.



(a) Drop rate *vs.* η_j ($\zeta = 10$)

(b) Algorithm comparison for η_J and ζ

Figure 2.4.11

We conclude the proposed analysis with the algorithm comparison of the best drop rate and for the response time when the drop rate is minimum, as in the previous analyses. Again, we refer to the η_J parameter for this study, and we consider two extreme values of ζ that are 10 (low probe impact) and 1 (high probe impact), respectively. Once again, as in this last set of experiments we aim to represent a scenario as realistic as possible, we consider that the network delay in Fog-to-Fog communication is caused by a bandwidth-constrained link, that can be subject to congestion.

In Figure 2.4.11b we observe that the minimum achievable drop rate of the sequential forwarding algorithm remains basically stable with the increasing network delay. On the other hand, for the probe-based algorithm, the delay has the already-proven negative impact on the drop rate. The ζ parameter has a limited effect on the drop rate, with the additional delay increasing slightly the drop rate due to its additional effect of the overall load balancing delay.

If we focus on the response time, we observe that the impact of ζ is much more significant: as the time for load balancing becomes comparable with the time for job forwarding, the delay experienced during the load balancing phase makes the probe-based algorithm slower than the sequential forwarding alternative. The effect is even worse when the network delay is higher than the service time. Further increasing the ζ and the η parameters would result in severe network congestion that can affect

negatively both response time and drop rate.

Chapter 3

Model-based approach

All models are wrong, but some are useful.

George E. P. Box

Mathematical models that have been presented in Chapter 2 are probabilistic models since they particularly fit the randomisation aspect of the protocols. In that models, Markov Chains were used to model the state of a node representing the number of tasks that are currently executing leveraging on M/M/1 and M/M/1/K queue models. In this Chapter, the approach followed is slightly different. Indeed, instead of relying on probability theory for modelling the system, we rely on dynamic systems theory. In the only work that belongs to the Chapter, we tried to model the function of the load as seen by every node, taking into account the cooperation for achieving the final goal of levelling the latency among all the nodes. Indeed, a node sees, as a load, the request rate coming from its clients and the rate that comes from the neighbours. However, at the same time, the node itself may decide to forward part of its load to neighbours according to specific ratios called *migration ratios*. By defining how the load changes over time (and therefore its "dynamics"), what we have is a system of ordinary differential equations (ODE) that, when it is solved, allows us to derive the trajectory of the migration ratios over time and find the solution at the convergence.

The algorithm and the results presented in this chapter (Section 3.2) has been published in [4].

3.1 Related Work

The main area in which the study presented in this chapter lies is the problem of load balancing in Edge and Fog computing [81], [82], [83] and [16]. In the aforementioned work, a QoS-oriented load

balancing algorithm is proposed. The approach targets the delay that users experience when using the deployed application. Similar works, like [84] propose the (OLBA) framework, which takes into account turn-around time and service delay and relies on Particle Swarm Optimisation (PSO) for finding the best load balancing strategy but the approach is not fully decentralised, the same approach is followed by [85]. Then, Tripathy et al. in [86] focus on the QoS parameters but in a smart city setting and a smart allocation scheme is performed through a genetic algorithm. However, the approach is not "online", and the scheduling decision is not taken for every task. More technological approaches instead, like the one proposed in [87], design algorithms specifically targeting well-known frameworks like Kubernetes. In that case, the authors propose a proxy-based approach that periodically monitors the pods' state, and according to the load, it forwards the requests to balance it; however, the approach does not consider node heterogeneity which can have the same load but generates different service latency. Similarly, Singh et al. in [88] propose a container-as-a-service (CaaS) load balancing strategy that is focused on energy efficiency, however, the approach is based on two steps service level agreement, while this study tries to use only one, moreover the results are only provided in simulations. A game theory-based approach is proposed by [89], however, no simulation or real implementation results are provided. Sthapit et al, in [90] propose a modelling of Edge computing layer as a set of queues and design a load balancing strategy which targets the job completion rate and the energy consumption, however, only simulation results are provided, like in [91].

A set of works, instead, focus on healthcare [92] and the "internet of healthcare things" [93]. For example, [94] proposes a load balancing framework which is able to avoid any failure in responsiveness and [95] which targets a smart city. Both approaches focus on the quality of service but they do not directly target the service latency, which is critical when having heterogeneous computing nodes.

By introducing even the Cloud layer [96] we increase the computation capability, although the cloud is not used in this study, we can still refer to the load balancing strategies offered by different works. For example, [97] proposes an Edge-Fog-Cloud algorithm for distributing the traffic in all of the three layers but the focus is not the latency optimisation, [98] provides a model based on queuing theory, [99] studies a load balancing approach for the Fog-Cloud environment classifying requests in real-time, important and time-tolerant but again the approach is not focused on latency levelling, then [100] proposes a scheduling approach based on blockchain and [101] a strategy to cope with failures by using Software-Defined Networks (SDN).

In conclusion, the last set of works worth mentioning focuses on load balancing by using intelligent approaches like reinforcement learning [102], [103], [7]. The heuristic proposed in the following Section 7 is not explicitly using reinforcement learning but it follows a strategy that mimics a learning process since the migration ratios are continuously adapted to meet a goal by using a learning rate α.

3.2 Latency-levelling load balancing algorithm modelled with a dynamical system

Service latency plays a crucial role in modern distributed applications [98]. In particular, in the Edge and Fog Computing environments, due to the geographic displacement of the nodes, some of them can be subjected to more traffic than others. In these situations, for designing an effective and QoS-oriented load balancing algorithm, it is not possible to consider only the typical hardware parameters that regard, for example, the CPU load, the RAM utilisation or the network traffic. This is because all of these performance indicators are both hardware and application-agnostic, they do not consider that the devices may be heterogeneous and the same application on different hardware performs differently. Suppose that we have two Edge or Fog nodes Node A and Node B with two different CPUs, CPU A and CPU B respectively. Suppose that we designed an algorithm that enables nodes to cooperate, and some nodes can forward part of their flow of tasks to be executed to another node. Also, suppose that we designed an algorithm which is able to level the CPU time and in the end both CPU A and B are levelled to 50%. If there are no differences in network delays, what can we say about the application that is running on both devices? Will the users that make task requests to Node A experience the same service latency as the ones that will make requests to Node B? Yes, but only in one case, the CPU A must be equal to CPU B, a characteristic of the system which is not common in Edge or Fog computing and even if we deploy the same hardware, we will never have exactly the same performances, due to background processes of the OS and intrinsic hardware differences. Given these conditions, it is necessary to change the performance indicators which drive the balancing, we need to design an algorithm which is able to balance the QoS that each user will experience: each user, independently from the node at which it will request the service, will have to experience the same service latency. The latency can be intended as a performance parameter which best describes how the application is behaving, independently of the effective load situation. Therefore by levelling the latency of the service, we will probably not balance the CPU load. Indeed, slower devices will be, in general, less loaded than the faster ones because they will saturate when the load is lesser than the faster ones. But in general, we will be sure that each user will experience the same QoS as the others since there will be no user that will experience a higher or a lower service latency than the other. The motivation of this study is clear, and the principal focus is designing, in a fully decentralised environment (that particularly fits the Fog and Edge Computing models) with no central entity, a load balancing algorithm that is able to level the service latency across all the nodes by tuning the percentage of tasks that a node can forward to another, a percentage that we call the *migration ratio*. In other words, each node can decide if and at which level it can cooperate with others offloading part of its work for reducing its service latency until it reaches a stable value that is equal across all the neighbours when this is possible, or at least closer to the value of the others.

The contributions of this Section can be summarised as follows.

- A continuous-time model which describes the dynamics of the system by using a system of differential equations that reaches stability when all the nodes experience the same service latency;
- An heuristic algorithm which tries to find a solution to the problem in a real environment by

continuously adapting the migration ratios in rounds of fixed duration;

- Simulation results of the proposed heuristic algorithm;
- Results of the implementation of the proposed algorithm in a testbed of Raspberry Pis which shows the efficacy of the solution even in a real setting.

Symbol	Meaning
	Model
$\mathcal{N}$	Set of nodes
A	Adjancency matrix
a_{ij}	Cell of the adjacency matrix that is 1 if node i can communicate with node j, otherwise 0
λ_i	Traffic to node i (in reqs/s)
μ_i	Service rate of node i (in reqs/s)
K_i	Maximum queue length for node i
$l_i(t)$	Service latency of node i at time t
$l_{a_i}(t)$	Average service latency between node i and its neighbours at time t
$m_{ij}(t)$	Percentage (of λ_i) of tasks forwarded from node i to node j at time t
	Adaptive Heuristic (Algorithm 7)
M	Matrix of migration ratios
m_{ij}	Current percentage (of λ_i) of tasks forwarded from node i to node j
α	Step size
ϵ	Tolerance of the average for which the algorithm stops the updating of the migration ratios (balance zone)
T	Round duration
	Trajectories and Experiments
d_t	Average service latency
d_a	Average service latency among all the nodes

Table 3.1: List of symbols used

3.2.1 Performance Model

In the presented model model, we suppose to have a set $\mathcal{N}$ of nodes, whose network topology is described by the adjacency matrix A, in particular, given any two nodes i and j, they can communicate only if $a_{ij} = a_{ji} = 1$ since we always suppose that the communication between nodes is bi-directional. Each node i receives a fixed traffic rate of requests λ_i req/s from the underlying clients and it is able to execute μ_i req/s, moreover, a node i is able to forward part of its load λ_i to a given neighbour node j, and we do not consider the network communication latencies. We call the percentage of forwarded requests from node i to j the "migration ratio" and it is expressed as m_{ij}. We also stress the fact that a node i cannot forward the load that it receives from other nodes and it can only forward the one from the clients, that is λ_i.

We now want to mathematically model the system and for doing so, we define which is the total load of a node i over time, and we call this function $x_i(t)$ that models the state node i in a given time t:

$$x_i(t) = \lambda_i - \sum_{j \in V} a_{ij}\lambda_i m_{ij}(t) + \sum_{j \in V} a_{ji}\lambda_j m_{ji}(t) \tag{3.1}$$

where the following conditions must be followed

$$0 \leq m_{ij}(t) \leq 1, \quad \sum_j m_{ij}(t) \leq 1 \qquad \forall\, i, j, t \tag{3.2}$$

and the initial condition, at $t = 0$, since $m_{ij}(0) = 0\ \forall i, j$ is

$$x_i(0) = \lambda_i \qquad \forall\, i \tag{3.3}$$

Equation 3.1 can be interpreted as follows. A node i, receives constant traffic by the clients that are connected to it, that is λ_i, then a part of this traffic can be forwarded to the neighbour nodes (for which $a_{ij} \neq 0$) and it is subtracted, but neighbour nodes may also decide to forward part of their traffic to i and this part is added to the total load of the node. For any node i, the functions $m_{ij}(t)\ \forall j$ describe the portions of incoming traffic λ_i that are forwarded to the neighbour nodes and they are the unknowns. By knowing the $m_{ij}(t)$, we will then need to find a time t^* where $m_{ij}(t) = m_{ij}(t^*),\ \forall i, j, t > t^*$, and the values $m_{ij}(t^*)\ \forall i, j$ will be the final migration ratios that each node will need to apply to reach the final goal. At this point, we need to model this final goal: the levelling of latencies. For finding the functions $m_{ij}(t)$, instead of trying to define them directly, it is easier to describe their variation over time, for this reason, we calculate the derivative with respect to the time of Equation 3.1 that is:

$$\dot{x}_i(t) = -\sum_{j \in V} a_{ij}\lambda_i \dot{m}_{ij}(t) + \sum_{j \in V} a_{ji}\lambda_j \dot{m}_{ji}(t) \tag{3.4}$$

Equation 3.4, describes the dynamic of the state of node i, that is how the load that every node i sees at time t changes over time. The formulation can be repeated for every node, thus we have a system of $|\mathcal{N}|$ Ordinary Differential Equations (O.D.E.). Before solving the system, we need to define the functions $\dot{m}_{ij}(t)$ that are still unknown but we remind that the solution to the system will allow us to know the original $m_{ij}(t)$.

Basically, we define the $\dot{m}_{ij}(t)$ as the multiplication of three factors logically derived from the fact that the objective is that, in every node, every task must have the same duration, and therefore the average service latency of each node must be the same. Moreover, we need to keep in mind two essential behaviours of the entire system: (i) when a node i migrates a portion of the incoming traffic to another node j the node i will see its average service latency decrease, while in the node j the average task service latency will increase. This is because the service latency function is a monotonically increasing function with respect to the load of a node. In the case, we suppose, for simplicity, that nodes can be modelled as M/M/1/K queues and the service latency at time t of node i can be expressed as (given $\rho_i(t) = x_i(t)/\mu_i$):

$$l_i(t) = \frac{1 - (K_i + 1)\rho_i(t)^{K_i} + K_i\rho_i(t)^{(K_i+1)}}{\mu_i(1 - \rho_i(t))(1 - \rho_i(t)^{K_i})} \tag{3.5}$$

Then (ii) the average delay between neighbours nodes plays a crucial role, because the average service latency of a given node can be higher or lower than the average, and trying to level them to the average proved to be the key strategy to solving the problem. But how we can level them to the average? There are three sub-strategies that we need to adopt to reach the goal and they concretize in

three factors:

1. the tasks migration must be performed only if the delay of the current node i, $l_i(t)$, is greater than the average delay between itself and its neighbours, called l_{a_i}, for this reason the first factor is:
$$\dot{m}_{ij}^{\alpha}(t) = \max\left[0, \frac{l_i(t) - l_{a_i}(t)}{l_i(t)}\right] \tag{3.6}$$
2. the tasks migration must be performed only if the delay of the current node i, $l_i(t)$, is greater than the delay of its neighbour j, $l_j(t)$, and therefore:
$$\dot{m}_{ij}^{\beta}(t) = \max\left[0, \frac{l_i(t) - l_j(t)}{l_{h_i}(t)}\right] \tag{3.7}$$
3. the tasks migration must be performed only if the delay of the neighbour node j, $l_j(t)$, is lesser than the average delay between node i and itself, and therefore:
$$\dot{m}_{ij}^{\gamma}(t) = \max\left[0, \frac{l_{a_i}(t) - l_j(t)}{l_{k_i}(t)}\right] \tag{3.8}$$

The final dynamic of the migration ratios is therefore

$$\dot{m}_{ij}(t) = \dot{m}_{ij}^{\alpha}(t) \cdot \dot{m}_{ij}^{\beta}(t) \cdot \dot{m}_{ij}^{\gamma}(t) \tag{3.9}$$

and the idea behind the formulation is that the dynamic of the state $\dot{x}(t)$ stops when at least one of them becomes zero, both for the received load and the forwarded one.

As already mentioned, the $l_{a_i}(t)$ is the average delay between the current node i and its neighbours:

$$l_{a_i}(t) = \frac{l_i(t) + \sum_{j \in V; i \neq j} a_{ij} l_j(t)}{1 + \sum_{j \in V} a_{ij}} \tag{3.10}$$

Finally, $l_{h_i}(t)$ and $l_{k_i}(t)$ are the summations of the differences over time:

$$l_{h_i}(t) = \max\left[0, \sum_{j \in V} l_i(t) - l_j(t)\right] \tag{3.11}$$

$$l_{k_i}(t) = \max\left[0, \sum_{j \in V} l_{a_i}(t) - l_j(t)\right] \tag{3.12}$$

We will resort to numerical calculus to find the time trajectories of the system of non-linear ODE described in Equation 3.4 with initial conditions $x_i(0) = \lambda_i,\ \forall i$ but unconstrained for simplicity. Then, after finding the numerical solution $x_i(t), \forall i$, we can easily find the effective behaviour of migration ratios over time considering that:

$$m_{ij}(t) = \int_0^t \dot{m}_{ij}(\xi)\, d\xi \tag{3.13}$$

We will consider the trajectory of the solution valid until the condition expressed in Equation 3.2 is respected.

3.2.1.1 Latency-levelling property

We now prove that when the trajectories of the solution of the system at Equation 3.4 converge, then the latencies are aligned to the same value. In the Appendix at Section 3.2.4 we instead prove the existence and the uniqueness of a set of steady states $x_i \forall i$ for which the latencies are levelled.

Theorem 3.2.1. *If the solution's trajectories of the O.D.E. system at Equation 3.4 converges, i.e.* $\exists\, t^*$ *s.t.* $\dot{x}_i(t) = 0\ \forall t > t^*, \forall i$ *then all the nodes have the same service latency, i.e.* $l_0(t) = l_1(t) = \ldots = l_i(t)\ \forall i$ *and this latency is the average latency* $l_{a_i}(t^*)$ *among all the neighbours of each node* i *at time* t^**.*

Proof. We can prove the theorem by contradiction. Suppose that the system solution converged at t^* but there exists one node i that has not the same service latency as the other nodes, i.e. $l_i(t) \neq l_j(t), \forall j \neq i, \forall t > t*$. We can distinguish two possible cases, for any $t > t^*$:

(a) $l_i(t) > l_{a_i}$, i.e. the service latency of node i is *higher* than the average latency between i and its neighbours, we point out that every other neighbour node's latency can be higher, equal or lower than the average latency but at least one node must have the latency below the average. From this fact we have that the $\dot{m}^{\alpha}_{ij}(t) \neq 0$ by definition, $\dot{m}^{\beta}_{ij}(t) \neq 0$ and $\dot{m}^{\gamma}_{ij}(t) \neq 0$ because there exist at least one node with average service latency below the average and the same node's latency is also lower than the latency of node i. This means that, from Equation 3.4 the negative part is not zero, the positive part instead is zero since i is the only one node with latency higher than the average it will not receive traffic from any neighbour. Therefore we showed that $\dot{x}(t) \neq 0$ for some $t > t^*$, and this is a contradiction ※;

(b) $l_i(t) < l_{a_i}$, i.e. the service latency of node i is *lower* than the average latency between i and its neighbours. As in the case (a) if the node i's latency is below than the average latency then there exists at least one neighbour j whose latency is higher than the average. The consequences are exactly the ones of case (a) and we proved the contradiction ※ ;

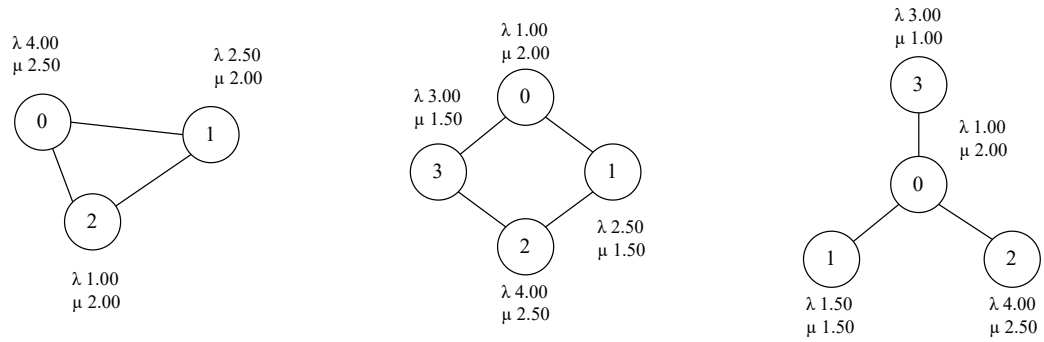
□

From these two cases emerges that the only possible case is that $l_i(t) = l_{a_i}$ and no other node can have a service latency that is higher or lower than the average l_{a_i}.

3.2.1.2 Trajectories and Topologies

We will now explore some configuration of nodes and parameters that we will reuse later in the simulations and in the experimental setting, the general idea is to show how this model can predict quite well the behaviour of a real system. The crucial point for the results to match is the alignment of the service latency, but the alignment value and the migration ratios may differ as will be clearer later. In this section,we study the behaviour of the latency over time $d_t(t)$, computed by using the Equation 3.5 and the migration ratios $m_{ij}(t)$ computed by using Equation 3.13.

We tested different network topologies, the first three are shown in Figure 3.2.1. These small topologies are taken into consideration because it is easy to have a direct comparison with the behaviour shown in simulations and in the real deployment. Finally, we tested a fully connected topology with 15 nodes.



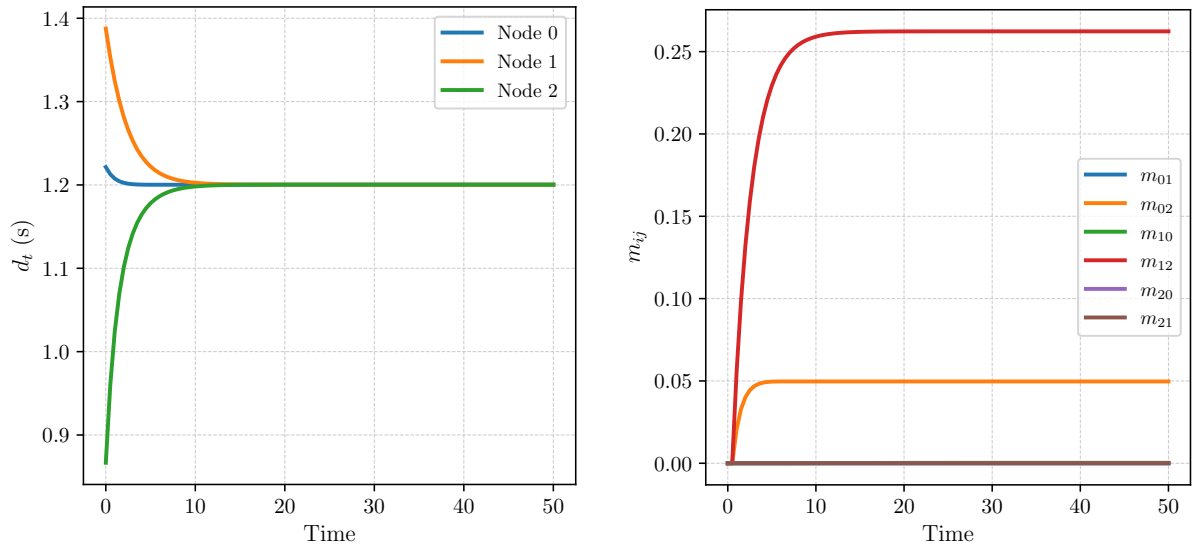
(a) Topology A **(b)** Topology B **(c)** Topology C

Figure 3.2.1: The nodes topology and parameters configuration used across the mathematical model, the simulations and the final experimental setting.

Topology A (Figure 3.2.1a) is composed of three nodes arranged in a fully connected graph, the Figure 3.2.2a shows the trajectories of the latency and the migrations ratios m_{ij} of the nodes. As we can observe, after the transient the system reaches the steady-state at about $t = 15$ where the latencies are levelled at 1.2s. From the migration ratio, Figure 3.2.2b we can observe that the Node 1 gives 25% of its load λ_1 to Node 2 since it has the higher service latency at $t = 0$ and part of its load is forwarded to the node that is below the average service latency, that is Node 2. Node 2 only has to receive load while Node 0 and Node 1 have to lose their load in order to balance the service latency, indeed even Node 0 forward exactly the 5% of its load to slightly reduce the service latency.

Topology B (Figure 3.2.1b) comprehends four nodes connected as a ring, the Figure 3.2.1b shows the numerical trajectories of the the performance parameters. Each node, from 0 to 3, starts with service latency, respectively, 0.86s, 2.06s, 1.22s and 2.18s and the end of the transient (Figure 3.2.3a) is levelled to 1.38s. At steady state and we can observe how (Figure 3.2.3b) Node 3 forwards about the 65% of its traffic to nodes 0 and 2 for lowering the latency, the same is done by Node 1 which forwards a total of about 60% of its load to Nodes 2 and 1, then Node 2 does not forward tasks because already close to the average latency while starting from $t = 10$ Node 0 starts to forward tasks to its neighbours up to the 10%. This means that the Node 1 must give back part of the load to Nodes 1 and 3 but these nodes already forwarded part of their load to Node 0, this behaviour is justified by the fact that the derivative of migration ratios functions $\dot{m}_{ij}(t)$ are always positive, therefore the only way for diminishing them is making a node to give back the load to the sender.

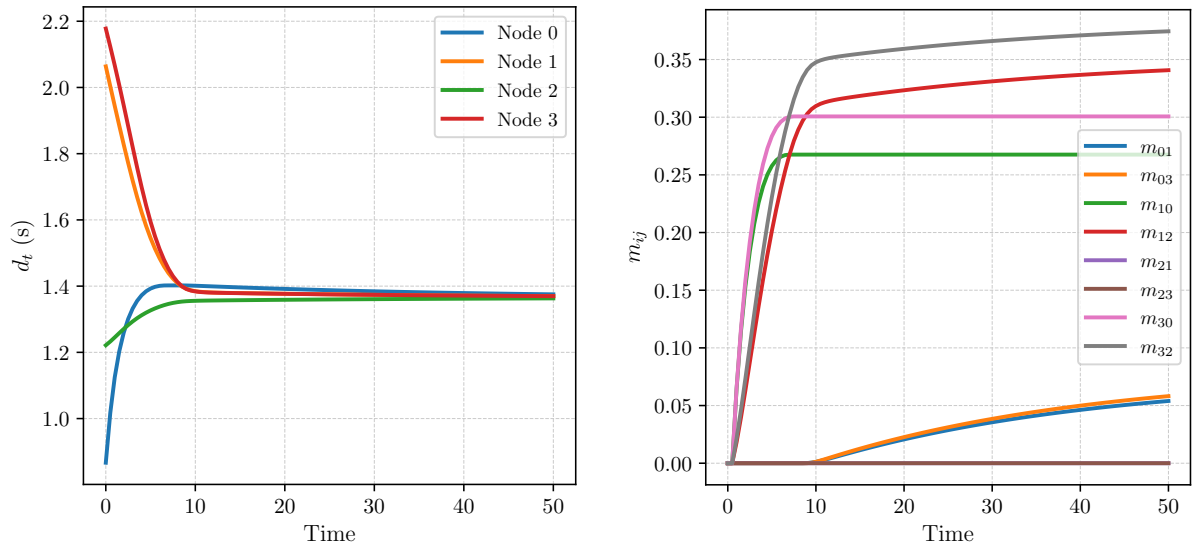
Topology C (Figure 3.2.1c) is a star topology and includes four nodes, but this particular configuration of the nodes is more challenging because one single node is connected to all the others while the others are only connected to the same node, and therefore the node at the centre can be overwhelmed by the load of the others. However, the model converges to a levelled latency of 1.4s (Figure 3.2.4a) but the



(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.2: Trajectories of the average latency d_t and the migration ratios m_{ij} for the three nodes described by Topology A (Figure 3.2.1a).

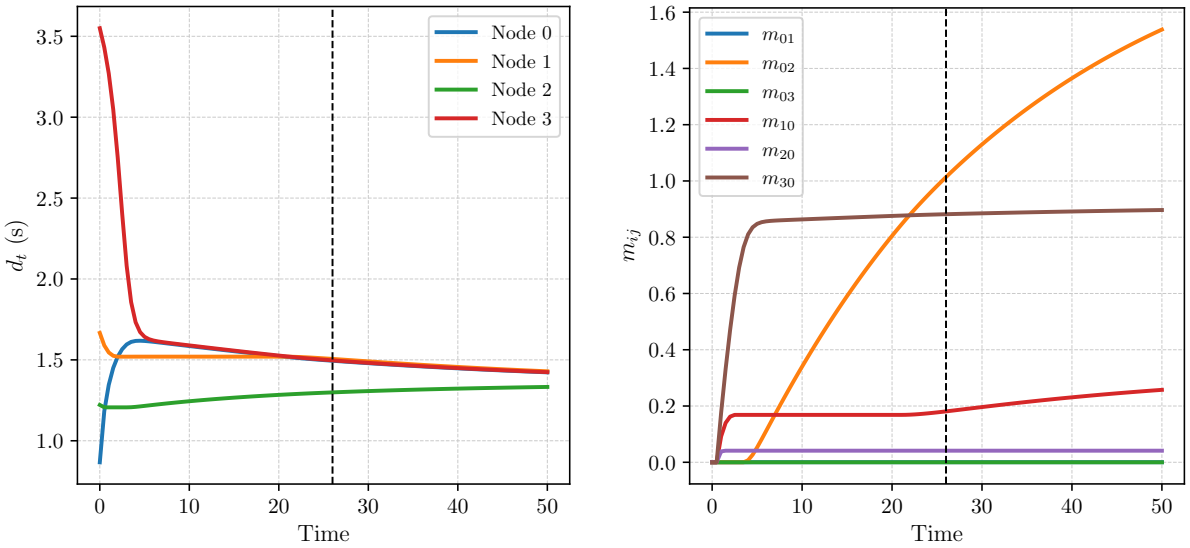


(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.3: Trajectories of the average latency d_t and the migration ratios m_{ij} for the four nodes described by Topology B (Figure 3.2.1b).

solution that is reached is actually not achievable because the condition expressed at Equation 3.2 is no more respected (Figure 3.2.4b), since the model is unconstrained. This does not mean that we cannot use the solution, indeed, it is sufficient to consider the transient as long as the condition is still met, i.e. at $t = 26$ and consider the migration ratios there. What is clear is that the exact levelling of the latency is not feasible but considering the solution, at $t = 26$ we still reached a point in which the latencies are closer, even if they do not exactly match. In particular, we recall that in this solution, the node m_{02} is required to forward all of its traffic λ_0 and execute only the traffic forwarded by the other nodes.



(a) Service latency d_t (s)

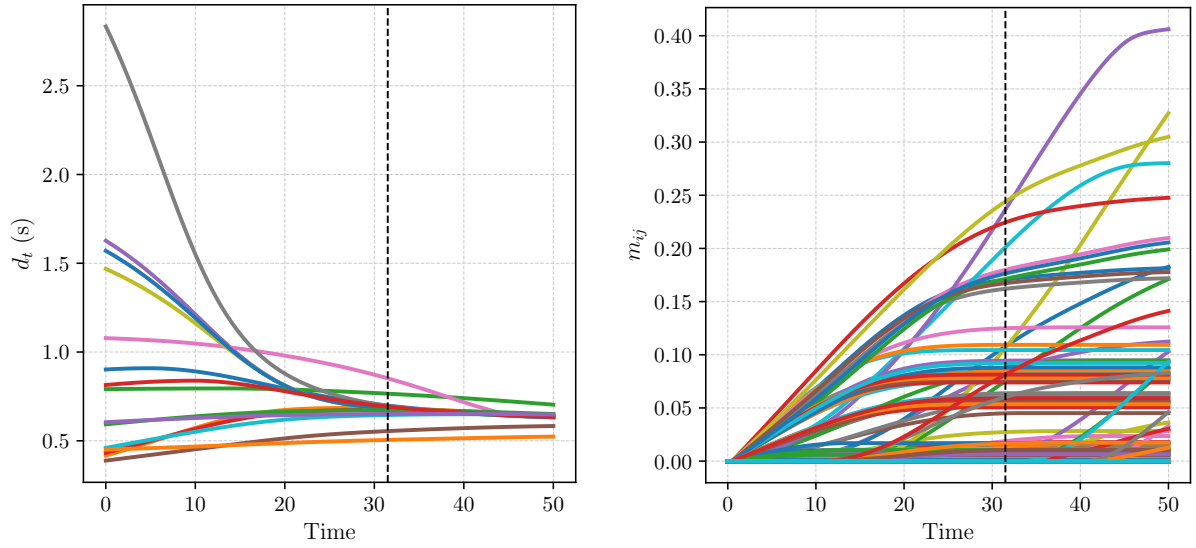
(b) Migration ratios m_{ij}

Figure 3.2.4: Trajectories of the average latency d_t and and the migration ratios m_{ij} for the four nodes described by Topology C (Figure 3.2.1c).

The last topology that we tested comprehends instead 15 nodes in a fully connected topology with $0 \leq \lambda_i \leq 4$, $0 \leq \mu_i \leq 4$ and $2 \leq K_i \leq 6$. All of these parameters are picked at random, but the purpose of this is to understand how the system behaves with many nodes. The SageMath[1] Python ODE solver took about 20 hours to derive the trajectories up to $t = 100$ with the numeric solver Runga-Kutta-Felhberg on a Ryzen 9 5800X processor. Figure 3.2.5a shows the behaviour of the latency for all the nodes and as we can see the system reduce their variance, but again we need to cut the solution at time $t = 31$ because $\sum_j m_{ij}(t) \geq 1$ for some i when $t \geq 31$ (3.2.5b).

This last result shows how the model scales with the number of nodes, however, we do not envision modelling a system of more than 20 Edge or Fog nodes, because aligning the latencies in a very large set of nodes may not be the best strategy for balancing the load. As we can see, some nodes can be obliged to forward all of their traffic and if the parameters λ_i and μ_i are particularly different then it would not be possible to level the latencies, without counting the difficulties of implementing the algorithm in the real world where the network latencies have a significant impact. Instead, it is more efficient to create groups of a maximum of 20 nodes and try levelling the latencies within the groups, these groups can, for example, represent neighbourhoods of a smart city.

[1] `https://www.sagemath.org/`



(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.5: Trajectories of the average latency d_t and the migration ratios m_{ij} for 15 nodes in a fully connected topology.

3.2.2 Adaptive Heuristic

We now want to effectively implement a strategy for levelling the latency among the nodes. The mathematical model tells us what are, at steady state, the migration ratios m_{ij} $\forall i, j$ but calculating them requires finding the trajectories of the model. Moreover, there are other 3 points that motivate the design of an algorithm. First of all, (1) the mathematical model assumes that we know the state of every node but in the real world, we want to have a fully decentralised approach, each node should be able to see the only state of its neighbours and tune the migration ratios accordingly, also that state must be explicitly requested when needed. Then (2) real nodes may be subject to variation in load conditions over time, thus the algorithm should react and re-tune the migration ratios to cope with the changes. As the last point, (3) the model does not take into account the communication latencies that exist between the nodes. Therefore, we now propose an adaptive strategy which follows a heuristic approach to find the most suitable set of migration ratios for every node in such a way the latency is made equal when it is possible or at least closer when it is not feasible.

The Figure 3.2.6 summarises the logic behind the heuristic. Firstly, we suppose to divide the time into rounds of T seconds each. The Algorithm 7 is run every time a round ends and has as a final objective the one of modifying the migration ratios when it is needed. We also divide the algorithm into steps for describing the rationale behind its design. The input that it takes comprehends the index of the current node i in which the algorithm is executed (we remind that the algorithm is fully distributed, there is no central entity or coordinator), the step size α, the set of nodes $\mathcal{N}$, the vector of migration ratios $\vec{M}_i$ which describe the percentage of tasks that is forwarded to each (neighbour) node, percentage on the average latency that defines the balancing zone ϵ and the incidence vector for node i that is $\mathcal{I}_i$ and describes which are the neighbours of the current node. Suppose that the round time T

just elapsed, and the algorithm does the following:

1. first of all, the node computes the average latency between itself and all the neighbours, moreover, it computes the upper and lower average limits by multiplying the average latency by $1 \pm \epsilon$, these limits allow us to relax the constraint that each node must exactly match the latency of each other, which in real scenarios is very unlikely due to the arrivals' distribution. As the last step, it is also computed the sum of all the migration ratios, which cannot exceed 1.0;
2. once the average is known, we proceed to the adjusting of the migration ratios; the first check that we perform is to see if the current node is below the average and if it is migrating tasks to other nodes. Indeed, if this happens, then it means that the node is forwarding too much traffic to the others. We remind from the mathematical model, that the strategy for making the algorithm work is that a node can only receive or give traffic to others at the same time, and, in general, only the nodes that are above the average must forward tasks to the ones that are below. Thus, a node that is below the average and it is giving traffic to others must reduce the ratios in such a way its average returns the balance zone ($d_{a_i} \pm \epsilon$). This is what the algorithm does in during this step for all the neighbours nodes by previously checking if the ratio given to the node does not reach negative numbers and this is done by using the auxiliary functions described in Algorithm 8. If the adjustment is done, the function returns with no further steps;
3. at this point, we check if the average latency of the current node is below the high level of the average zone, because if this is true then it means that the node latency is in the average zone, then no further action is needed;
4. if we reach this step, then the node's latency is out of balance, i.e. it is above the high level of the zone, then we need to adjust the migration ratios for every neighbour node, but we can distinguish the following two cases:
   (a) if the average latency of neighbour node j is above the balance zone, then we reduce the migration ratio towards it of the step size α since it means that we are forwarding too much traffic;
   (b) if the average latency of the neighbour node is below the balance zone, then we increase the migration ratio towards it of the step size α, this will cause the latency to be reduced and its one to increase, approaching the balancing zone.

3.2.2.1 Simulations results

We now show some results of the proposed algorithm in a discrete event simulator written in Python by using the library "Simpy"[2] and published as open source[3]. We will use the same topologies and parameters (Figure 3.2.1) used in for computing the trajectories of the mathematical model in order to have terms of comparison.

In the simulator, we again assume no communication delays between the nodes and the same nodes are modelled as M/M/1/K queues since the objective of simulations is to understand if the migration

[2] `https://pypi.org/project/simpy/`
[3] `https://gitlab.com/gabrielepmattia/simulator-2022-mswim`

Algorithm 7 Adaptive Heuristic for leveling latencies

Require: $i, \alpha, \mathcal{N}, \vec{M}_i, \epsilon, \mathcal{I}_i$
currentNode $\leftarrow \mathcal{N}[i]$
[*1. Compute the average latency among all the neighbour nodes*]
averageLatency $\leftarrow$ currentNode.latency
numberOfNeighbours $\leftarrow 0$
for all j in $|\mathcal{N}|$ **and** $\mathcal{I}_{ij} \neq 0$ [*Loop over the neighbours*] **do**
averageLatency $\leftarrow$ node.latency **and**
numberOfNeighbours $\leftarrow$ numberOfNeighbours + 1
end for
averageLatency $\leftarrow$ averageLatency / numberOfNeighbours
averageLatencyLow $\leftarrow$ averageLatency $\cdot\ (1.0 + \epsilon)$
averageLatencyHigh $\leftarrow$ averageLatency $\cdot\ (1.0 - \epsilon)$
totalRatiosGiven $\leftarrow \sum_j m_{ij}$
[*2. If under average and migrating, then reduce migration*]
if currentNode.getAverageLatency() $\leq$ averageLatencyLow **and** totalRatiosGiven > 0 **then**
for all j in $|\mathcal{N}|$ **and** $\mathcal{I}_{ij} \neq 0$ **do**
if $\mathcal{N}[j]$.getAverageLatency() $\geq$ averageLatencyHigh **then**
if canBeSubtractedToNode(j, α) **and** canSubtract(α) **then**
$m_{ij} \leftarrow m_{ij} - \alpha$
totalRatiosGiven $\leftarrow$ totalRatiosGiven - α
end if
end if
end for
return
end if
[*3. If latency below the high zone limit, then the node is balanced*]
if currentNode.getAverageLatency() $<$ averageLatencyHigh **then**
return
end if
[*4. If latency greater or equal the high limit we need to migrate*]
for all j in $|\mathcal{N}|$ **and** $\mathcal{I}_{ij} \neq 0$ **do**
[*4a. Reduce the ratio to neighbour above the average high limit*]
if $\mathcal{N}[j]$.getAverageLatency() $\geq$ averageLatencyHigh **then**
if canBeSubtractedToNode(j, α) **and** canSubtract(α) **then**
$m_{ij} \leftarrow m_{ij} - \alpha$
totalRatiosGiven $\leftarrow$ totalRatiosGiven - α
end if
end if
[*4b. Increase the ratio to neighbour below the average low limit*]
if $\mathcal{N}[j]$.getAverageLatency() $\leq$ averageLatencyLow **then**
if canBeGiven(α) **then**
$m_{ij} \leftarrow m_{ij} + \alpha$
totalRatiosGiven $\leftarrow$ totalRatiosGiven + α
end if
end if
end for

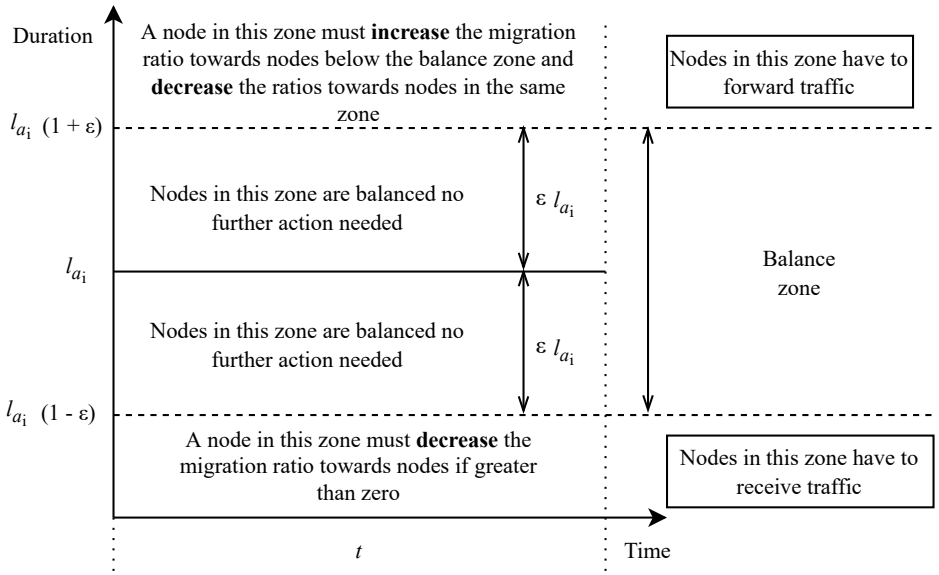


Figure 3.2.6: Representation of the logic behind the adaptive heuristic for a node i in a given time t. We suppose the average delay l_{a_i} between the node i and its neighbours to be fixed during an instant time t.

Algorithm 8 Auxiliary functions

Require: $i, \alpha, \mathcal{N}, \vec{M}_i, \bar{z}, \underline{z}, \mathcal{I}_i$, totalRatiosGiven

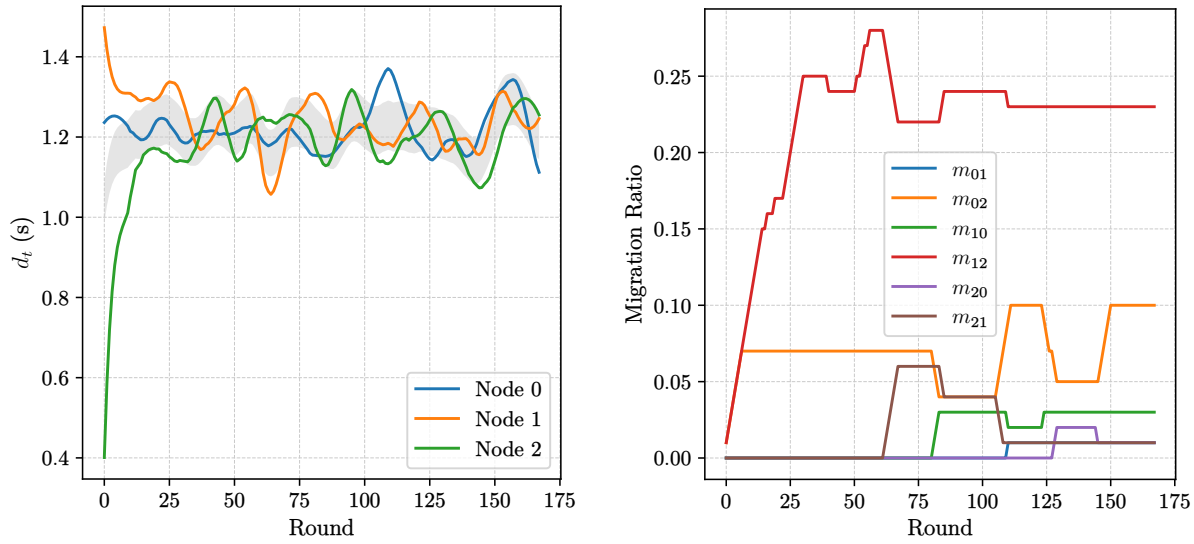
```
[Check if the specified amount of ration can be given]
def canBeGiven(alpha: float): boolean
  return totalRatiosGiven + alpha ≤ 1.0
end def

[Check if the specified amount of ratio can be subtracted]
def canSubtract(alpha: float)
  return totalRatiosGiven - alpha > 0.0
end def

[Check if the specified amount of ration can be subtracted to a node]
def canBeSubtractedToNode(j: int, alpha: float)
  return m_ij − alpha > 0.0
end def
```

ratios found by the heuristic match the model. All the tests are done with the simulator to use a round time $T = 60s$ and the behaviour of the average latency are filtered with a Savitzy-Golay filter with window size 20 and polynomial degree of 4. Moreover, the balance zone uses $\epsilon = 0.05$, the step size $\alpha = 0.01$ and $K_i = 4 \; \forall \; i$. A peculiar characteristic of the simulator is that the average latency is computed as the average of the last 10 rounds, this is done in order to stabilize the curves, otherwise due to the exponential distribution of the inter-arrival times and of the execution times the average latency may be subjected to significant variations.

Figure 3.2.7 shows the results of the simulations of Topology A. First of all, we can observe how after 25 rounds, the average latency starts to stabilize at about 1.2s (Figure 3.2.7a), we have highlighted in grey the balance zone that is the average delay $d_a \pm \epsilon$ and in the chart the average it is computed across all of the nodes. We can notice how the latency result is perfectly matching the model compared to Figure 3.2.2a, the fluctuations around the average is due to the exponential inter-arrival times and execution times. For levelling the latency the migration ratios found by the algorithm are represented in Figure 3.2.7b. In particular, we can observe that m_{12} stabilizes at around 0.24 and m_{02} at around 0.07 while the others are less than 0.03. Again these result matches the ones of the model shown in Figure 3.2.2b, in which m_{12} and m_{02} stabilizes at 0.26 and 0.05 respectively, while the others are set to 0.

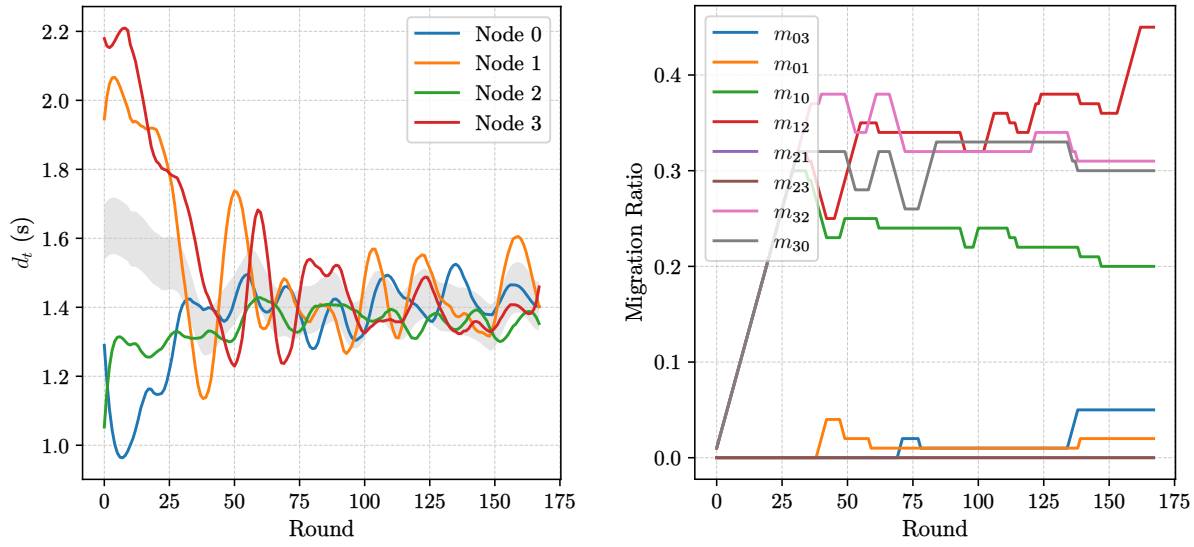


(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.7: Behaviour of the average latency d_t and migration ratios for Topology A (Figure 3.2.1a) in the simulated environment.

Topology B results are shown in Figure 3.2.8. As far as regards the average service latency (Figure 3.2.8a) we can observe how it stabilizes at about 1.4s which is in line with the mathematical model shown Figure 3.2.3a. The same holds for the migrations ratios, for example, the Node 0 gives 5% of the λ_0 to its two neighbours respectively that match the model, Node 1 gives about 20% of its traffic to Node 0 but the model 26% and about 45% to Node 2 while the model 34%. The same slight differences hold for Nodes 3 and 4 and are due to the traffic variability.



(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.8: Behaviour of the average latency d_t and migration ratios for Topology B (Figure 3.2.1b) in the simulated environment.

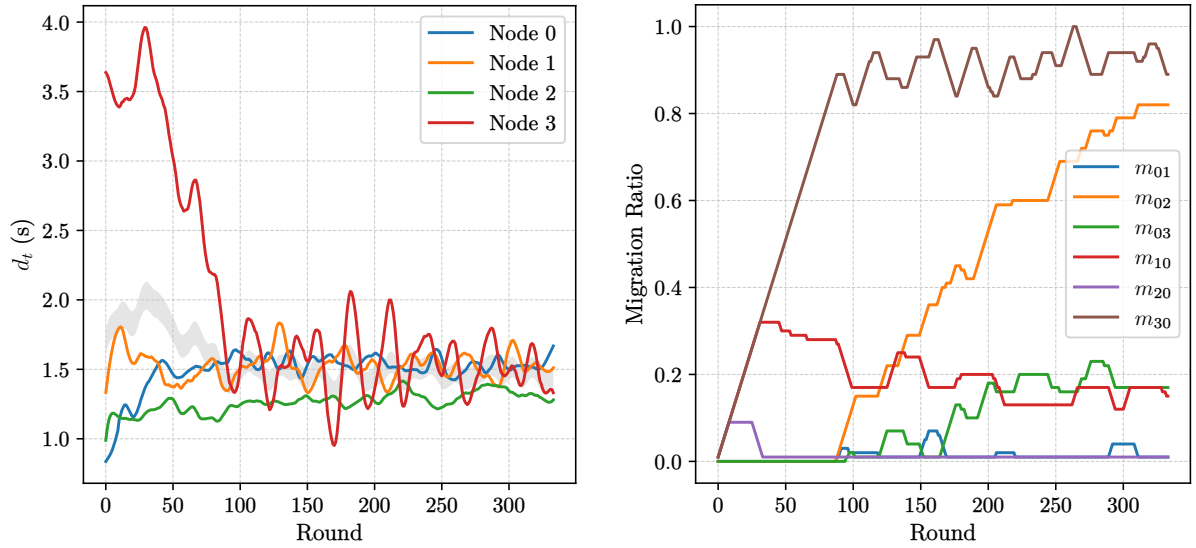
Topology C results are shown in Figure 3.2.9. Regarding the service latency (Figure 3.2.9) we can see how it does not converge to the same value for each node, and this behaviour is the same presented in the model in Figure 3.2.4a where we truncated the trajectory at $t = 26$. Indeed, the same values are obtained in the simulation, Node 0, 1 and 3 align at about 1.5s while Node 2 stabilizes to 1.3s because it cannot receive enough traffic from Node 0 in order to increase its latency to match 1.5s. This does happen in the model after $t = 26$ but Node 0 would forward more traffic than the one that is available. Regarding instead the migration ratios, shown in Figure 3.2.9b, we can observe that as the latency, they match with the truncated solution of the model (Figure 3.2.4b) with slight differences. In particular, m_{30} reaches 0.9, m_{02} reaches 0.9 while in the model 1.0, then m_{03} and m_{10} reach 0.2 respectively while in the model 0.0 and 0.2.

3.2.3 Experimental Setting

After testing the proposed adaptive heuristic in simulations, we finally implemented it in a testbed of Raspberry Pi 4[4] connected with Gigabit ethernet to a dedicated subnet. Each node implements a Python web server based on the Flask[5] library, that once deployed with Docker, receives the traffic from a machine that acts as a traffic generator. The source of the application is published as open source[6]. The webserver implements the scheduling decision, indeed, when a new task arrives, it decides to execute it locally or forward it to another neighbour node according to the current configuration of migration ratios. Migration ratios are updated according to Algorithm 7 every T seconds

For implementing the tasks of variable duration we used a loop that performed the same operation

[4] `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/`
[5] `https://pypi.org/project/Flask/`
[6] `https://gitlab.com/gabrielepmattia/framework-2022-mswim`



(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

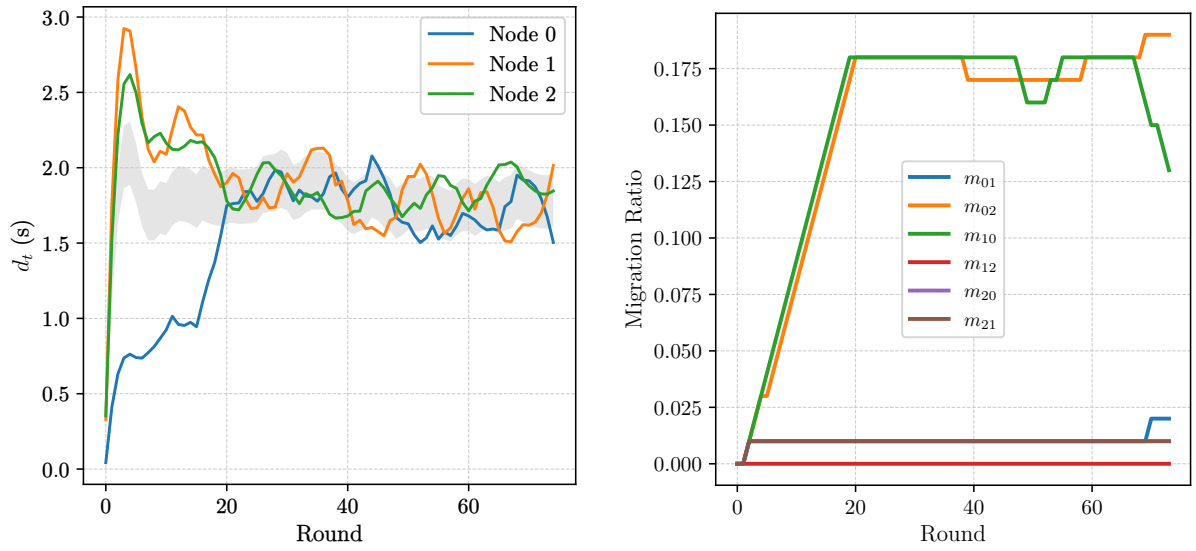
Figure 3.2.9: Behaviour of the average latency d_t and migration ratios for Topology C (Figure 3.2.1c) in the simulated environment.

repeated a fixed amount of times, we measured the duration of a single iteration and from there we compute the number of iterations to match the desired μ_i parameter for each node. The operation carried out in the loop is the computation of the SHA-512 hash of the same (20 bytes) string. We measured that, the operation in question, in a Raspberry Pi 4 has an average duration of $4.721\mu s$ (on 30'000 iterations repeated 10 times). Therefore, for example, setting $\mu = 2$ is equal to perform $(1/2)/4.721^{-6} \approx 105'900$ loop iterations.

Deployment The deployment process involves two phases. (I) After the container is started in every node, the webserver is put on wait for the configuration that is passed via POST. The configuration is a JSON file where the main parameters are declared, for example, the queue length K, how many rounds are used for computing the average latency, the round duration T and the balance zone size ϵ. This structure contains also some parameters that regard the identification of the node: the IP, the ID, the name, μ, the step size α and λ. The last part regards the topology of the network that defines with which nodes the communication is possible. After the configuration is received (II) each node starts 2 threads: the *update* thread that is in charge of updating the migration ratios at every round and collecting all statistics parameters used by the algorithm as service latency, the number of executed tasks and the queue length; and the *worker* thread that is in charge executing a service execution request by picking the first available from the internal queue. Now the node is ready to receive the requests from the task generator and the adaptive heuristic (Algorithm 7) updates the migration ratios accordingly every T seconds.

Results All of the topologies shown in Figure 3.2.1 have been run in the above-mentioned framework, we will now illustrate the results obtained. In all the experiments we set $K_i = 4$, $\forall\, i$, the round time

$T = 30$s, the tolerance $\epsilon = 0.1$, $\alpha = 0.01$ and all the curves have been filtered with the Savitzy-Golay filter by using window size 20 and polynomial degree 4. The Figure 3.2.10 shows the behaviour of the average service latency and of the migration ratios for the Topology A (Figure 3.2.1a). Regarding the latency (Figure 3.2.10a) we can observe how the alignment value is slightly different from the model (Figure 3.2.2a) and the simulations (Figure 3.2.7a), in particular, the average service latency is levelled to 1.7s and this represents an increase of 0.5s with respect the other tests, but as we can notice the latency at round 1 is not matching the simulations nor the model and this is justified by the fact that the model of the queue M/M/1/K is not representing well the behaviour of a real node, moreover we ignore the eventual background work of the CPU that may interfere with tasks that we are sampling. However, the algorithm manages to level the latencies among all the nodes but with migration ratios that are different from the model. Indeed, in Figure 3.2.10b we can observe how the Node 0 forwards about the 17.5% of its traffic to Node 2 and the Node 1 forwards about the same amount of traffic to Node 0. This solution found by the heuristic is quite different from the one predicted because we point out that the solution, i.e. the combination of m_{ij} ratios may not be unique.



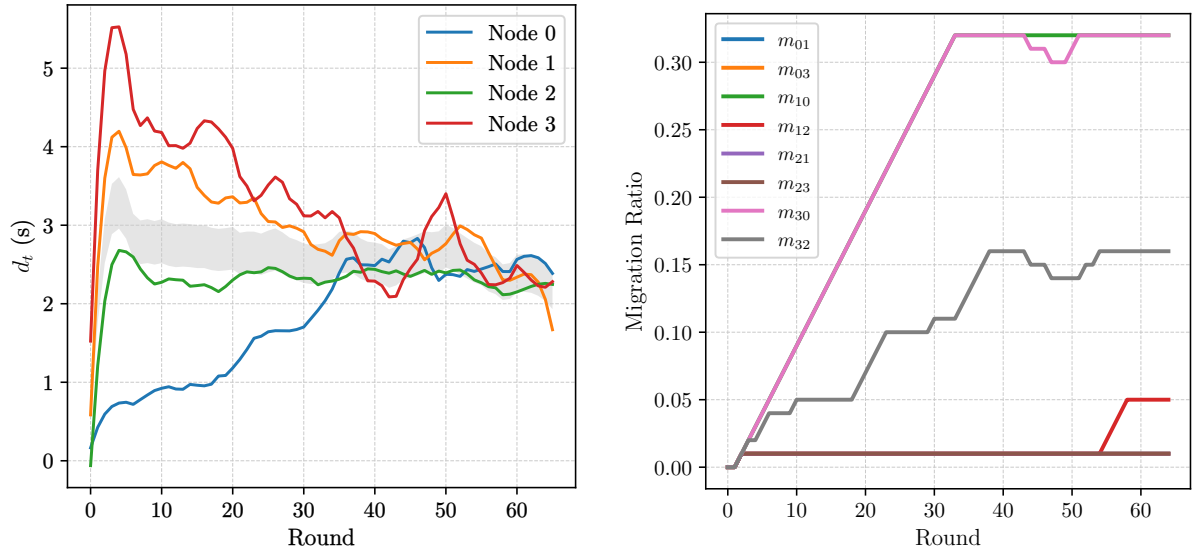
(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.10: Behaviour of the average latency d_t and migration ratios for Topology A (Figure 3.2.1a) in the experimental setting.

The Figure 3.2.10 shows the behaviour of the average service latency and of the migration ratios for the Topology B (Figure 3.2.1b). As the previous result, the final alignment latency is again different, we pass from 0.8s, 4.1s, 2.6s, 5.5s (respectively from Node 0 to 3) to 2.5s for each node with respect 1.5s in the model and in the simulations. The algorithm manages to level the latency by making Nodes 1 and 3 forward about the 30% of their traffic to Node 0, and Node 3 forward the 15% of its traffic to Node 2 at steady state.

The final test on the real deployment regards Topology C (Figure 3.2.1c) and its result is shown in Figure 3.2.10. As we can observe, latencies (Figure 3.2.12a) are higher than the ones predicted of 1.5s, however, the final result is the same, since Nodes 0, 1 and 3 are aligned while Node 2 instead cannot



(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.11: Behaviour of the average latency d_t and migration ratios for Topology B (Figure 3.2.1b) in the experimental setting.

reach the alignment latency (see Figures 3.2.5a and 3.2.8a). This is also reflected in the migration ratios (Figure 3.2.12b) in which we have the Node 3 which forwards the 70% of its load to Node 0 while the Node 0 will try to forward all of its traffic to Node 2, even if the Figure is cut to $t = 120$.

Concluding, the results in a real testbed of Raspberry Pi showed how the adaptive heuristic algorithm allows reaching the final goal of levelling latencies with a behaviour that was predicted both in the model and in the simulations. However, due to the absence of a more precise model of a real node, the predicted alignment latencies and migration ratios are not the same but this does not limit the applicability of the proposed heuristic, rather the tests showed how it can work even in a real deployment.

3.2.4 Appendix

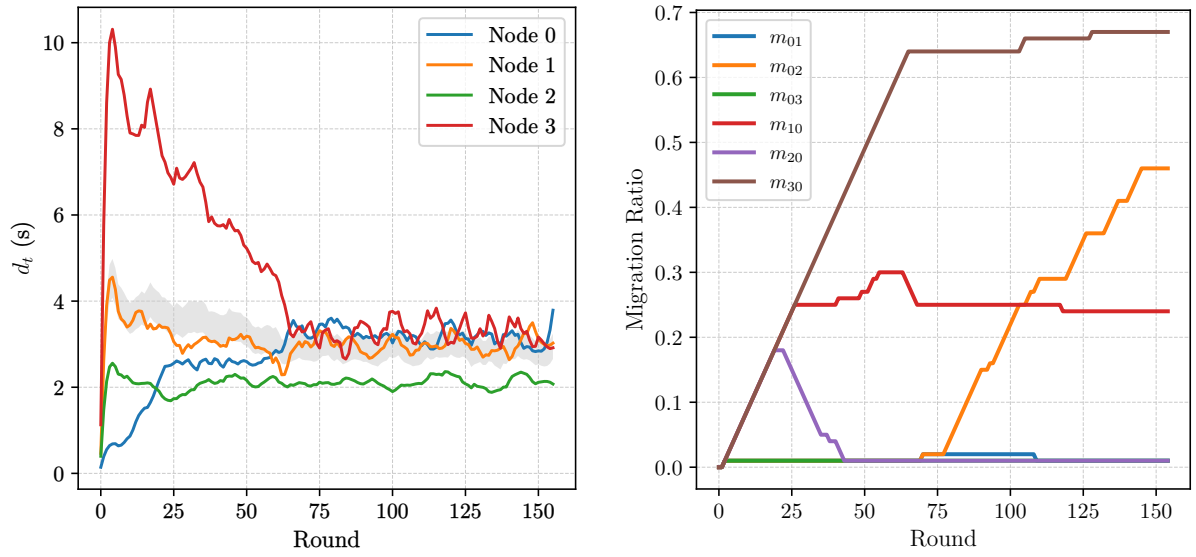
In the following, we consider a completely connected topology, and a generic load-delay relationship $f_i(\lambda)$ which is a monotonically increasing continuous function, with $f(0) = 0$ and $\lim_{\lambda\to\infty} f_i(\lambda) = d_{M_i}$. The transmission delay is not considered, but the same proof sketch can be used by adding the transmission delay to the definition of f_i.

Property 3.2.1 (Existence of balanced loads)**.** *Given a vector* $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_N)$ *of loads,* $|\Gamma| \stackrel{def}{=} \sum \lambda_i = \lambda_T$ *it there exists another vector* $\Lambda' = (\lambda'_1, \lambda'_2, \ldots, \lambda'_N)$ *such that:*

$$f_1(\lambda'_1) = f_2(\lambda'_2) = \ldots = f_N(\lambda'_N) = d, |\Gamma| = |\Gamma'|$$

Proof. Let consider the function:

$$\lambda'_T(d) = f_1^{-1}(d) + f_2^{-1}(d) + \ldots + f_N^{-1}(d)$$



(a) Service latency d_t (s)

(b) Migration ratios m_{ij}

Figure 3.2.12: Behaviour of the average latency d_t and migration ratios for Topology C (Figure 3.2.1c) in the experimental setting.

and let $d_M = min\{d_{M_i}\}$. Due to the property of f, this function is continuous and increases monotonically with d; moreover, $\lambda'_T(0) = 0, \lambda'_T(d_M) = \infty$. Since $\lambda'_T(0) - \lambda_T < 0, \lambda'_T(d_M) - \lambda_T > 0$, due to the Bolzano's theorem, it there exits a value $d < d_M$ such that $\lambda'_T(d) - \lambda_T = 0$, i.e. $\lambda'_T(d) = \lambda_T$. □

Property 3.2.2 (Unicity of balanced loads). *For a given d, the vector Λ' is unique.*

Proof. Follows from the properties of f_i. □

Property 3.2.3 (Migration). *Given two load vectors of size N, Λ and Λ', $|\Lambda| = |\Lambda'|$, where Λ' is such that $f_i(\lambda'_i) = d > 0$, it there exists at least an $N \times N$ migration matrix M such that:*

$$\Lambda' = \mathbf{M}\Lambda$$

where $0 \leq m_{ij} \leq 1, \sum_i m_{ij} = 1$.

Proof. Since $|\Lambda| = |\Lambda'|$, Λ can be partitioned in two sets, $\mathcal{A}$ and $\mathcal{B}$, namely the set of nodes such that $\lambda'_i \leq \lambda_i$ and the set of nodes such that $\lambda'_i > \lambda_i$ - unless $\Lambda = \Lambda'$ in which case $\mathbf{M} = \mathbf{I}$.

First of all, observer that all nodes in $\mathcal{B}$ have $\lambda'_j > \lambda_j$, so we can set $m_{ij} = 0, j \in \mathcal{B}, i \in \mathcal{A}, m_{jj} = 1$ (these nodes only receive load from others).

The other values $m_{ij}, j \in \mathcal{A}, i \in \mathcal{B}$ are constrained as following:

$$\lambda_i = \lambda_i + \sum_{j \in \mathcal{A}} m_{ij}\lambda_j \;\; i \in \mathcal{B}$$

There are $B = |\mathcal{B}|$ of these equations, each with A unknowns, $A = |\mathcal{A}|$. In addition there are other A constrains on the coefficients, i.e. $(1 - \sum_i m_{ij})\lambda_j = \lambda'_j$. Of these equations one is redundant since it

must be $\Sigma_i \lambda_i = \Sigma_i \lambda'_i$, so only $A + B - 1$ are truly independent. Overall, we have AB unknowns and $A + B - 1$ equations. Since $A + B = N$, $AB \geq A + B - 1$, i.e. the unknowns are at least equal to the number of constrains. The system of equations has then either one solution or it is undetermined and infinity solutions exist. Since each node $j \in \mathcal{A}$ migrates a fraction $m_j < 1$ of λ_j towards nodes of $\mathcal{B}$ (this is true since $f_i(0) = 0$ and $d > 0$) $m_j = \frac{\lambda'_i}{\lambda_i} < 1$ and all the coefficients are <1. □

Example of migration matrix with $N = 4, \mathcal{A} = \{1, 2\}$ showing 4 unknowns:

$$\begin{pmatrix} \lambda'_1 \\ \lambda'_2 \\ \lambda'_3 \\ \lambda'_4 \end{pmatrix} = \begin{pmatrix} 1 - m_{31} - m_{41} & 0 & 0 & 0 \\ 0 & 1 - m_{32} - m_{42} & 0 & 0 \\ m_{31} & m_{32} & 1 & 0 \\ m_{41} & m_{42} & 0 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{pmatrix}$$

Chapter 4

Reinforcement Learning based strategies

> Our intelligence is what makes us human, and AI is an extension of that quality.
>
> — Yann LeCun

Machine learning changed the perspective of decisional algorithms. In particular, the extraordinary flexibility of Reinforcement Learning made it a very powerful tool to be used when we want to create a decision mechanism that is efficient and adaptive over time. In this Chapter, we present two works that tried to improve the performances of *power-of-n* strategies by creating agents that are able to implement the direct forwarding to a given node without probing the state. The main idea was that Reinforcement Learning is able to infer the state of another node according to the reward that it receives when a task is scheduled for it. This not only eliminates the need for probing but also creates a solution that is adaptive over time. Results show, for example, that when a node goes down, and it starts to reject all of the requests, the learned scheduling policy is automatically adapted to compensate for the loss of the node, moreover, it returns to the previous state when the faulty node is restarted.

This Chapter is organised as follows. Section 4.1 presents the related works, Section 4.2 shows an extension of the *power-of-n* strategy presented in Chapter 2 with Reinforcement Learning specifically targeting Fog Computing in a Smart City setting, instead in Section 4.3 a QoS approach is presented, where RL is again used for online scheduling of tasks, but it is programmed for matching as much as possible users FPS requirements.

Part of the work presented in Section 4.2 has been published in [7], while the work presented in Section 4.3 has been published in [104].

4.1 Related Work

The principal topic of this Chapter is the problem of scheduling and load balancing in Fog or Edge computing environments by using Reinforcement Learning. This problem has its roots in the classic "job-shop" scheduling problem which has countless declinations and particularly fits the Fog computing paradigm, due to its distributed nature. However, one of the first attempts to solve it with a learning approach was introduced in 1999 by [105]. From there the scenario changed radically with the development of new machine learning techniques and with the spread of distributed systems which concretised in the Fog and in the Edge computing layers [2].

In Table 4.1, we offer a summary of the related works classified according to the following criteria: if they deal with (a) online or offline scheduling according to the fact that the scheduling decision is taken on a per-task basis (online) or for a group of tasks (offline), if they consider a task deadline (b), if they use a geographic set up (c), and finally, if they provide (d) a real or pseudo-real implementation results and they may propose a framework in which to run the algorithm. Moreover, we specify which algorithm they use for finding the policy (i.e., Q-Learning, Sarsa, A3C) and how they solve the RL problem (for example by using the Q-Table or Deep Neural Networks).

	(a) o.	(b) d.	(c) g.	(d) i.	RL Policy Alg.	RL Solver
[106]	✓	✓	-	-	Q-Learning	Two DNNs
[107]	✓	-	-	-	Q-Learning	Two-level DNNs
[108]	-	-	-	-	DDPG	Two DNNs
[109]	✓	✓	-	-	Custom	Single DNN
[110]	✓	-	-	-	Custom	RNN
[111]	✓	-	-	-	Q-Learning	Two DNNs
[112]	-	-	✓	-	Q-Learning	Single DNN
[113]	-	-	-	-	Q-Learning	Single DNN
[114]	✓	✓	-	-	Q-Learning	Q-Table
[115]	✓	✓	-	-	Q-Learning	Q-Table
[116]	-	✓	-	-	A3C	Residual RNN
[117]	-	-	-	-	Q-Learning, Sarsa	Q-Table
[105]	-	-	-	-	Actor-Critic	Residual RNN
[118]	-	-	-	-	Q-Learning	Two DNNs
[119]	✓	-	✓	-	Q-Learning	Q-Table
[120]	-	-	-	-	MRL	Custom
[121]	-	-	-	-	Custom	Two DNNs
[122]	-	-	-	-	Actor-Critic	Custom
[123]	-	-	-	-	Q-Learning	Q-Table
[124]	-	-	-	-	Q-Learning	DNNs
[125]	-	-	-	-	Q-Learning Custom.	DNNs
[126]	-	-	-	-	Actor-Critic	DNNs
[127]	-	-	-	-	DDPG	Two DNNs
[103]	✓	-	-	-	Q-Table	Q-Learning
[128]	-	-	✓	-	DDPG	Custom
[102]	-	-	-	-	Q-Table	Q-Learning
[129]	-	-	-	✓	-	-
this	✓	✓	✓	✓	Sarsa	Lin. Approx. + QTable

Table 4.1: Summary of related works to scheduling solution with reinforcement learning in Fog or Edge computing. The criteria listed in columns are: (a) online scheduling, (b) task deadlines, (c) geographic approach, (d) real implementation.

Similarly to the works presented in this chapter, [117] uses Sarsa and Q-Table for implementing the scheduling in a generic heterogeneous distributed system by using reinforcement learning, however, the authors do not consider task deadlines and we also use the average reward approach that best fits a continuous learning task. A similar geographic approach, by using a similar traffic dataset is followed by [119], instead [103] focuses on smart cities but does not use a real traffic dataset, and [128] which

uses a real node topology.

In [106], the authors present a Deep Reinforcement Learning approach, based on Q-Learning, in a MEC environment, for selecting the best Edge server for offloading in order to minimise the energy consumption (also studied in [115], [124]) while at the same time maximising the number of tasks that meet the deadline. In this work, two DNNs are used: one is kept fixed during an episode, while the other is updated and at the end of the episode they are swapped. Differently from the works presented in this chapter, that approach is not fully decentralised and requires a central entity which collects information about the state of each node and takes the decision (as also studied in [123]). However, the followed approach is very common and also used in [108], [111], [118] and [127]. Other works instead relies for example on Meta Reinforcement Learning (MRL) [120], blockchain [122], or even a genetic algorithm [102].

Pandit *et al.* in [107], propose a scheduling scheme based again on two DNNs but they are used for two different decisions, the first one is in charge of deciding if the task should be offloaded to the cloud, but if not, the second decision level chooses the best suitable Fog node to which schedule the task. In the presented works instead, we do not rely on a neural network, since we keep the state as small as possible since we noticed that when dealing with deadlines the inference time is critical.

The approach followed by [108], but in the context of MEC cells (as in [121]), is the one of defining the reward as the weighted sum of energy consumption, delay and cache fetching cost, then the Deep Reinforcement Learning approach is followed by using a Deep Deterministic Policy Gradient (DDPG) method which solves the problem of the state discretisation in the standard Q-Learning approach. In the presented works instead, the state is discrete and we rely on basic RL approach for reducing the inference time.

Main *et al.* [109] focus on real-time task assignment but consider the evolution strategies approach instead of the backpropagation for updating the weight of the DNN. The task model is very similar to the one that we study but the work uses a DNN and does not provide real implementation results of the proposed algorithm.

In [110], an approach based on a recurrent neural network (RNN) is proposed for online task scheduling, however, the authors suppose the existence of a central orchestrator which is able to maintain the state and take the decision. Although this can work in simulations, in real scenarios, with task deadlines, this could introduce a non-negligible scheduling latency which we avoid by making each node a learner agent.

Tuli *et al.*[116] uses the Asynchronous Advantage Actor Critic (A3C) algorithm in a Edge-Cloud environment for scheduling in a supposed high number of host nodes, however, the task deadlines are not considered and the authors did not provide a real-world prototype implementation, rather they use simulators. Instead, [126] uses again the Actor-Critic paradigm for a size adaptive caching scheme.

In a broader sense of schedule, other works are instead focused on resource allocation [130], [131], [118], [124], [125] but the task model does not fit the one that is studied in this section. Other solutions targets vehicular networks [111], [114] or crowdsensing [112].

The Reinforcement Learning algorithm that is used in the works presented in this chapter is specifically tailored for a continuous learning approach and not an episodic one, indeed the scheduling that we follow is online and a per-job decision task must be taken. The approach is called Differential

Semi-Gradient Sarsa and it is presented in [132]. However, many works in literature rely on Deep Q Learning for solving the task scheduling problem in Edge or Fog computing, especially because of the high dimensionality of the state space. For example, in [133], the authors propose a deep reinforcement approach for resource allocation in a MEC system, differently from this study, the allocation scheme is based on time slices and the objective is to minimise the execution time. In [134], the focus is instead on base stations that must be selected by the client in an ultra-high-dense network, the authors show through numerical experiments the beneficial effect of their solution. The authors of [135] specifically study task scheduling in Edge computing and use reinforcement learning for deciding the order of the execution of the tasks and in which machine they have to be executed, the approach uses Deep Reinforcement Learning but the scheduling is not done online. In [136] a solution for caching at the Edge is proposed, the authors use reinforcement learning for finding an optimal stochastic allocation policy, and the approach is tested with simulations. In [137], the scenario studied is the one in which mobiles can appear randomly in a cell, the authors take as reference the uplink transmission, the device selection and the power allocation. The proposed approach uses reinforcement learning and stochastic gradient descent for the online improvement of the system. More similar to this study is [138] which focuses on online scheduling and using time differential learning, but the tasks do not have to meet a deadline. Then [139] introduces a specific study on task placement in the Edge-to-cloud computing continuum.

Other works are still focused on scheduling but targeting the energy consumption [113], [140], vehicular networks [141], [142], network resources allocation [143] or security [144].

4.2 Online scheduling in a geographic setting

Fog Computing [2] is a well-known computing paradigm that is, not only, but usually chosen when the computation must be distributed in a geographic domain. This "must be" is generally given by the fact that the application has to be deployed as near as possible to users who have to use it. Indeed, in such situations, a cloud approach cannot be feasible, especially when the tasks that the application should carry out are strict in their deadlines, for instance, when we refer to a shared Virtual Reality (VR) [145] or Augmented Reality (AR) experiences. Distributing the load involves the setup of different computing nodes, called Fog Nodes, which, for example, can be spread across a city. Issues that arise within this setup essentially regard the fact that these nodes should be able to interact in some way in order to reach a common goal: each task requested to be executed by a user, to any Fog node, has to be able to meet its deadline. This kind of interaction is needed not only because we want to create a sort of an ecosystem in which the application can live, so that it can be reachable in any Fog node, but also because in such dynamic environments, for instance, some nodes can be overwhelmed by unpredictable traffic load, some instead can go down or others can become idle since no user is requesting them to execute tasks. Indeed, if we want them to be able to cooperate, a smart scheduling algorithm should be able to change its scheduling policy according to the current situation, by always having in mind the same previous goal we suppose that each Fog node receives requests to execute tasks from the clients and we need to make a scheduling decision on a per-task-request basis. This requires that, for making an optimal scheduling decision, we need to know the state of the other nodes. However, since this environment is completely distributed and decentralised, the only way for knowing the state (i.e. its current load level) of another node is to explicitly ask for it.

A well-known approach, that is proven to perform efficiently in this setting, is the power-of-random choice paradigm [19] (presented in Chaper 2), where every scheduling decision, that is done on a per-task basis, is preceded by a random probing to another node, with the purpose to retrieve its current load. Once this information is retrieved, the task is scheduled internally or forwarded to the random-probed node. Executing a probing for each request is not always the best behaviour, indeed, adding a control threshold to decide when to trigger a new probing request [13] is shown to be an effective way to increase the performance over the standard approach. However, even this improved algorithm has limitations, for example, the scheduling policy (i.e. when to trigger the probing) is a fixed step function (i) of the current load, e.g. the probing is performed only if the current workload exceeds the threshold, moreover it is also fixed over time (ii) and it cannot react to load variation on the nodes, and finally, it doesn't take task heterogeneity into account (iii). The purpose of this study is to overcome these limitations by designing a dynamic scheduling policy based on the Reinforcement Learning (RL) paradigm, where the probing decision is a function defined over the whole set of load states and the task performance requirements (expressed as a deadline). As a further step from the power-of-random of choice paradigm, we also study a scheduling policy that directly forward the tasks to a specific node with no probing at all. In the proposed approach, we encode a learner *agent* as a Fog Node. This agent chooses the correct *action*, that is a per-task scheduling decision (e.g. forward the task to another node, or execute the task locally), from a set of predefined actions by looking at its current *state*. Then, after the task is finished we assign to the chosen action a *reward* signal that will

drive the learning process. This reward will be positive only if the scheduling decision that has been taken has satisfied a simple constraint: the task has completed in the defined deadline.

We can summarise the main contributions of this study as follows.

- Design of a decentralised RL-based algorithm to be implemented in every Fog node that is able to choose the best scheduling decision according to the current situation, which is a step forward the power-of-random choice approach, that allows for more complex policies than simple and fixed threshold-based decisions, to be dynamic over time reacting to the current load situation of nodes and to deal with different kinds of time-constrained (i.e. with deadline requirements) tasks.
- Study of a Geographic setting which involves six Fog nodes deployed in the city of New York and in which the algorithm can be deployed.
- Simulation Results on a delay-based simulator prove the efficiency of the algorithm in a previously defined geographic environment compared to the classic power-of-choice strategy.
- Results from a pseudo-real deployment with the proposed prototype framework "P2PFaaS" in a rack of 12 Raspberry Pis which shows that the algorithm achieves the same performance indicators even outside the simulation.

4.2.1 System Model and Problem Definition

In order to reach the goal of adaptability and optimality, we frame the problem as a Markov Decision Process (MDP), which is solved using the Reinforcement Learning (RL) techniquewe use a model-free approach with the advantage that it doesn't require the knowledge of the details of the underlying mathematical model, such as the state transition probabilities. Rather, it is enough to observe and interact with the environment. In addition, once tested using simulations, the algorithm can be ported on a real deploy. To proceed the discussion, it is needed preliminarily to identify the two main entities of RL: the environment and the agent.

4.2.1.1 Environment

The environment is composed of a set of N communicating and nearby Fog nodes $\mathcal{F} = \{F_1, F_2, \ldots, F_N\}$ with the same computing power. Every Fog node serves an area from where users can require the execution of a task and we define the total rate of the arriving requests to a node i to be λ_i req/s. One example of application scenario for such tasks is Virtual Reality (VR), where a user needs to execute compute-intensive tasks like recognising and tracking objects or activities. Tasks have a deadline T associated with them, which represents the absolute time before which they must be processed, e.g. 10ms in a VR scenario [145]. They also have a physical size b, that is represented by the number of bytes of the payload that it is needed to transmit for executing the task (e.g. an image, a set of video frames).

A Fog node F_i has a performance profile defined by its queue capacity K_i, representing the maximum number of pending tasks waiting to be processed, and the rate of execution of the tasks that is μ task/s that is supposed to be equal for each node. A Fog node is capable of executing one task at a time but since it has a queue of K_i this is exactly equal to say that it is capable of executing K_i tasks at a time

	Learning Environment
AG	Learner agent
ENV	Environment in which the agent acts
$\mathcal{F}$	Set of Fog nodes
$\mathcal{A}$	Set of actions which comprehends *execute-locally* and *probe-and-forward* actions
$\mathcal{A}'_i$	Set of actions of node i which comprehends $\mathcal{A}$ plus the direct forwarding to a specific neighbour node
R	Reward for task j
ϵ	Parameter of the ϵ-greedy strategy for action selection
ι	In-deadline rate
Z	Window size of completed tasks that triggers the training process
	Fog Nodes
λ_i	Rate of arrival to node i (task/s)
μ	Service rate of node i (task executed per second)
ρ_i	Load to node i (λ_i/μ)
$\hat{\mu}_X$	Arrival rate for a node i starting from which request are dropped for payload Image X (A or B)
γ	Percentage over d_t that is added to the deadline of a task
	Queues
Q_e	Execution Queue
Q_p	Probing Queue
Q_t	Transmission Queue
	Times and Delays
d_e	Total time spent in Q_e for a given task
d_p	Total time spent in Q_p for a given task
d_t	Total time spent in Q_t for a given task
W	Total completion time of task (as seen by clients)
T	Task deadline
	Probabilities
P_B	Probability of a task to be rejected by the node
P_f	Probability of a task to be forwarded to another node

Table 4.2: List of symbols used

in time-sharing with no queue, that is a mechanism closer to reality.

The total load to a specific node i is:

$$\rho_i = \frac{\lambda_i}{\mu} \tag{4.1}$$

Nodes can communicate with each other and each transmission requires a time interval d_t, determined by the data rate, r of the link connecting the two communicating nodes:

$$d_t = \frac{b}{r} \tag{4.2}$$

Moreover, a node A can probe another node B, meaning that A can ask B its current queue length in order to take scheduling decision. The queue length of a node it is usually referred as its current "state".

4.2.1.2 Agent

Upon each task request arrival to a Fog node from a client, a scheduling decision has to be taken. This is an online decision process that is carried out by an agent, running at each Fog node. Each agent has associated the same set of actions $\mathcal{A}$ that can perform. The action $a \in \mathcal{A}$ to take is determined by a function $\pi(s)$, called *policy*, of the current observed state $s \in \mathcal{S}$.

$$\pi : S \to \mathcal{A} \tag{4.3}$$

4.2.1.3 Observed states

The agent running on node F_i is able to observe its current state, that is referring to the number of tasks in its execution queue Q_e at time t, i.e. $k_i^t \leq K_i$. Moreover, we consider the case in which tasks of multiple types can arrive in the node, for this reason the final observed state by the agent is the aggregation of the number of tasks in the queue for each type and the type of the newly arrived task. This means that the decision about the action to choose is only done by observing the queue length of the current node. As stated in the introduction of the section, since the environment is fully distributed we cannot have an updated information about the state of the other nodes, we could suppose that each node periodically polls the other nodes and updates its internal cache of the states but this is left as future work, here we consider no knowledge of other nodes involved in the learning process. However, the agent should be anyway able to perform the correct decision, this because the load is inferred from the reward of a given action in a particular state.

Finally, the state that is derived as described and used for the learning process is not taken as is, indeed the tiling technique [132] is used for representing it as a vector $v \in \mathbb{N}^8$.

4.2.1.4 Actions

We studied separately two sets of actions that can be performed by the agent. In the first case (i), the agent selects an action from the set $\mathcal{A} = \{0, 1\}$, where 0 means to execute the task locally, while 1 to probe another node at random and offloading the execution of the task to that node only if its queue length is lesser than the one of the current node (we call this strategy "probe-and-forward") otherwise the task is executed locally. In the second case (ii) instead, the agent of node i selects an action from $\mathcal{A}'_i = \mathcal{A} \cup \mathcal{F}_i$, where $\mathcal{F}_i$ contains additional direct forwarding actions that depends on the Fog node i. The action $f_j \in \mathcal{F}_i$ means to directly forwarding a task to the neighbour j without probing, obviously with $j \neq i$.

It is worth to remind that, in any case, when the task is scheduled to be executed locally, despite being forwarded from another node, it can be rejected if there is no room for being executed, i.e. the queue is full (at a certain time t, $k_i^t = K_i$).

4.2.1.5 Reward

The immediate reward given to a specific action is given by the fact that the task has been executed within the deadline or not. Therefore, the reward is a function of the state and the action performed

given the state. For a given task j, the reward assigned to action a performed when the state was s is, given the total completion time W:

$$R_j(s,a) = \begin{cases} 1 & \text{if } W \leq T \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Obviously, the agent cannot know the reward until a task has been completed its execution path and in the meanwhile other tasks may arrive and need to be scheduled. Moreover, even if two tasks are scheduled sequentially, the second can terminate before the first, altering the causal order for the decision path. To overcome this problem, the learning step is put on hold until a number equal to Z (the window size) of tasks have been completed. This is discussed in Section 4.2.2.

For measuring the performances of the algorithm we use the reward rate, also called the in-deadline rate ι that is the average reward per second.

4.2.1.6 Delay model

Dealing with deadlines requires a fine-grained model of the delays. In the proposed study, the environment is simulated as a network of N nodes in which every node is composed of three different internal queues:

- the execution queue (Q_e) represents the queue of tasks that have been scheduled to be run in the current node; a node can execute one task at the time and the total time that a task spends on this queue is d_e, but the actual execution time of a single task follows a Gaussian distribution;
- the transmission queue (Q_t) represents the queue of tasks that are in the transmission phase; the transmission can occur: (i) from client to node, (ii) from node to node, (iii) from node to client. The total time that a task spends on this queue is d_t and it follows a Gaussian distribution with μ equal to Equation 4.2;
- the probing queue (Q_p) represents the queue of tasks for which there is a probing request to run; the total time that a task spends on this queue is d_p;

The flow according to which a task transits among the queues is represented in Figure 4.2.1. Suppose that a client sends a task request to node A that can decide to forward the task to a node B. The task enters in the transmission queue Q_t:

1. when the client transmits it to a node A, in this case, afterward, the scheduling decision is taken that can include a probing and the task will be added to the probing queue Q_p;
2. when it is transmitted from a node A to a node B;
3. when it returns from a node B to a node A and afterward the task will be again added to the transmission queue to simulate the retransmission to the client;
4. when it returns to the client from node A.

When the task is scheduled to be run in the current node, the task is added to the execution queue Q_e.

The total time of a task to be executed, from the client perspective, is the summation of all the

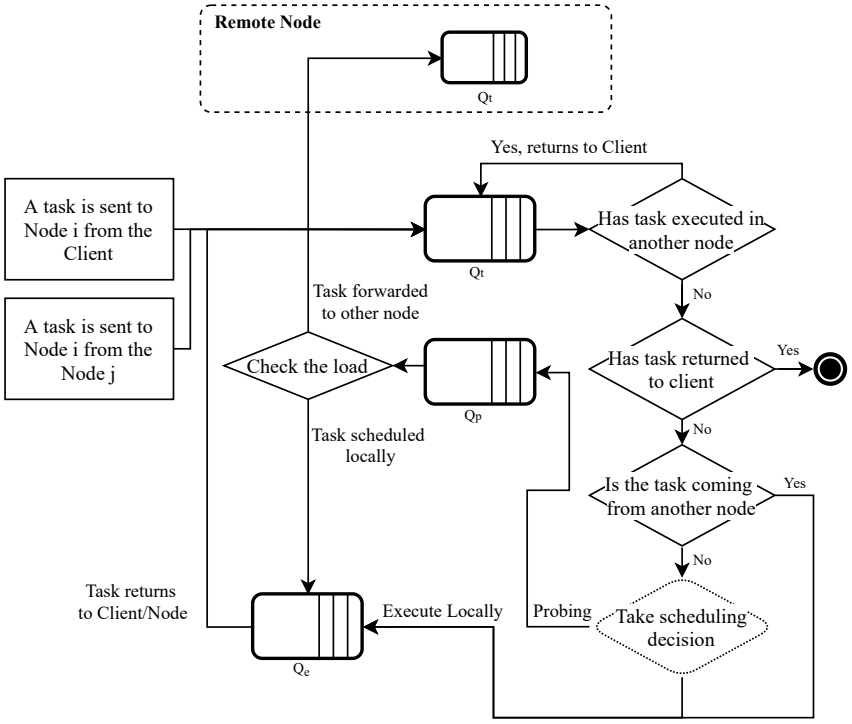


Figure 4.2.1: The logic of the delay model.

time spent in all the queues during its entire execution path, it is referred as W and it is measured in seconds.

4.2.1.7 Geographic Scenario

A peculiar characteristic of Fog computing is that nodes can be positioned in a geographic scenario [2]. In order to evaluate the adaptivity of the proposed solution in a real environment, we used open data of New York city[1] to estimate the average daily traffic in specific points of the city. The data is referred to taxi trips of year 2013, and for every trip we considered the start coordinates, the end coordinates, and the total trip time. Then we placed six Fog nodes, as in Figure 4.2.2, considering 1 km of radius for the service to be available, this is inline with the capability of a RRU to which has been attached a computing node. We estimated the taxi traffic by dividing the day in 15-minute time slots (for a total of 96 time slots) and counting the number of taxis within the area of the nearest Fog nodes during their tripwe used the first three months of data by averaging the daily traffic within each time slot for every Fog node.

By normalising in the range between 0 and 0.9 the final traffic distribution is represented in Figure 4.2.3. This range is given for simplicity, and derives from a reasonable assumption that Fog nodes never saturate, leaving the opposite case as future work. Moreover, the final curves have been smoothed with Savitzky–Golay filter, using an order 4 polynomial and a window of size 17.

The final result of this study has been used in a simulated environment by setting the load to a

[1] https://web.archive.org/web/20210424121526/https://chriswhong.com/open-data/foil_nyc_taxi/

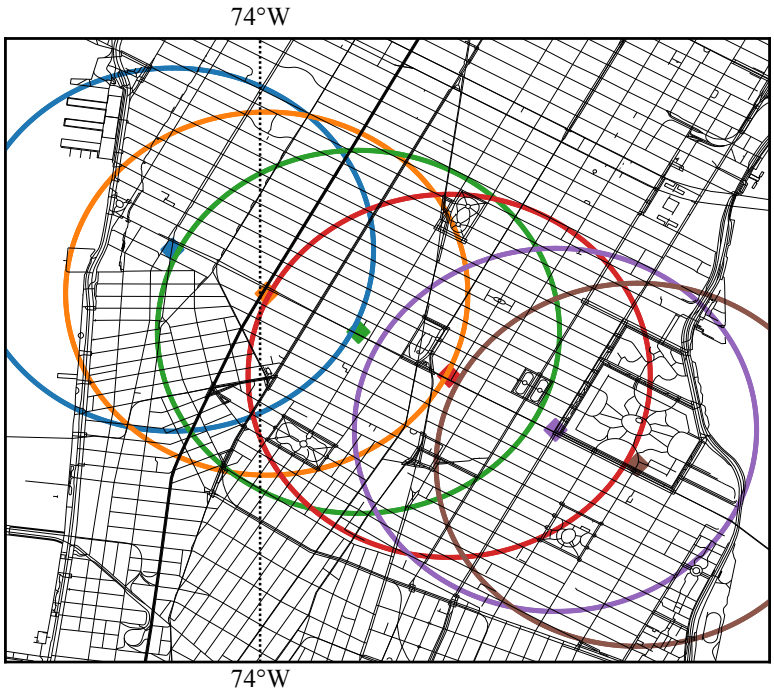


Figure 4.2.2: Fog nodes position (diamond symbols) in New York city used in the experiments, from left to right Node 0 to Node 5. The radius of the circle for each node is 1 km.

specific Fog node i (ρ_i) to be equal to the value of the respective curve in that precise moment of the simulation.

4.2.2 Online scheduling decisions with RL

The final objective of the agent is the one of learning a scheduling policy π that maximizes the long-term reward. Since each decision must be taken online, we cannot envision episodes but we treat the problem as a continuing learning task.

In a continuing learning task it is not useful to discount future rewards but it is better considering the current average reward for taking the right direction. Given a state $s \in \mathcal{S}$, we perform the action $a \in \mathcal{A}$, we obtain the immediate reward r the next state is $s' \in \mathcal{S}$ then the optimal policy (that is the policy which maximizes the long-term reward) will result in the optimal q_* function defined as [132]:

$$q_*(s,a) = \sum_{r,s'} p(s',r|s,a)\left[r - \max_{\pi} r(\pi) + \max_{a'} q_*(s',a')\right] \tag{4.5}$$

Where $r(\pi)$ is a function which returns the average reward of the policy π. At certain time t and given a weight's vector $\vec{w}$ the differential form of the error, following the Sarsa approach for learning the policy, can be expressed as [132]:

$$\delta_t = R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \vec{w}_t) - \hat{q}(S_t, A_t, \vec{w}_t) \tag{4.6}$$

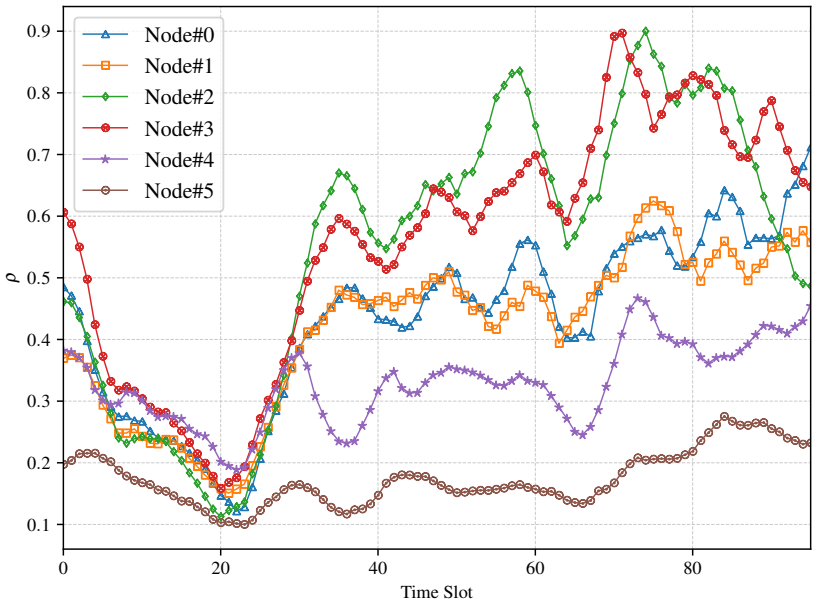


Figure 4.2.3: The average distribution of the traffic during the day for the picked Fog nodes.

When used in practice, the $q(s, a, \vec{w})$ is approximated by using the linear combination of the coordinates given by the tiling technique, as described in Section 4.2.1.2. The final used strategy for learning the policy is called Differential Semi-Gradient Sarsa (Algorithm 10).

In the experimental setting (subsection 4.2.4), we used a variant of this scheme since the state space is so small that a table for storing the Q values can be used, therefore we used the QTable as the final function approximation mechanism, and in practice the cells of the table has been updated according to the following Equation 4.7.

$$Q(S_t, A_t) \leftarrow Q(S, a) + \alpha\Delta_t \tag{4.7}$$

The Δ at time t can be written by following the Sarsa approach and still considering the average reward $\bar{R}_{t+1}$, as in the following Equation 4.7.

$$\Delta_t = [R_{t+1} - \bar{R}_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4.8}$$

As in the linear approximation solution, we also apply a discount factor to the average reward, which becomes as in Equation 4.9.

$$\bar{R}_{t+1} = \bar{R}_t + \beta\Delta_t \tag{4.9}$$

However, in the analysed setting, we do not have a real notion of immediate reward because we it can be known only after a task has been executed or rejected, for this reason we set a window size of Z tasks and right after the execution of every task we check if the window is reached and every task in the window has been executed or rejected. This is explained in the following algorithms descriptions.

The Algorithm 9 is run whenever a new task to be executed arrives. First of all, we append the task

to the array of pending tasks ("TasksArray") then we compute the state (as described in Section 4.2.1) and we retrieve the best action to perform given the current $q(s, a, \vec{w})$. If the action is 0, the task is immediately executed locally, if is 1 the node asks the state to a random node and the task is forwarded only if the random node's state is better that the current one. These two actions are of $\mathcal{A}_1$, in any other case the task is directly forwarded to the chosen node ($\mathcal{A}_2$), unless the picked node is the current one, in that case the function *forwardTo()* only executes the task locally.

Algorithm 9 Scheduling Decision

Require: Node, Task, TasksArray, $\vec{w}$, $\mathcal{A}$
  TasksArray.append(Task)
  $s \leftarrow$ aggregate(Node.getLoad(), Task.getType())
  $a \leftarrow \max_{a \in \mathcal{A}} q(s, a, \vec{w})$ with prob. $1 - \epsilon$ otherwise random($\mathcal{A}$)
  Task.saveStateAction(s, a)
  if a == 0 (*Execute Locally*) **then**
    Node.execute(Task)
  else if a == 1 (*Probe-and-Forward*) **then**
    RandomNode $\leftarrow$ pickRandom(Node.getneighbours())
    if RandomNode.getLoad() $<$ Node.getLoad() **then**
      forwardTo(Randomneighbour, Task)
    else
      Node.execute(Tasks)
    end if
  else
    Node $\leftarrow$ pickNode(a)
    forwardTo(Node, Tasks)
  end if

Every time that a task completed its execution (that means that result payload of the task is returned to the client), whether it is local or remote Algorithm 10 is executed. First of all, we record the task reward and then we start to iterate over the array of pending tasks ("TasksArray") for checking if the first Z tasks of the array are finished, if this is not the case the function returns, otherwise we go on by retrieving the information about the first Z tasks by popping them from the array. This information is used to train the weights vector $\vec{w}$ using the semi-gradient differential Sarsa algorithm.

4.2.3 Simulation Results

The results that we are going to present in this section follows the assumption that there are 6 Fog Nodes ($N = 6$) and for every Fog node i the maximum queue length is $K_i = 5$. Moreover, the arrival distribution of the tasks is a Poisson with mean $\mu_i$we suppose that nodes are connected with a link of 1Gbps, the payload of each task is 100kb and the probing delay is 5ms. Q_t and Q_p are unlimited in size, for each node. The rationale behind this number of nodes is that to avoid a high offloading delay among nodes, the cooperating nodes are physically close one with the other and hence they amount to a few units. In other words, cooperation occurs only among nearby nodes.

In the following subsections, we evaluate the proposed algorithm in the scope of three settings. The first one (Section 4.2.3.1) is when the loads (i.e. ρ_i to every node i) are fixed in time to each node. This setting is basically used as a validity check of the RL approach, we test every possible policy with

Algorithm 10 Learning with Differential Semi-Gradient Sarsa

Require: Task, TasksArray, Z, $\vec{w}$, $\bar{R}$, α, β
  Task.setReward()
  i $\leftarrow$ 0
  for all j in TasksArray **do**
    if !j.isDone() **then**
      return
    end if
    if i == Z **then**
      break
    end if
    i $\leftarrow$ i + 1
  end for
  i $\leftarrow$ 0
  $j_0 \leftarrow$ TasksArray.pop(0); $s \leftarrow j_0$.getStateSnapshot()
  $a \leftarrow j_0$.getAction(); $r \leftarrow j_0$.getReward()
  for i = 0; i $<$ Z; i++ **do**
    $j \leftarrow$ TasksArray.pop(0)
    $s' \leftarrow j$.getStateSnapshot()
    $a' \leftarrow j$.getAction()
    $\delta \leftarrow r$ - $\bar{R}$ + $q(s', a', \vec{w})$ - $q(s, a, \vec{w})$
    $\bar{R} \leftarrow \bar{R}$ + $\beta\delta$; $\vec{w} \leftarrow \vec{w}$ + $\alpha\delta\nabla q(s, a, \vec{w})$
    $s \leftarrow s'$; $a \leftarrow a'$; $r \leftarrow j$.getReward()
  end for

the power-of-choice strategy and we show that the agent is able to learn the best one by using the set of action $\mathcal{A}$. In the second setting (Section 4.2.3.2), instead, we vary the loads to the nodes but again making them fixed over time. Here we show how the RL approach can outperform the classic power-of-choice strategy founding a solution that maximises ι in every node when using the set of actions $\mathcal{A}'$. In the last setting (Section 4.2.3.3), we apply the traffic study results (Section 4.2.1.7) and we make nodes to follow the load traces in Figure 4.2.3. In this setting we again show how the RL approach can outperform the standard one when also when the traffic is variable over time.

In all of these experiments, the proposed RL algorithm is labelled as "Sarsa" while the power-of-choice one "Pwr2", that is specifically referring to the power-of *two* choices since only one node is probed random and therefore the scheduling decision is between the current node and the probed one, that are two choices.

4.2.3.1 Homogeneous Loads

The first approach that it has been followed is the one of supposing the load is constant to every Fog node. The learning agent AG, that we remind is present in every Fog node, can choose among the two actions in $\mathcal{A}$. Then, if the agent chooses to probe a random node, only if the queue length of the remote node is lesser than the current one, the job will be forwarded to it (we call this approach, "probe-and-forward"), otherwise the task will be executed locally. In this context, the policy π can be easily binary encoded $\pi = (a_0, a_1, \dots a_K)$, where $a_i \in \mathcal{A} = \{0, 1\}$ is the action executed when the state of the agent is i. For example, for $K = 5$ the policy 000111 means that the task is executed locally when the state is 0, 1, or 2 (action 0), and forwarded otherwise (action 1).

For having a term of comparison with the learning algorithm, we tested every possible scheduling policy statically, without the learning framework but leaving all the infrastructure valid. This means that the environment, the set of actions and the reward function are the same, we only consider the policy as staticwe set the mean task duration for every Fog node i as $1/\mu_i = 0.023s$, the load as $\rho_i = 0.6$ and the deadline $T = 0.042s$. The bar plot in Figure 4.2.4 compares the reward of each possible policy, expressed as in the form of rate of in-deadline tasks ι. What we can observe is that the optimal policy is 001111 ($\iota = 0.8164$) that is the one of performing the random probing if the load is greater or equal to 2. This is a sort of cooperation threshold that perfectly matches the results of [13], although it considered no deadlines. However, this policy is very similar, in terms of performances, also to 001110 ($\iota = 0.8147$), 001100 ($\iota = 0.8145$) and 001101 ($\iota = 0.8114$) which respectively disable the cooperation when the load is equal to 5, 4 and 5, and only 4. This is justified by the fact that the rewards gained in such situations that are referring to load values greater than 4 are negligible with respect the other states. In the chart, we can also observe that the forwarded rate (in green), that is referred to the percentage of jobs that are scheduled to run in a remote node, is strictly depending on the cooperation trend of the policy and that the rejected rate (in orange) is greater than zero for policies that are less willing to cooperate, this confirms the validity of the experiment. Surprisingly, the policy that performs the worst is not the one that never cooperates (i.e. 000000) but 100001 ($\iota = 0.4990$), this behaviour emerges from strictness of the task deadline, indeed cooperating when the state is zero leads to more tasks that cannot be executed in deadline.

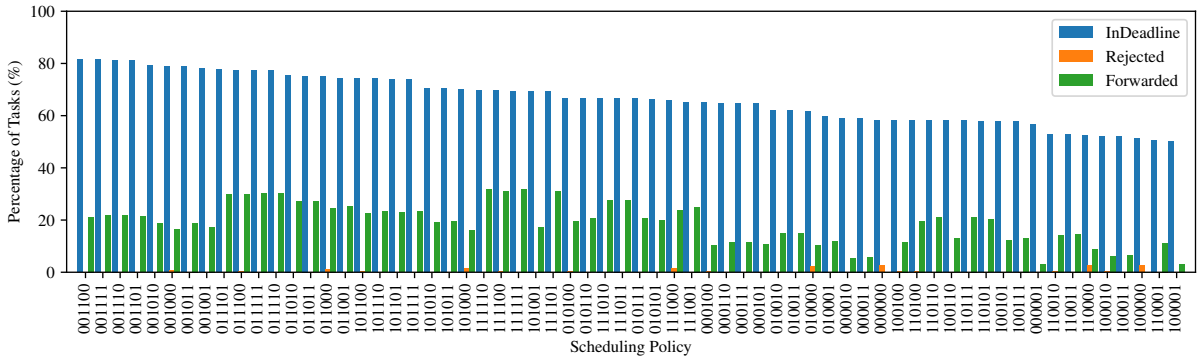


Figure 4.2.4: Percentage of In-Deadline, Rejected and Forwarded tasks of all the possible policies with 6 nodes, a policy is encoded in binary where 1 means probing and 0 means executing locally. In this experiment $\rho = 0.6$, deadline is = 0.043s and job duration is 0.020s.

In the same set up, we used the learning approach described in Section 4.2.2. At the end of the simulation, the policy learned by the learner is the one in Figure 4.2.5, i.e. 001110. This is not the optimal policy, but this is again justified by the fact that the contribution of state 5 is so small that is not appreciable by the learner. For this reason, we defined a decision confidence by just taking into account how many times the decision has been made in each possible state. Given that s_j is the number of tasks whose scheduling decision has been made when the state was j, and J the total number of jobs, the value in the cell i, j of the heat map is given by the following choice confidence function of

the action in i when in state j:

$$C(i,j) = \begin{cases} \frac{s_j}{J} & \text{if } i = 0 \text{ (Non-Probe)} \\ -(1 - \frac{s_j}{J}) = \frac{s_j}{J} - 1 & \text{if } i = 1 \text{ (Probe)} \end{cases} \tag{4.10}$$

This formulation allows us to understand the confidence in the choice of the agent. In particular, from Figure 4.2.5 we can appreciate how the choices in the lower states are more confident since the states have been more frequent.

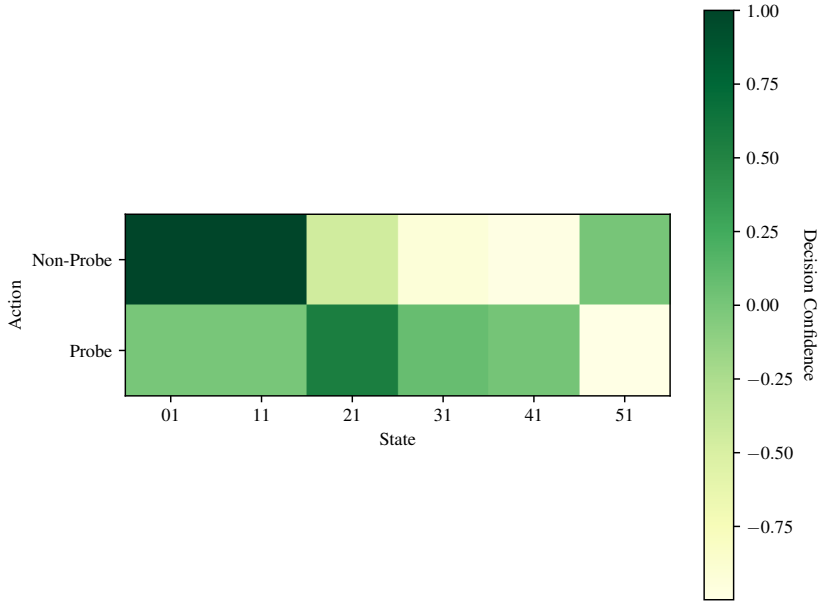


Figure 4.2.5: The policy learned by the agent in the same setting of Figure 4.2.4

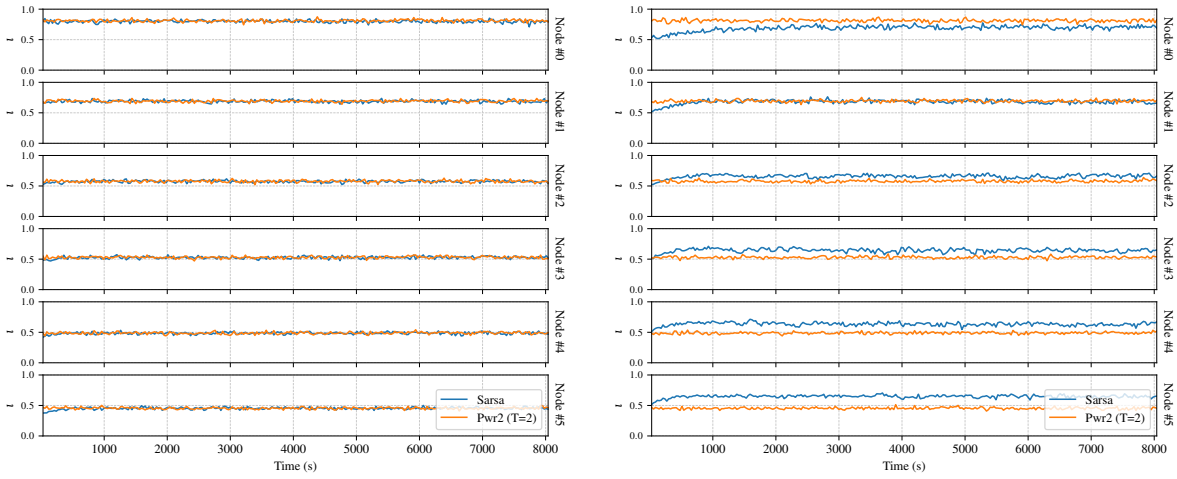
4.2.3.2 Heterogeneous Loads

During the previous experiments, we assumed only one kind of task, which has deadline 43ms and its average duration is 20ms. By using the same framework we switched to two kinds of tasks, one (type 0) that is expected to run at 60fps and therefore we supposed that it has deadline 16ms and mean duration 8ms ($\sigma = 0.4$ms) and one (type 1) that is expected to run at 30fps and therefore it has deadline 40ms and mean duration 20ms ($\sigma = 0.4$ms).

Figure 4.2.6a shows the behaviour of the in-deadline rate ι, and therefore of the reward, when every node has a different load (from node 0 to 5 we set: 0.2, 0.4, 0.6, 0.7, 0.8, 0.9) but it is stationary over time. The chart compares the proposed reinforcement learning approach and the power-of-choice with the threshold set to 2 (i.e. policy 00111). What emerges again is that the learning approach exactly mimics the power-of-choice under the point of view of the performances (ι). Since here we use the set of actions $\mathcal{A}$ we also deduce that the policy is likely to be the same of the power-of-choice with threshold set to 2.

At this point, we wonder how we can achieve better performances, over than using the same reduced

set of policies. The only way of achieving better results is obviously allowing to increase the action space of the agent. For this reason, we introduced the set of actions $\mathcal{A}'_i$. Figure 4.2.6b shows the behaviour of the in-deadline rate when the loads are not balanced, as in the previous experiment, but stationary over time. The only difference here is that the agent can choose to directly forward tasks to a given node. After the first 1000s, which is the period in which the ϵ is greater than 0.1 (the lower limit), we can observe how ι is fixed over time and it is equal to every node. This means that even the nodes that with the policy 001111 could have a better reward they are not selfish but voluntarily decrease its reward for making the others to achieve the best rewardwe do believe that this behaviour is inherent to the distributed usage of the reinforcement learning approach, since every node is able to understand the situation only from the reward, that in the end declares the goodness of the chosen action. This means that allowing the nodes to forward directly (set of actions $\mathcal{A}'_i$) we make them understand which is the best node to forward the jobs in a given time, and this enables the fact that acting like selfish will deteriorate its reward. Anyway, this aspect will be further studied as a future work.



(a) Fixed load and same to every node

(b) Fixed load but heterogeneous

Figure 4.2.6: Comparison between Sarsa and Pwr2 regarding the behaviour of the in-deadline rate ι for every node

4.2.3.3 Geographic Scenario

As described in Section 4.2.1 the open data for New York city has been used for generating the traffic to six Fog nodes positioned as in Figure 4.2.2. Starting from this setting, by using the same assumptions of the previous experiment, we only enable the load to change according to the derived distribution for the open data (Figure 4.2.3). In Figure 4.2.7 we can observe as the behaviour that was shown in the fixed load case is again confirmed even if the load follows a variable distribution. Every node reaches the same level of the reward, even if by following the policy 001111 (power-of-choice with threshold 2) could make a node able to reach a better reward, and this behaviour is an invariant with respect to the traffic variability.

As a final note, we remark that the policy that is chosen by this reinforcement learning approach,

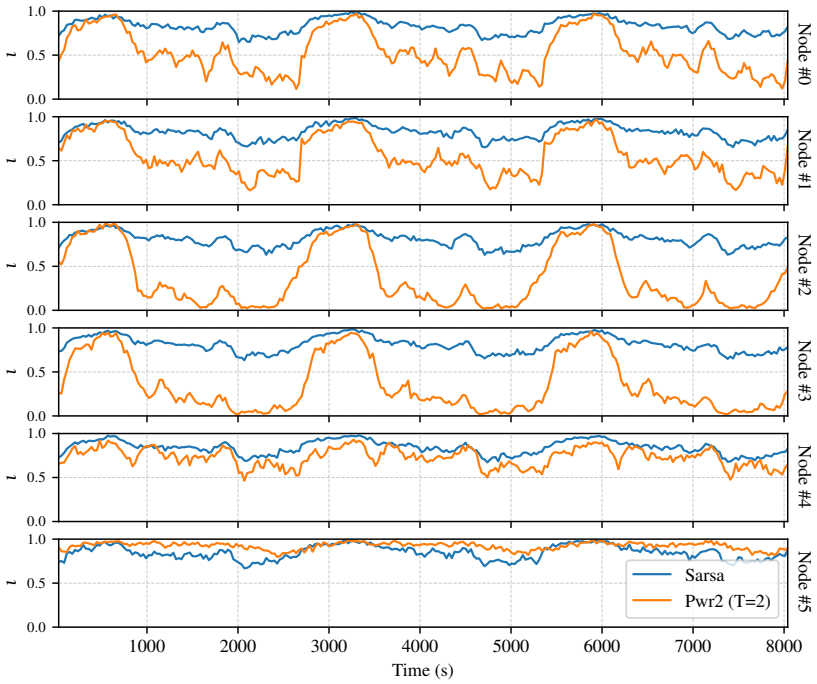


Figure 4.2.7: Comparison between Sarsa and Pwr2: behaviour of the in-deadline rate ι for every node when the load is variable according to the geographic scenario, Figure 4.2.3

both in the fixed and in the geographic scenario, is not trivial and easy to be manually determined but it dynamically change over time and unfortunately it is not feasible to be represented in a figure, for this reason it has been skipped.

4.2.4 Experimental Setting

In this section, we introduce and describe the implementation of the proposed RL scheduling policy in a pseudo-real setting by using the proposed prototype framework. The environment is "pseudo-real" because the hardware is real but the nodes are not placed in a real smart city and the traffic is still simulated, however, we believe that this has not a great impact on the trustworthiness of the final results achieved.

4.2.4.1 Practical Setting

The P2PFaaS framework has been installed in a cluster of twelve Raspberry Pis 4. In particular, the nodes from #0 to #5 have 4GB of RAM and the nodes from #6 to #11 have 8GB of RAM. The installation of the framework to all of the nodes and its continuous updating during the development has been made possible thanks to the open-source OpenBalena[2] IaaS framework which allows the simultaneous deployment and management of Docker containers in clusters of devices of the same type. The Figure 4.2.8, shows the diagram of the final experiment setting. The Raspberry Pi nodes are attached

[2]https://github.com/balena-io/open-balena

to a 1 Gigabit network switch, then there is a router and a server which has two purposes: the first is the one of hosting the OpenBalena framework and the second is to generate the traffic to the nodes and collect the data. The traffic generator program is still written in Go, and it generates one thread of traffic to each node. We preferred it over Python since Python threads are not true parallel threads, due to the Global Interpreter Lock (GIL).

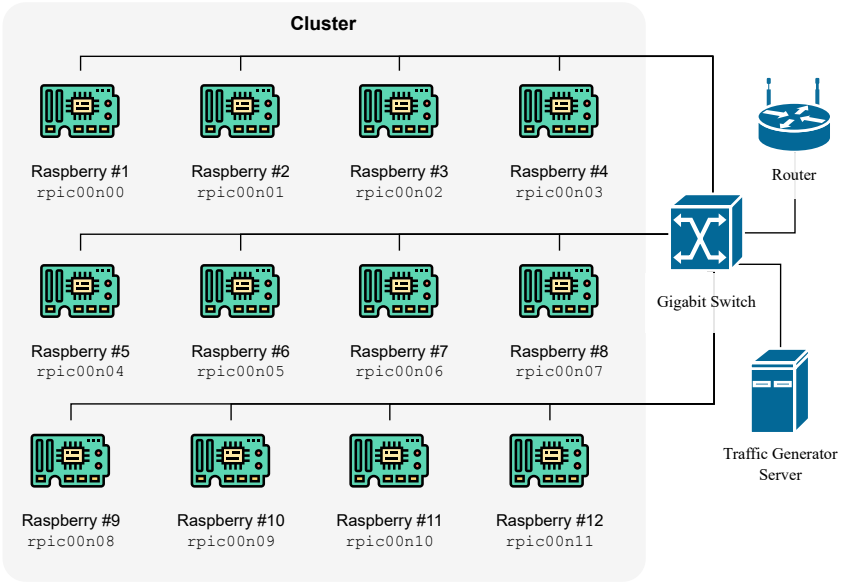


Figure 4.2.8: The diagram of the practical setting in which we run the experiments. We can observe that the set of nodes are attached to a network switch as well as a traffic generator server.

4.2.4.2 Single Node Behaviour

The FaaS function used in all the following tests is a face detection function[3] written in Go and ported from the OpenFaaS project. The function takes as input payload an image and returns the same image with the faces highlighted in JPG format with compression of 80%. The image payload has a great impact on the duration of the function, indeed, as shown in Table 4.3, two different images have been used. In the table, we can observe both the effective execution time d_e and the total one d_t and how the difference among them is in the order of $\approx$ 8ms. That specific delay comprehends the transmission time of the payload (both when invoking the function and when receiving the output) and the TCP connection time that is irreducible unless we do not relay anymore on HTTP calls for triggering the function but we use raw sockets. Again, the keep-alive feature cannot be used for the same aforementioned reasons.

Differently from the simulations, in the experimental setting, we use real devices. In the simulations, for the sake of simplicity, we assumed that a node could be able to execute one task at a time and have a queue of length three tasks, this means that the maximum number of tasks in the system is four ($K = 4$), if a new request arrives when the maximum is reached then it will be "blocked", i.e. discarded. This assumption is not very far from reality when we deal with low latency tasks and it is equal to saying

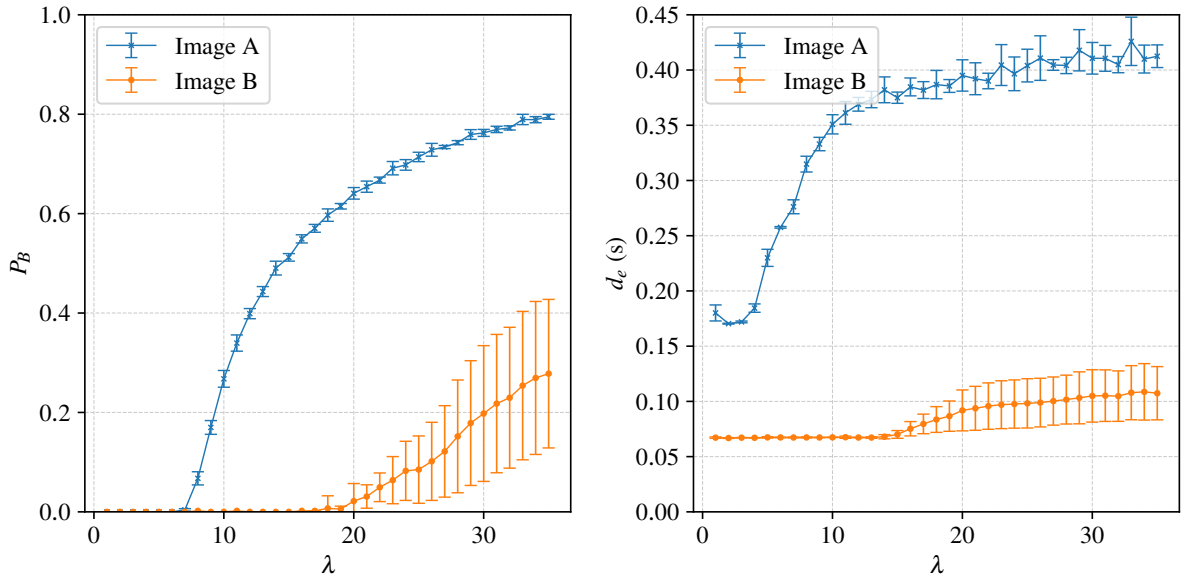
[3] https://github.com/esimov/pigo-openfaas

	Image A	Image B
Resolution	320x210	180x118
Size (kB)	28.3	23.8
d_e (ms)	$180.08 \pm 7.25^*$	$67.19 \pm 0.64^*$
d_t (ms)	$188.24 \pm 7.27^*$	$74.95 \pm 0.81^*$

Table 4.3: Resolution, size and delays of the image payloads used for the tests. The delays are computed on the average of 200 requests with $\lambda = 1.0$, the error is computed from t-Student distribution (p-value = 0.01)

that the node is able to execute four tasks at a time with a single core that obviously interleaves the execution of the tasks. Indeed, the total time for executing all the tasks, on average, is four times their execution. Now, Raspberry Pi 4's CPU has four processing cores, therefore the simulation assumptions do not hold anymore but we expect to have very similar results. However, in this real environment, to have four processing cores is not equal to saying that we have four independent servers, as in the *M/M/k/k* or in the *M/M/1/k* queue models. In real world, due to the underlying operating system (in this case balenaOS) and in the end the Linux Kernel, continuously balance the load among the different cores and it is quite difficult to have a task that strictly holds a core for its entire processing time, moreover there is an underlying operating system which has to be scheduled meanwhile. This means that we cannot know in advance which is the processing capability of a single node and therefore we cannot establish the load to give to each node in order to obtain a desired ρ. Since we do not have a reliable model for a real Raspberry Pi node we proceeded to make an estimate. Therefore, for the two payload images, we used a benchmark script which triggered the execution of a FaaS at gradually increasing (λ) rates, the node is configured without a queue, with only four executable tasks in parallel, and the arrival distribution is deterministic (i.e. inter-arrival time is fixed). Figure 4.2.9 shows the results of this experiment, on the left the blocking probability (P_B) is the percentage of requests that are blocked, while on the right the average execution time d_e. The blocking probability allows us to understand when a node starts to reject requests, in the case of Image A we notice that it starts from $\lambda \approx 7$ reqs/s and for the Image B $\lambda \approx 18$ reqs/s. This result could not be derived in theory, since logically, suppose the Image A and its average processing time of 180ms, a node with a single core should have a maximum processing rate $\mu \approx 1/0.180 \approx 5.55$, with four core this should be multiplied by four but as we have seen it is only 7 req/s, this is justified by the fact the cores are not independent, indeed, we can observe in the Figure 4.2.9b how the execution time increases when the arrival rate (λ) increases and this drastically reduces the service rate (μ) of the node. For example, again for Image A, when $\lambda = 7$ req/s then the average delay $d_e \approx 310$ms, increased about 72% with respect the duration when $\lambda = 1$ req/s.

Concluding, we derived that the service rate is a function of the arrival rate ($\mu(\lambda)$), but since delineating a model of the node is out of scope for this study it is enough to derive the maximum service rate starting from which a node rejects requests, that we call $\hat{\mu}$: for the Image A we have $\hat{\mu}_A = 7$ req/s and for the Image B, $\hat{\mu}_B = 18$ req/s.



(a) Blocking probability (P_B)

(b) Average task execution time (d_e)

Figure 4.2.9: Results of the face detection function from a single node with no cooperation, two payload images have been tested: Image A and Image B. Every point is the average of 500 requests and the confidence intervals have been derived from t-Student (p-value = 0.01), the test has been repeated 10 times.

4.2.4.3 Performance metric

We now introduce a performance metric which will allow us to compare the different scheduling policies. In the experiments, the purpose is obviously to maximize the reward, but an approach that also distributes it with respect one that makes every node behave selfishly is definitely preferable because it will be able to offer the best service across all the geographic domain. For this reason, the performance metrics that we will consider will be:

- ι, that is the sum of the reward divided by the number of tasks completed in a second, that, given the reward definition in Section 4.2.1.5, it is equal to the percentage of tasks that are completed within the deadline
- P_f, the percentage of tasks that have been forwarded, that is used to understand the level of cooperation among the nodes.

Similarly to the metrics of the simulations, we represent their behaviour over the entire simulation time, this is needed to grasp how the learning algorithm improves the performances over time. However, for drawing conclusions more analytically we even consider the last seconds of the tests, where the RL algorithm settled the performances, and then compare not only the mean of the metrics but also the variance, because, as we have seen in the simulations, the learning algorithm tends to equalize the reward among the nodes, therefore a lower variance of the mean ι among all the nodes will be expected, this will be clearer in the next sections.

The set of actions used in all the tests is the one that performed well in the simulations, namely $\mathcal{A}'$ (Section 4.2.1.4) and the Sarsa algorithm is compared with the Power-two-choices, as in the simulations,

but now with threshold $T = 1$, since we now allow $K = 3$ tasks to be executed in parallel. However, we shall remind that a study of power-of-two choices with an M/M/k/k model (the most similar to a real environment), or even better, a model that mimics real nodes has been not studied yet, therefore the $T = 1$ decision is not fully justified, however, we expect to be the best threshold to use for comparison.

4.2.4.4 Results

In this section, we will present the test results of the proposed policy called, as in the simulations, "Sarsa", that run in the presented pseudo-real environment. Table 4.4 shows the complete list of all the experiments performed:

- experiment 1 has been used to test the learning infrastructure, it uses no deadlines, no queues and only one payload;
- the series of experiments 2.x uses the deadlines, the queue is set to 2 and they last 3600s each, their major characteristic is that we tried to estimate the beneficial effect of the learning algorithm with different deadlines since it is clear that the wider the deadline the higher the probability of a task to be completed within it and the higher the rewardwe tested seven different deadlines measured in percentage of the total duration d_t of the task with Image A and Image B;
- experiment 3 instead uses only one deadline but the traffic differently from the other nodes is geographic and derived from the study in Section 4.2.3.3, however, since the study was on 6 nodes we applied the 6 curves to 12 nodes by assigning to two nodes the same curve, for example, the traffic curve of node #1 in the study has been assigned to the Raspberry Pi node #1 and #2, and so on.

In every experiment and for every node, the traffic is distributed according to a Poisson distribution, moreover, the ϵ-greedy policy for action selection starts with $\epsilon = 0.9$ and has a decay of 0.9995 that is applied for each new task.

			Deadlines (ms)					
#	Traffic	Payload	γ	T_A	T_B	K	$\|Q_e\|$	Duration (s)
1	Fixed Heterogenous	Single	-	∞	∞	3	0	3600
2.1	Fixed Heterogenous	Multi	0.00	188.25	74.95	3	2	3600
2.2	Fixed Heterogenous	Multi	0.05	197.66	78.70	3	2	3600
2.3	Fixed Heterogenous	Multi	0.10	207.08	82.45	3	2	3600
2.4	Fixed Heterogenous	Multi	0.15	216.49	86.19	3	2	3600
2.5	Fixed Heterogenous	Multi	0.20	225.90	89.94	3	2	3600
2.6	Fixed Heterogenous	Multi	0.25	235.31	93.69	3	2	3600
2.7	Fixed Heterogenous	Multi	0.30	244.73	97.44	3	2	3600
3	Geographic	Multi	0.05	197.66	78.70	3	2	8000

Table 4.4: Summary of all the experiments performed in the cluster of Raspberry Pi with specification (from the left) of the traffic type, the number of payloads, the deadlines, the maximum parallel tasks K, the execution queue length $|Q_e|$ and the duration.

4.2.4.5 Experiment 1 - No deadline

In this experiment, we used fixed and heterogeneous traffic, in particular, the traffic λ from Node #0 to Node #11 is 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 11 and 12. The image payload used is only the Image A and the learning parameters are $\alpha = \beta = 0.01$ with a window size $Z = 20$. In Figure 4.2.10 we can observe the behaviour of the percentage of tasks that are forward from the perspective of every node, we skipped the chart of the reward because in this first experiment we did not perform any comparison, we only evaluated the behaviour of the learning algorithm. As in the simulations, we can appreciate an initial phase in which the rate is very high, that for the ϵ-greedy approach, the policy progressively stabilizes even not following a gradual approach, this is probably because forwarding to particular nodes (especially the ones that are more loaded) drastically reduces the reward and when this happens the policy changes drastically up to a stabilised situation. Moreover, we can notice how the P_f is not depending on the load, by we remind that in this case, deadlines are infinite, and therefore the learner only tries to execute more tasks as possible without rejecting them.

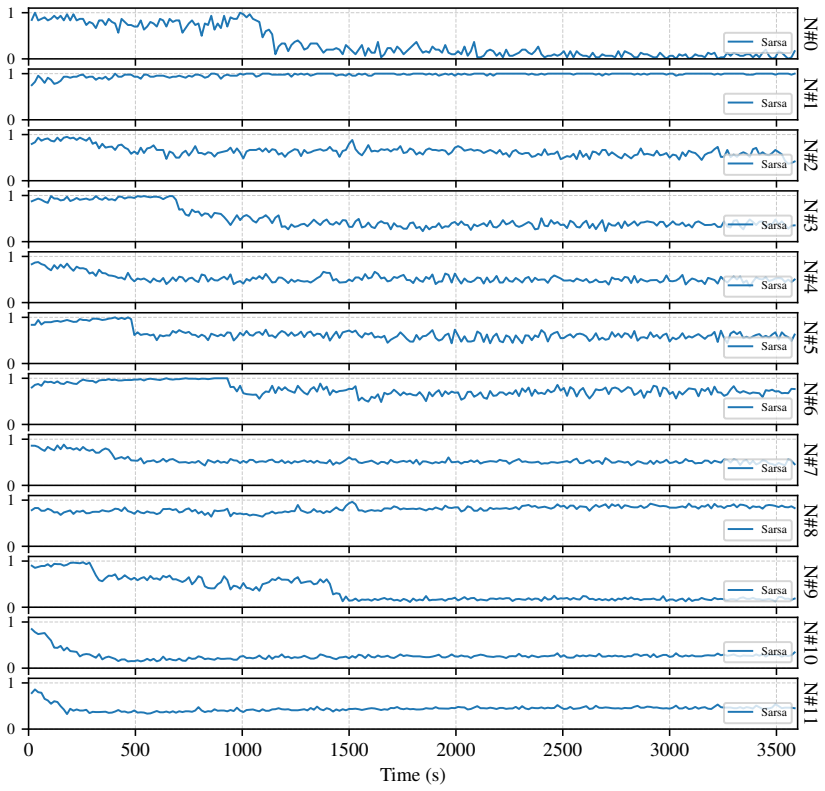


Figure 4.2.10: Experiment 1: P_f, behaviour of the percentage of forwarded tasks during the experiment, average is performed every 15 seconds. In this experiment, deadlines are not considered and only a single payload is used, Image A.

4.2.4.6 Experiment 2 - Fixed load

In this experiment, we introduce, as in the simulations, the concept of deadlineswe use the two presented payloads (Table 4.3) Image A and Image B, assigning two different deadlines. We calculate the deadline on the average d_t as shown in Figure 4.3. The deadline T_X where X is the (Image) A or B is given by equation

$$T_X = d_{t_X} + \gamma d_{t_X} \tag{4.11}$$

For example, for Image A and choosing $\gamma = 0.05$, $T_A = 188.25 + 0.05 \cdot 188.25 = 197.66$mswe suppose that the task arrives in percentage 50/50, and the fixed traffic λ from Node #0 to Node #11 is: 8, 8.5, 9, 9.5, 10, 10.5, 11, 12, 12.5, 13, 13.5, 14. The maximum load of 14 reqs/s derives from the study in Section 4.2.4.2, since $\hat{\mu}_A = 7$ reqs/s and $\hat{\mu}_B = 18$ reqs/s, given the 50/50 distribution of the payloads, the maximum $\hat{\mu}_{A|B} = 0.5 \cdot 7 + 0.5 \cdot 18 = 12.5$ reqs/s. Now, from this value with tried to go slightly over it, reaching 14.0 to understand how the algorithm behaves in a slight overload condition.

Regarding the deadlines, we start from γ = 0, which maps to a deadline equal to the average duration of a task, to $\gamma = 0.3$. The proposed "Sarsa" approach is compared with the Power-of-two choices with $T = 1$, as widely presented in Section 4.2.3.

In every experiment performed, the proposed Sarsa approach performs better of Power-of-two choices "Pwr2", but for space issues, we only report the reward (ι) behaviour of experiment 2.5 (Figure 4.2.11), in which the effect is more evident. What we can observe is a very similar behaviour to the simulations, in particular to the Figure 4.2.6b, indeed in the less loaded nodes, as Node #0 and Node #1 the performances are almost equal to the Pwr2 approach, while, increasing the load, Sarsa performs well, especially in the Node #11 where the average improvement, across all the nodes, is of $\approx$ 20% (performing the average only on the latest 200s of the test). This is expected because, for the least loaded nodes is easier to meet the deadline, since their traffic is lower, on the contrary, the nodes that are heavily loaded struggle to meet the deadline, unless an intelligent policy comes into play. The intelligent policy will forward the tasks to the nodes that are less loaded (and this is inferred by the learning algorithm), instead of forwarding them blindly at random, as the Pwr2 algorithm does.

For having a clearer picture of the results of the proposed algorithm across all the nodes we analysed the mean and the variance of the reward (ι) and of the forwarding probability P_f when the deadlines vary. Figure 4.2.12a shows the comparison of the average reward ι and the forwarding probability P_f in all the tested deadlines. As expected, the reward increases when the deadline increases, the proposed Sarsa approach performs better but inevitably when the deadline increases the difference with the Pwr2 approach reduces, but not progressively. Indeed, the deadline in which Sarsa perform better is for $\gamma = 0.05$, with an improvement of 55% over Pwr2, while when $\gamma = 0.3$ the improvement is 11%. In general, we pass from an in-deadline ι rate of 28% when $\gamma = 0$ to a rate of 85% when $\gamma = 0.3$.

Figure 4.2.12b shows the comparison of variance regarding the reward ι and the forwarding probability P_f. What we can observe is that, when the deadline increases, the variance of the reward ι reduces and this is a clear confirmation of what has been found in the simulations, namely that the Sarsa approach tries to equalize the reward in every node, and this effect is more evident when the deadline is higher. In particular, when the deadline is 0% of the total duration of the task, the variance of the reward is 0.010, while when the $\gamma = 0.3$, the variance is 0.0009. Conversely, this does not hold for

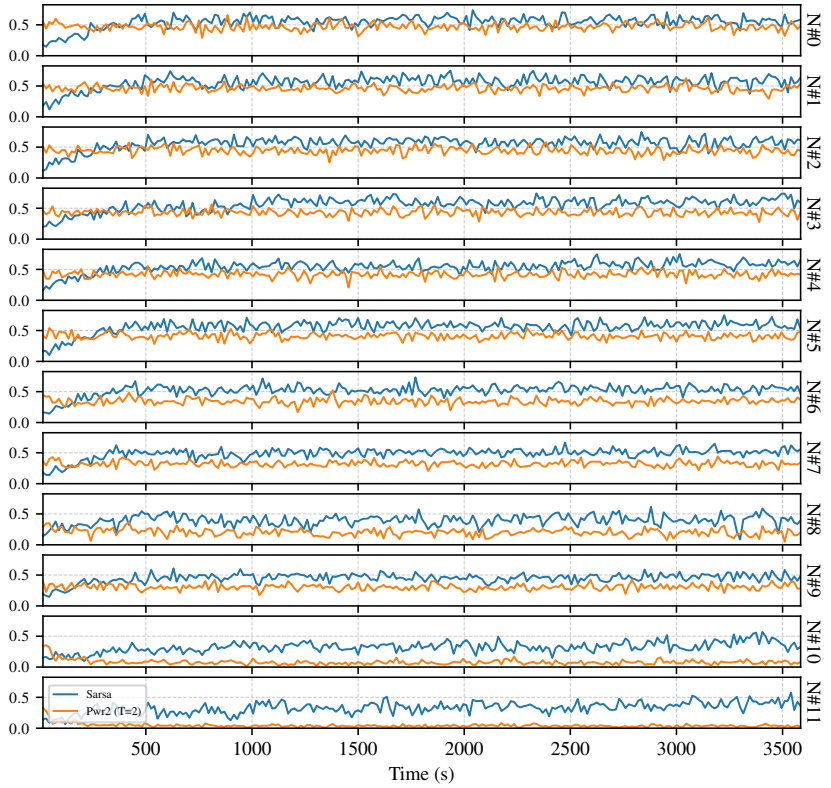
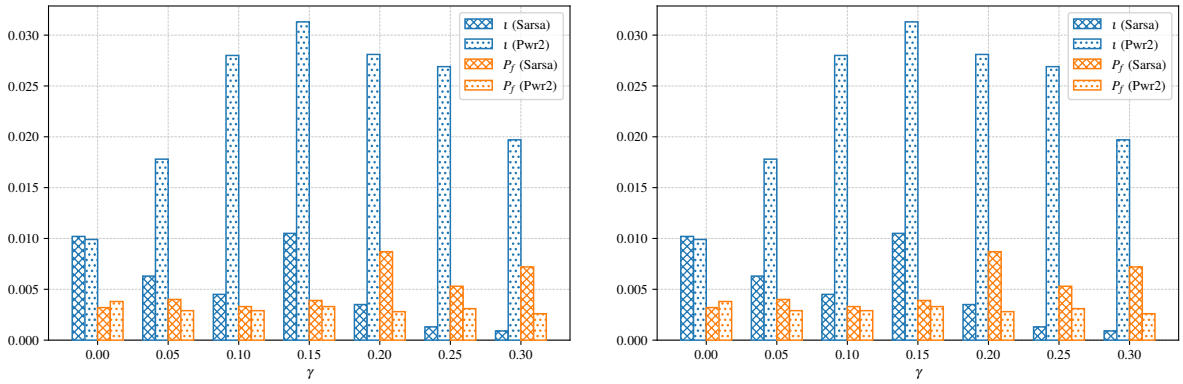


Figure 4.2.11: Experiment 2.2: behaviour of ι during the entire test duration with deadline set at 5% (γ = 0.05) of the total task duration d_t. Values are averaged every 15 seconds.

the Pwr2 approach, in which the variance is higher when the deadline is $\gamma = 0.15$ (that is 0.0105). This confirms that the Pwr2 approach makes the nodes behave selfishly with respect to the Sarsa which tries to reach the best reward rate for every node in such a way no one gains more than another.



(a) Mean **(b)** Variance

Figure 4.2.12: Experiments 2.x: Comparison of average reward ι and forwarding probability P_f across all the 12 Raspberry Pi nodes in the last 200s of a 3600s test.

4.2.4.7 Experiment 3 - Geographical

The final experiment uses a traffic distribution that is derived from the study in Section 4.2.3.3. As already anticipated, the traffic of the 6 nodes is applied to 12 nodes in such a way nodes 0-5 and nodes 6-11 maps to nodes 0-5 of Figure 4.2.3. However, since the traffic curve describes only which is the ρ, namely the load of a single node over time, we scaled the traffic to the range of lambdas from 0.0 to 20.0, therefore when in the curve $\rho = 0.9$ then the actual load to the node is $0.9 \cdot 20 = 18$. As described in the fixed load experiment (Section 4.2.4.6) in which the maximum load was set to 14.0, here we increase it to 20 for understanding how the algorithms behave even in during an overload condition.

The traffic curve is repeated 3 times within a test of 8000s duration. What we can observe is that Sarsa performs better of the Pwr2 choices in almost all the entire duration of the test, except for the initial training phase of the algorithm. However, the most evident improvement can be recognised at the #8, #9, #10 and #11, in particular in the moments in which the traffic is higher, which are between 2000s and 3000s, 4800s and 5800s, and from 7000s to end. In that specific moments, the nodes which match the same traffic load, that are nodes #3, #4 and #5 perform well with Pwr2, they match Sarsa but in the others, Pwr2 is not able to maintain the same level of reward, this behaviour is determined by the hardware differences between the nodes. In real world is quite impossible to have two nodes that have exactly the same performance characteristics, in the experiments in particular, nodes from #6 to #11 are equipped with 8GB of RAM instead of 4GB. Even if this can be counter-intuitive, the nodes with 8GB of RAM are slightly less performing when they saturate with respect to the nodes with 4GB of RAM. This has been shown by repeating the test and changing the node's IPs by leaving unaltered the traffic trace, however, the charts have been omitted for space.

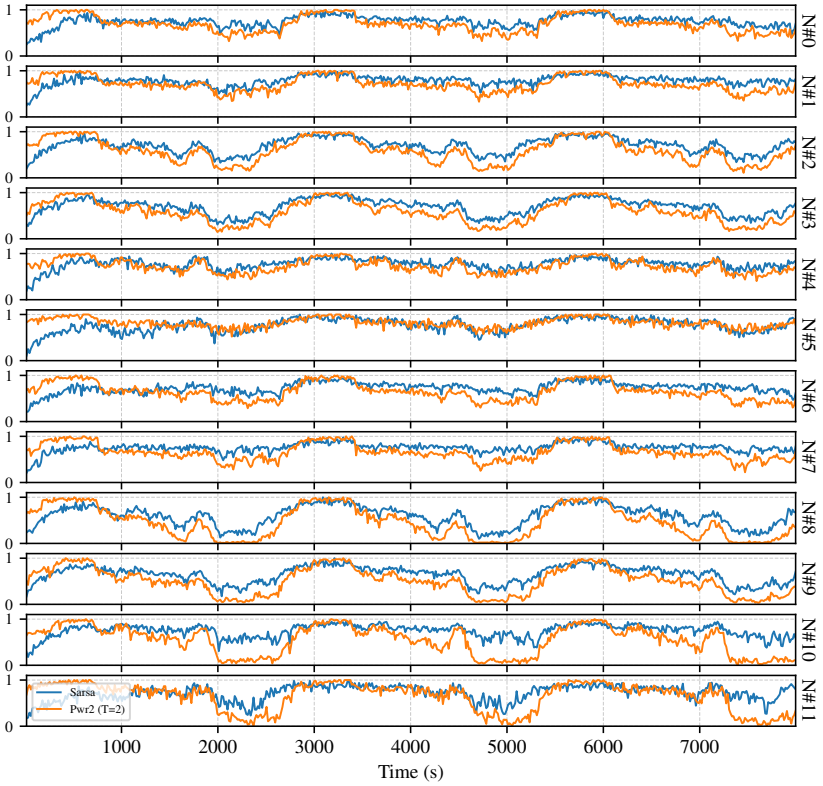


Figure 4.2.13: Experiment 3: behaviour of ι during the entire test duration with deadline set at 5% of the total task duration d_t. Traffic to every node is the one described in 4.2.3. Values are averaged every 10 seconds.

4.3 Online scheduling in Fog environment with clusters and user QoS parameters

In this study, we assume that we have an AR or VR application and a task is a frame processing. This kind of tasks must be real-time and therefore their execution cannot exceed a certain amount of time, because otherwise its output is useless, since the frame must be shown to the user in time. The solution that we propose in this study addresses the scheduling problem of the tasks, that is where to schedule them, by using an online Reinforcement Learning approach.

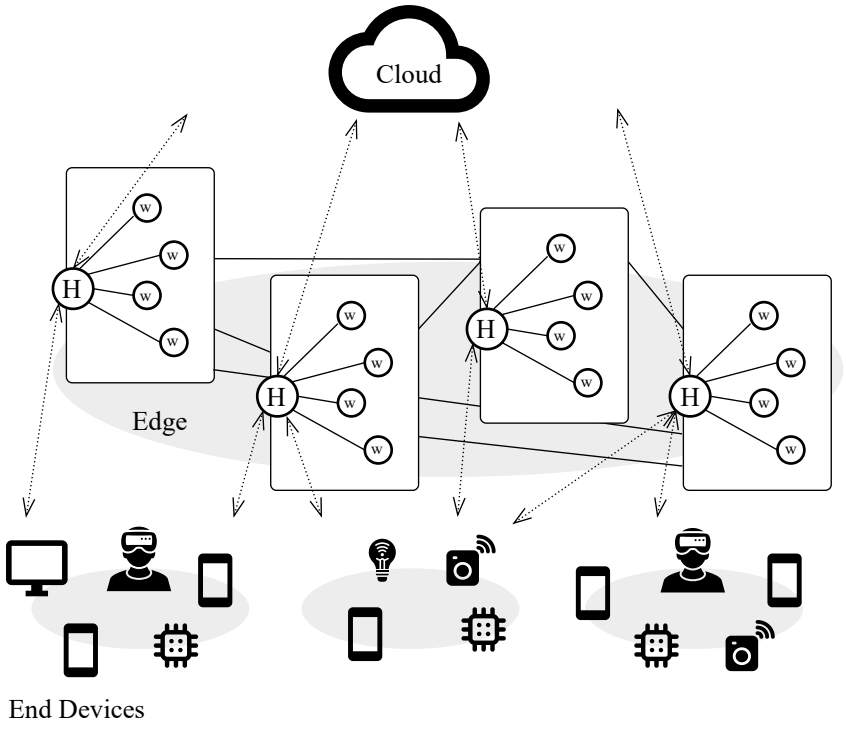


Figure 4.3.1: The Edge-to-Cloud continuum environment considered as computing scenario.

Figure 4.3.1 illustrates the computing environment that is considered. The intermediate layer can be the Fog or the Edge, the only requirement is that we consider it as composed by different clusters and every cluster has a scheduler node and a certain number of worker nodes which may vary over time. For avoiding confusion, in this study, the intermediate computation layer is considered and Edge computing layer. In the scheduler of each cluster we place a *learning agent* which, for every task execution request that comes from the end users, it is able to observe the *state* of the worker nodes and to take an *action* which coincides with a scheduling decision and it can be to execute the task in the worker node w_i, execute it in the Cloud or even reject the request. For every task that is executed within the deadline or near the deadline, the learner will receive a positive reward that will drive its learning process. The online learning approach that is followed in this study is not episodic but it is driven by the average reward that the learner obtain over time. With this approach we make able the Edge-to-Cloud computing continuum to adaptively apply the best possible scheduling policy without

knowing the nodes computing power.

The main contributions of this section can be summarised as follows.

- **Design of a RL based online scheduling algorithm** for the computing continuum that is able to cope with node inhomogeneity and to satisfy user defined processing frame rate requirement;
- **Simulation results** of the proposed algorithm in two main settings, one cluster or more clusters in the Edge layer, within a simulator that is focused on replicating fine-grained delays that a job may encounter during its execution path;

	Environment
$\mathcal{C}$	Set of clusters
$\mathcal{C}_i$	Set of clusters without the cluster i
$\mathcal{W}_i$	Set of worker nodes in the cluster i
$\mathcal{H}$	Set of schedulers
$\mathcal{A}_i$	Set of actions without inter-cluster cooperation for cluster i
$\mathcal{A'}_i$	Set of actions with inter-cluster cooperation for cluster i
T_k	Generic task T of type k
	Learning Parameters
α, β	Learning parameters of Diff. Semi-gradient Sarsa
ϵ	Parameter of the ϵ-greedy strategy for action selection
Z	Completed tasks window that triggers the training process
	Edge/Fog Nodes
S_{ij}	Computing speed of worker j in cluster i
B_{cs}	Bandwidth between a client and a scheduler node
B_{ss}	Bandwidth between a scheduler and another Scheduler
B_{sw}	Bandwidth between a scheduler and a Worker
B_{sc}	Bandwidth between a scheduler and the Cloud
	Times and Delays
ω_n	Nominal rate of frames generation from the device (fps)
ω_m	Minimum frame rate requested for the application (fps)
ω_e	Effective frame rate for processing (fps)
d_e	CPU time for processing a frame (ms) in a worker with $S_{ij} = 1.0$
d_t	Total response time for processing a frame (ms)

Table 4.5: List of symbols used

4.3.1 System Model and Problem Definition

The online scheduling problem that is configured in this study is formalised as a Markov Decision Process (MDP) and its solution is found by using Reinforcement Learning (RL). No prior assumption is done on the underlying mathematical model, therefore a model-free approach is followed. The learning agent observe the current *state* of the environment, performs an *action* by using its knowledge (exploitation) or randomly (exploration). The action coincides with a scheduling decision that is where a task must be executed. Then, after the task is completed a reward signal is obtained, a value that will drive the learning process. The entire process has been wired in a delay-focused discrete events

simulator written by using the Simpy[4] library in Python, but theoretically the solution can be directly and easily applied to a real-world scenario that fits the task model that we are going to present.

What follows in this section is the specification of all the entities that come into play, that are: the environment, task and delay model, the state, and the reward. Finally we will define the performance metric to measure the performances of the proposed algorithm.

4.3.1.1 Environment

As depicted in Figure 4.3.1, we envision a computing continuum environment and we place the learner agent in Edge layer. In particular, this layer is composed by computing clusters that have exactly one scheduler node and a certain number of worker nodes that may vary in each cluster. We call $\mathcal{H} = \{h_1, h_2, \dots\}$ the set of the schedulers nodes and $\mathcal{W}_i = \{w_{i1}, w_{i2}, \dots\}$ the set of worker nodes for cluster c_i, then $\mathcal{C} = \{c_1, c_2, \dots\}$ is the set of clusters, where, for example, in the cluster c_i we have the scheduler h_i and the set workers $\mathcal{W}_i$. For convenience, we also define $\mathcal{C}_i = \mathcal{C} - c_i$, in other words, the set of clusters without the current one i. The scheduler node does not execute tasks but it only receives task execution requests from the underlying clients (end users) and take a decision that can be the rejection or where to execute the task if locally in the cluster (in particular, to which worker node), in the cloud or to forward it to another cluster. This because all of the clusters can communicate with each other. The scheduler node of the cluster i receives a traffic of λ_i requests per second and for each of these requests a scheduling decision is made by using RL. Regarding the worker nodes instead, they can execute one task at a time and they have a fixed size queue that is K. If a task is scheduled on a node and the current number of elements in the queue is equal or greater than K the task is rejected. A peculiarity of these nodes is that they are inhomogeneous, therefore we associate to every of them an execution speed S_{ij} (of the worker i in the cluster j) that is a time extension factor of the tasks that are executed in that node, for example, if $S_{11} = 0.8$ and a task has a nominal duration of 16ms, then on the worker node 1 of the cluster 1 its duration will be $16/0.8 = 20$ms. The concept of execution speed is representing, in the real world, the available CPU time that a worker node can dedicate to execution of the task and it may be subject of fluctuations over time, however, in this study we consider it as fixed and the dynamic case is left as future work.

4.3.1.2 Task and Delay models

In the presented model, we consider tasks as some work that can be executed independently from others and even sandboxed, matching one to one the FaaS paradigm (Function-as-a-Service). Therefore we can see that tasks as function invocations that are then dispatched on a specific worker node. The main characteristic of these tasks is that they have a nominal rate of execution ω_n, that is the nominal processing time on machines in which $S = 1.0$ and the minimum rate of execution ω_m which is the lower-bound admissible for the task, given as a user requirement. As presented later, in the experiments, we define four kinds of tasks (Table 4.8 to which are associated different execution times and constraints.

Regarding the delays that are experienced by the task during its execution, they are simulated by equipping each scheduler node with a transmission queue, in which, as for the execution queue of the

[4] `https://pypi.org/project/simpy/`

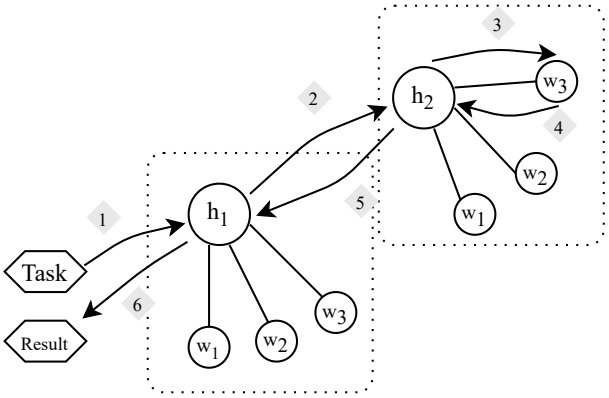


Figure 4.3.2: The task request path from the client to the final worker node when s_1 decides to forward the task to another cluster. The transmission latency is simulated with a transmission queue per scheduler node.

workers, each transmission is elaborated one at a time but the queue has no fixed size. Figure 4.3.2 shows the example path when a task is forwarded to another cluster. First of all, when the task is generated it is added (1) to the transmission queue of s_1 by using the bandwidth between the client and the cluster B_{cs}, when it exits from the queue s_1 makes a decision and then the task is added again to the transmission queue (2) in order to simulate the transmission to the s_2 and the bandwidth is B_{ss}; now, s_2 can only decide to forward the task to a worker or to the cloud, in the image the task is forwarded to w_3 (3) and the bandwidth used is B_{sw} (for the cloud it could have been B_{sc}). After the execution, the task result is returned back to s_2 (4), then s_1 and finally the client (6) using again the same values for the bandwidth and the same payload size for the task.

4.3.1.3 The Agent

The agent is in charge of learning a scheduling policy π that is a function of the state:

$$\pi : \mathcal{S} \rightarrow \mathcal{A} \tag{4.12}$$

Therefore, given a state $s \in \mathcal{S}$ the policy returns an action $a \in \mathcal{A}$, given a generic set of actions $\mathcal{A}$we remind that, given the cluster i, $\mathcal{W}_i$ is the set of workers node in the cluster i. Moreover we define $\mathcal{C}_i$ as the set of clusters, excluding the current cluster i. Beside the action of *rejecting* the task, and forwarding it to the *cloud*, in the first set of experiments (Section 4.3.3.1), the task can be assigned only to the worker nodes. The set of actions for the node i can be described as:

$$\mathcal{A}_i = \{\text{reject}, \text{cloud}\} \cup \mathcal{W}_i \tag{4.13}$$

In the second setting (Section 4.3.3.2) we enable the cooperation even among clusters, and therefore:

$$\mathcal{A'}_i = \{\text{reject}, \text{cloud}\} \cup \mathcal{W}_i \cup \mathcal{C}_i \tag{4.14}$$

We remind that only the scheduler node of a cluster receives the task requests and takes scheduling decisions.

4.3.1.4 State representation

The set $\mathcal{S}$ contains all of the possible states in which the environment can be represented. It is fundamental for the agent, in order to decide which action to perform, to observe a representation of the environment that contain as much as possible information to make the correct decision. In the proposed setting, the only information that we are accessed to is the number of the current scheduled tasks in each given worker node and of which type. This because every task must pass within the scheduler, therefore when a task of type i arrives and it is scheduled to a worker j, a counter for the tasks of type i of worker j is increased by one, conversely when the same task finishes its execution and returns to the scheduler the counter is decreased by one. If it is true that we know the type of the task that is arriving, we cannot know the speed of nodes for the reasons presented in the introduction.

As an example, the Figure 4.3.3 illustrates the state representation for a scheduler node with three worker nodes and two task types. First of all, we have the task type that is an integer (e.g. type t_1 is number 1) that is going to be scheduled, then we associate a tuple for each worker node, describing the number of tasks of that types in the queue.

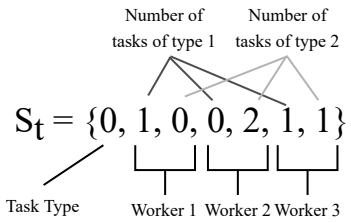


Figure 4.3.3: The state representation of a scheduler node at time t with three worker nodes.

However, the state is not used as is for the learning process, indeed, the tiling [132] technique is used for mapping the vector to another vector but in a 24-dimensional vector space.

As presented in subsection 4.3.3, the task type is an information that is defined by the specific user and it embodies all of the characteristics of the tasks and of its traffic flow, such as, for example, the arrival and desired execution rate.

4.3.1.5 Reward

The definition of the reward is crucial for obtaining the desired results of meeting the user QoS constraints. In the analysed case, the attention is focused on the particular applications in which frames are generated from the devices and they must be processed one-by-one by a back-end server (e.g. AR application) and the result of the elaboration is shown to the user to a screen or to a VR headset, supposing that the refresh rate of the screen is the same of the frame generation and they are synchronised, namely when the screen is refreshed a frame is sent to the server (with a minimal oscillation depending on a Gaussian distribution). The constraint that we want to impose is that in the client, which generates frame at ω_n, there is no lag or frame loss but we can tolerate a minimum response frame rate from the server (called ω_e) to be equal to ω_m. For understanding the best possible definition of the reward which allows us to achieve the desired result, we analyse which are the main situations in which the frames received. In the Figure 4.3.4 four main possible cases are identified, the

general idea behind of the scheme is we can define three main qualifiers for performances:

- the effective rate of frame processing ω_e, which only depends on the machine that will execute the task (i.e. the execution time of the task in the worker node);
- the lag τ which instead mainly depends on the network delays;
- if the order of received frames is the same of the ones sent. This last case is not studied in this study, but in general, if a frame f_1 is sent before f_2 and the result of the processing f_2, r_2, reaches the client before r_1 then r_1 may be lost and not shown to the user if the application has no buffer, this is left as future work.

These points are used for deriving the following cases. In case (a), the rate of the received frames ω_e is the same as the one sent ω_n. Moreover responses are returned before the next frame generation time ($1/\omega_n$), this is the best case and the user will be allowed to have an experience without noticing that frames are offloaded to a server, this because, supposing that the refresh rate of the screen is the same of the generated frame, the next frame will contain the results of the elaboration from the server. In the case (b) instead, we suppose that responses do not arrive before the generation of the next frame but still $\omega_e = \omega_n$, in this case the user will experience a lag, that in this study is computed in seconds but in the end, from the user point of view, is the number of the frames that are skipped, because the refresh of the screen. For example, if the response r_1 of f_1 arrives after the generation of f_2, r_1 cannot be shown on the refresh tick of f_2 but of f_3 so one frame has been skipped. In the case (c), we suppose that there is again a lag but the rate of the responses is not the same of the generation ($\omega_e \neq \omega_n$). The user will experience a drop in the frame rate with a lag, and the device may adapt its ω_n in order to match the one of the server. The latest case (d), as introduced earlier, describes the case in which the order of the received frame is not the same rate of the generation, in this case, depending on the application, frames can be skipped, the study of the reward with this case is left as future work.

In the light of these cases, we defined the reward as shown in Figure 4.3.5. Let us focus on a specific traffic flow and consider one generated frame of this flow, say f_1. Let r_1 be the result of the processing of the frame f_1 received by the client, and let d_t the time elapsed from when f_1 was generated. The reward is defined as follows:

$$R(s,a) = \begin{cases} 2 & \text{if } d_t \leq 1/\omega_n \\ 1 & \text{if } 1/\omega_n < d_t \leq 1/\omega_m \\ -1 & \text{if } d_t > 1/\omega_m \end{cases} \tag{4.15}$$

Where s is the state seen by the scheduler when frame arrived, a the chosen scheduling action chosen, ω_n and ω_m are specific of the given traffic flow.

However, the reward received by the agent is never immediate after a scheduling action, this because only when the task returns to the client we can know its total execution time. For this reason, we maintain a window of Z executed tasks and updated the weights only after all the tasks in the window terminated and reached the client.

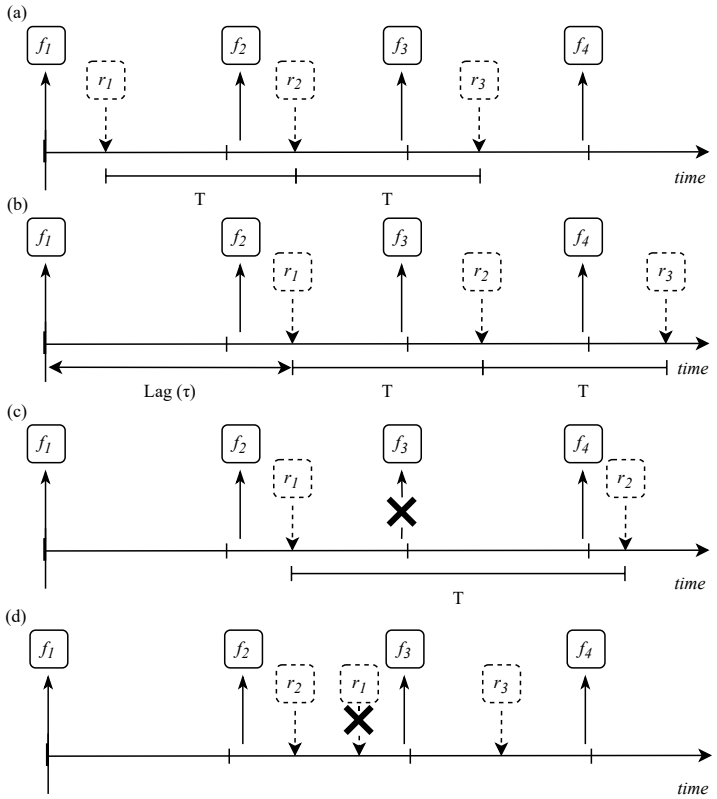


Figure 4.3.4: The frame generation and process in real-time applications, four cases of identifying performances. In case (a) rate of frame generation is the same of the frame result receiving and there is no lag, in case (b) a lag is introduced, in case (c) the rate of the received frame is different by the one of the frame sent, finally in case (d) the order of received frames is altered.

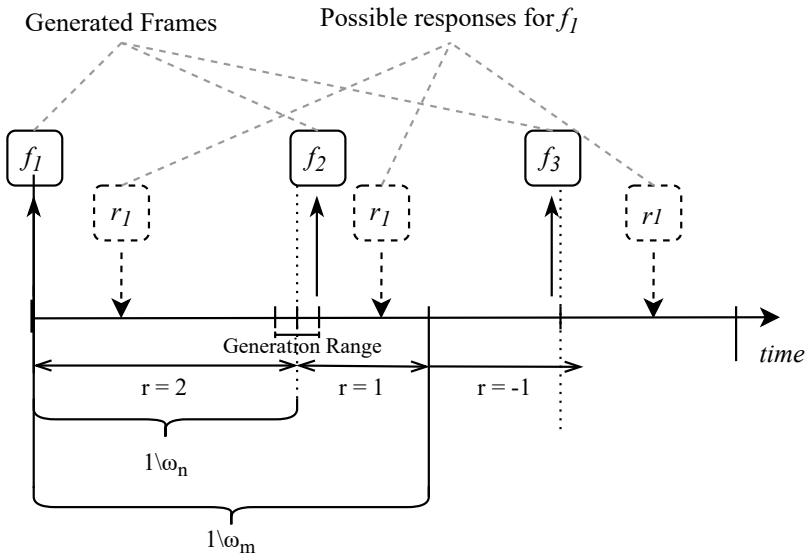


Figure 4.3.5: Diagram that illustrates how reward is assigned when the task is executed and the frame f_1 returns to client after being processed (r_1).

4.3.1.6 Performance Parameters

To understand the performance of the proposed RL-based scheduling algorithm, we delineated the following performance parameters that are computed and shown over the simulation time:

- the *total reward* (R), defined as in Equation 4.15;
- the *effective frame rate* (ω_e) measured in frames-per-second and computed as the sum of the total number of frames successfully processed every second (not rejected);
- the *total response time* (d_t) measured in milliseconds and computed as the average response time of all the tasks finished every second;
- the *lag time* (τ) measured in milliseconds and computed as (depicted in Figure 4.3.4.b):

$$\tau = d_t - 1/\omega_n \tag{4.16}$$

4.3.2 Online scheduling decisions with RL

The final objective of the agent is the one of learning a scheduling policy π that maximizes the long-term reward. Since each decision must be taken online, we cannot envision episodes but we treat the problem as a continuing learning task.

In a continuing learning task it is not useful to discount future rewards but it is better considering the current average reward for taking the right direction. Given a state $s \in \mathcal{S}$, we perform the action $a \in \mathcal{A}$, we obtain the immediate reward r the next state is $s' \in \mathcal{S}$ then the optimal policy (that is the policy which maximizes the long-term reward) will result in the optimal q_* function defined as [132]:

$$q_*(s,a) = \sum_{r,s'} p(s',r|s,a) \left[r - \max_{\pi} r(\pi) + \max_{a'} q_*(s',a') \right] \tag{4.17}$$

Where $r(\pi)$ is a function which returns the average reward of the policy π. At certain time t, by using the Sarsa algorithm for learning the policy and given the weights vector $\vec{w}$, the differential form of the error can be expressed as [132]:

$$\delta_t = R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \vec{w}_t) - \hat{q}(S_t, A_t, \vec{w}_t) \tag{4.18}$$

This form can be applied to any function approximation algorithm for estimating the q_*, in the analysed case we used the tiling technique [132]. As already introduced, in the proposed setup, the reward is never immediate because we know it only after a task has been executed or rejected and it returned to the client, for this reason we set a window size of Z tasks and right after the execution of every task we check if the window is reached and every task in the window has been executed or rejected, if this is true, then the weights are updated for all the tasks in the window.

The Algorithm 11 is run by the scheduler i whenever a new task to be executed arrives, supposing the set of action $\mathcal{A}'_i$ (with the inter-cluster cooperation). First of all, we append the task to the array of pending tasks ("TasksArray") then we compute the state (as described in Section 4.3.1) and we retrieve the best action to perform given the current $q(s, a, \vec{w})$. If the action is 0, then the task is immediately rejected, if it is 1, then the task is forwarded to the cloud, otherwise, we check the action number and we derive the index of the worker of of the cluster to which the task must be forwarded. In particular, in the case in which the task is scheduled to be executed in a worker node we check if the current queue length is equal or exceeding the limit K, because in that case the task is rejectedwe remark that, once a task has been forwarded to a worker or to the cloud, then it will be necessarily executed there if room, otherwise it will be rejected, therefore no further decision is taken for its scheduling.

Algorithm 11 Scheduling Decision (scheduler of cluster i)

Require: Scheduler, Task, TasksArray, $\vec{w}$, $\mathcal{A}'_i$, $\mathcal{W}_i$, $\mathcal{C}_i$, K
  TasksArray.append(Task)
  $s \leftarrow$ aggregate(Scheduler.getWorkersLoad(), Task.getType())
  $a \leftarrow \max_{a \in \mathcal{A}'_i} q(s, a, \vec{w})$ with prob. $1 - \epsilon$ otherwise random($\mathcal{A}'_i$)
  Task.saveStateAction(s, a)
  if a == 0 **then**
    Scheduler.reject(Task)
  else if a == 1 **then**
    Scheduler.forwardToCloud(Task)
  else if $a > 1$ **and** $a < |\mathcal{W}_i| + 2$ **then**
    workerToForwardTo $\leftarrow$ Scheduler.getWorker($a - 2$)
    if workerToForwardTo.getQueueLength() $< K$ **then**
      Scheduler.forwardToWorker(workerToForwardTo, Task)
    else
      Scheduler.reject(Task)
    end if
  else if $a \geq |\mathcal{W}_i| + 2$ **and** $a < |\mathcal{W}_i| + |\mathcal{C}_i| + 2$ **then**
    Scheduler.forwardToCluster($a - |\mathcal{W}_i| - 2$, Task)
  end if

Every time that a task completes its execution (which means that result payload of the task is returned to the client), whether it is local or remote, Algorithm 12 is executed. First of all, we record the task reward and then we start to iterate over the array of pending tasks ("TasksArray") to check if the first Z tasks of the array are finished. If this is not the case, the function returns otherwise we go on by

retrieving the information about the first Z tasks by popping them from the array. This information is used to train the weights vector $\vec{w}$ using the semi-gradient differential Sarsa algorithm.

Algorithm 12 Learning with Differential Semi-Gradient Sarsa

Require: Task, TasksArray, Z, $\vec{w}$, $\bar{R}$, α, β
  Task.setReward()
  i $\leftarrow$ 0
  for all j in TasksArray **do**
    if !j.isDone() **then**
      return
    end if
    if i $==$ Z **then**
      break
    end if
    i $\leftarrow$ i + 1
  end for
  i $\leftarrow$ 0
  $j_0 \leftarrow$ TasksArray.pop(0)
  $s \leftarrow j_0$.getStateSnapshot()
  $a \leftarrow j_0$.getAction()
  $r \leftarrow j_0$.getReward()
  for i = 0; i $<$ Z; i++ **do**
    $j \leftarrow$ TasksArray.pop(0)
    $s' \leftarrow j$.getStateSnapshot()
    $a' \leftarrow j$.getAction()
    $\delta \leftarrow r$ - $\bar{R}$ + $q(s', a', \vec{w})$ - $q(s, a, \vec{w})$
    $\bar{R} \leftarrow \bar{R}$ + $\beta\delta$
    $\vec{w} \leftarrow \vec{w}$ + $\alpha\delta\nabla q(s, a, \vec{w})$
    $s \leftarrow s'$
    $a \leftarrow a'$
    $r \leftarrow j$.getReward()
  end for

4.3.3 Results

In this section, we present the results of the simulation in the described environment of the proposed RL-based scheduling algorithm both in a single cluster (Section 4.3.3.1) and in a multi-cluster setting (Section 4.3.3.2). The results are structured as follows.

First of all, in the single cluster case we show that the agent is able to learn a scheduling policy that matches the requirements provided by the users, this is given by the fact that it manages to learn the nodes speeds that are unknown to it. The proposed approach not only make it possible to reach the desired frame rate for each traffic flow but also minimises the lag time. Then, by using the same setting we simulate a failure of a node, the faster one, and we observe that the agent is able to recover the situation by dynamically adjusting the scheduling policy. In the second part of the section, we apply the proposed algorithm in a multi-cluster environment by simulating three different clusters that can also cooperate.

In all of these experiments, the execution speeds of the workers in the clusters, i.e. S_{ij}, and the maximum queue length K, have been derived from the technical parameters of real devices as shown in Table 4.6. Specifically, the service rate is normalised with respect to the highest clock speed in the

group, e.g., the service rate of the Asus Tinker (1.8 GHz) is $\S = \frac{1.8}{2.0} = 0.9$. The cloud instead runs always with a speed equal to 1.0.

The traffic flows arriving to each cluster is defined in Table 4.6. Table 4.7, instead, shows the network parameters used in the simulations.

Brand name	Frequency	Parallelism	S	K
Odroid-C4	2.0 GHz	4 cores	1.0	4
Asus Tinker	1.8 GHz	4 cores	0.9	4
Rock Pi N10	1.4 GHz	4 cores	0.7	4
Raspberry Pi 3	1.2 GHz	4 cores	0.6	4

Table 4.6: Specifications of worker nodes used in the experiments.

Parameter	Value	Description
d_c	20ms	Round-trip time between Scheduler-Cloud
B_{cs}	200Mbps	Client - Scheduler bandwidth
B_{ss}	300Mbps	Scheduler - Scheduler bandwidth
B_{sw}	1GBps	Scheduler - Worker bandwidth
B_{sc}	1GBps	Scheduler - Cloud bandwidth

Table 4.7: The specification of the network parameters in the simulation.

4.3.3.1 Single Cluster

The setting of this series of experiments if depicted in Figure 4.3.6. We have one single cluster which receives exactly four flows of traffic, described in Table 4.8. The first three flows, namely tf_1, tf_2 and tf_3 represent three hypothetical users which require respectively a processing rate ω_n of 60, 30 and 15 fps and they tolerate a minimum service frame rate ω_m of 50, 20 and 10 fps, respectively. The tasks of these flows, arriving to the cluster, are periodic and the inter-arrival time is picked from a Gaussian Distribution with $\mu = 1/\omega_n$ and σ as described in the table. Then there is a fourth flow that is not periodic but the inter-arrival time is picked from an exponential distribution for simulating a background traffic to the cluster.

The duration time of a single task, that is a frame processing, is again picked from a Gaussian distribution with $\mu = d_e$ and $\sigma = 0.0003$, the d_e value differs among the different task types and it is referring to the execution of the task in a worker that has execution speed $S = 1.0$. The payload of each task, independently from its traffic flow, it is fixed at 50kb.

Normal Operation The Figure 4.3.7 shows the performance metrics of the proposed algorithm in a simulation with the single cluster and the already discussed conditions. Metrics are plot over the simulation time, we have the reward in the first line, then the effective frame rate ω_e, the total response time d_t and the lag time τ. Specifically, for the ω_e and the d_t we also show the desired ranges in which

	ω_n	ω_m	**Distr.**	σ	d_e	**Distr.**	σ	**Payload**
tf_1	60 fps	50 fps	G. Periodic	0.001	10 ms	Gauss.	0.0003	50 kb
tf_2	30 fps	20 fps	G. Periodic	0.002	20 ms	Gauss.	0.0003	50 kb
tf_3	15 fps	10 fps	G. Periodic	0.01	55 ms	Gauss.	0.0003	50 kb
tf_4	10 fps	-	Exp.	-	100 ms	Gauss.	0.0003	50 kb

Table 4.8: The list of traffic flows used for the simulation, each traffic flows tf_i generates tasks of type i.

the metrics must reside, that are the ones requested by the users. What we can observe is that after the initial phase in which the ϵ parameter of the ϵ-greedy approach is progressively reduced in order to favouring exploitation over exploration (that is choosing the action at random), the learner in the scheduler node manages to reach the desired requisites for the three flows tf_1, tf_2 and tf_3. In particular, we can see how the designed reward scheme allows, at the same time, to reach the desired ω_n and the d_t and to reduce the lag time τwe remind that, if the reaching of the desired frame rate ω_n is matching a correct scheduling decision to the correct worker in the correct moment, and we remark that the agent does not know which is the speed of any of the workers in the cluster, the lag time strictly depends on the network latency. This is why, for example, tf_3 cannot reach a low value of the lag τ.

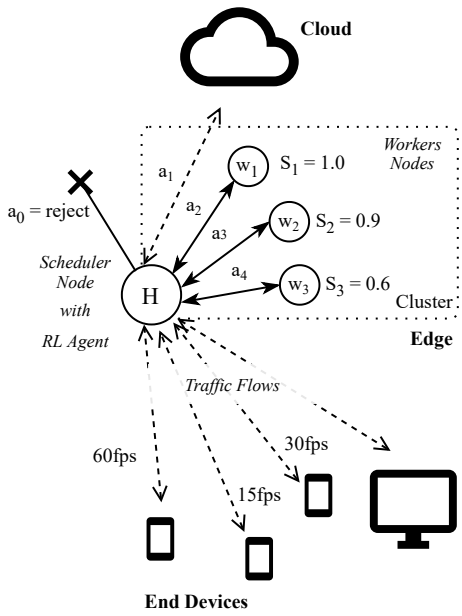


Figure 4.3.6: The setting of the of the experiments on a single cluster with three workers nodes and four traffic flows (Section 4.3.3.1).

The Figure 4.3.8, instead, shows the percentage of the actions that are chosen by the agent in the scheduler node over time. What we can observe is that the workers' speed is learned well and very fast, this because distribution of the action follows the speed of the worker nodes, indeed, the worker #1, the faster, is the most chosen, then we have worker #2 and worker #3. We also can see that the cloud is chosen as well, this essentially because there are the background noise of the tasks that have no deadline.

Finally, the Table 4.9 shows the comparison of the proposed algorithm, after the training phase (referred as "Sarsa Trained") and other two scheduling strategies: the least loaded approach, which always schedules the action to the least loaded node, i.e. with the lowest queue length, and the random, which chooses the worker node to schedule to the task at random. As shown in the charts, the proposed approach allows us to meet the user requirements and minimise the lag time τ, this is because the objective is to maximize a reward based on the total response time of the tasks.

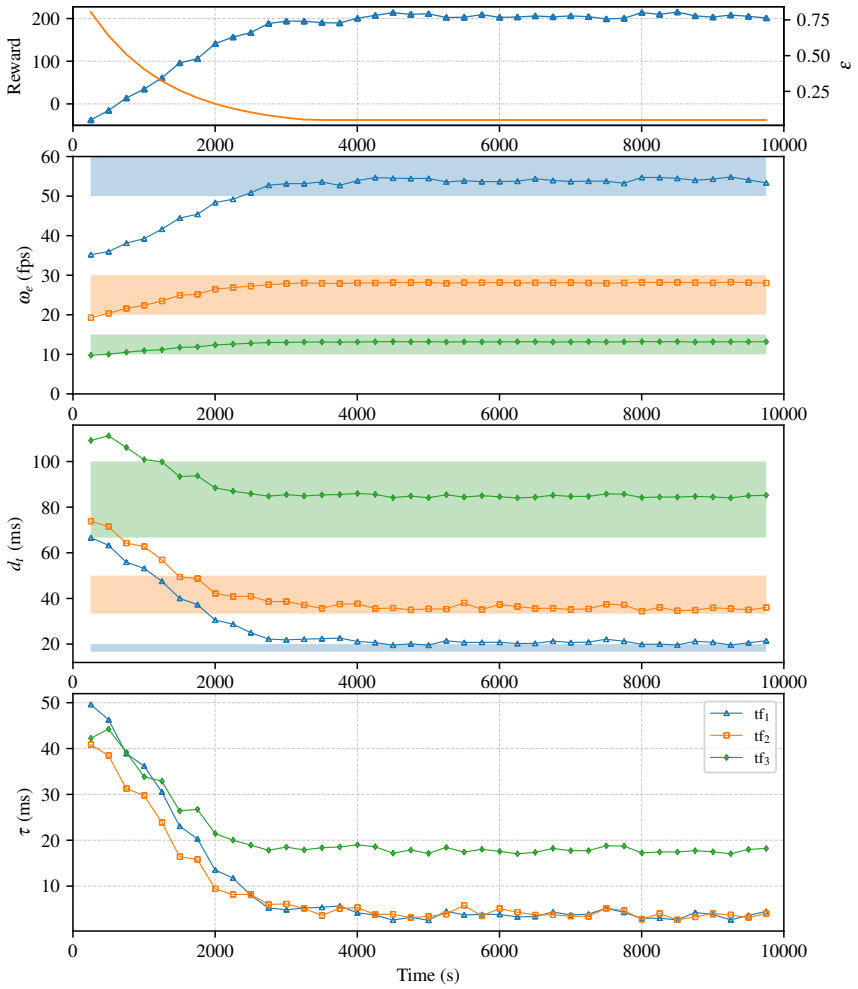


Figure 4.3.7: Results of the simulation of a single cluster and three worker nodes (Section 4.3.3.1), regarding, from top to bottom, the reward, the effective frame rate ω_e, the total response time d_t and the lag time τ.

Failures In the same setting of the single cluster we simulate that the faster worker, worker #1, after 4000s fails and each task request sent to him is rejected. The Figure 4.3.10 shows the results of the simulation with the same structure of Figure 4.3.7. As we can see, when the worker #1 fails there is a drop in the reward, in the frame rate ω_e and in the response time d_t, and for the tf_1 the requirements are not met anymore, tf_2 finds its response time to increase but still in the requirements and finally the lag time is increased both for tf_1 and tf_2 of about 5ms. However, the proposed approach finds a new scheduling policy for solving the problem and restoring at least the response time requirement and the effective frame rate of tf_1.

In the Figure 4.3.10 we can appreciate which are the modifications done to the scheduling policy after the failure. Almost immediately the percentage of actions for scheduling towards node #1 drops, moreover, more tasks are scheduled to worker #2, #3 and to the cloud. The change in the actions is generated from the fact that the learner starts to receive negative reward when scheduling to the node #1, indeed the percentage of actions towards it progressively reaches zero, then, again the faster worker (worker #2) receives more traffic than the other (worker #3).

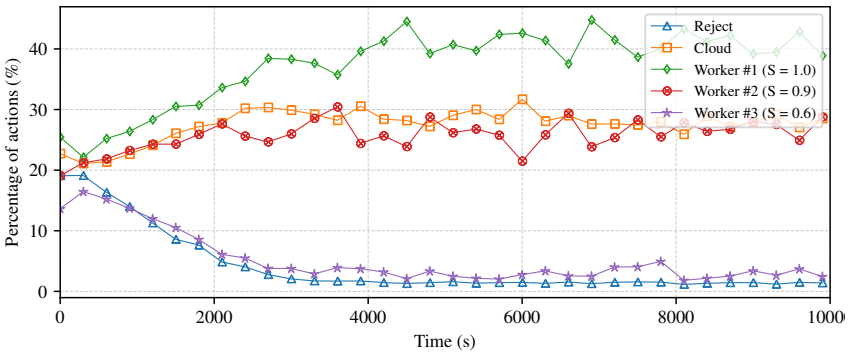


Figure 4.3.8: The distribution of the actions made by the agent on the scheduler node in the experiments with a single cluster and three worker nodes (Section 4.3.3.1).

	Sarsa Trained			*Least Loaded*			*Random*		
	ω_e	τ	d_t	ω_e	τ	d_t	ω_e	τ	d_t
tf_1	54.08	19.63	20.57	37.85	48.75	96.05	29.61	65.23	100.71
tf_2	28.15	36.00	35.69	19.04	72.22	110.50	15.72	88.35	109.64
tf_3	13.17	76.34	84.71	9.52	120.35	148.80	8.11	142.38	144.25

Table 4.9: Comparison regarding the effective frame rate (ω_e), the lag time (τ) and the total response time d_t between the proposed algorithm "Sarsa Trained" and other two approaches: scheduling to the least loaded node and random scheduling.

4.3.3.2 Multiple Clusters

In this setting we suppose to have multiple clusters which can cooperate, therefore the set of actions used by the scheduler i is now $\mathcal{A}'_i$, for each schedulerwe suppose that each cluster receives the flows described in the Table 4.8 and we use the proposed algorithm to each scheduler of each cluster. The clusters are the following:

- cluster #1 has three nodes with speeds 1.0, 0.9 and 0.6;
- cluster #2 has two nodes with speeds 0.9 and 0.6;
- cluster #3 has three nodes with speeds 1.0, 0.7 and 0.6.

The Figure 4.3.11 shows the results of the simulations, in particular we can observe that behaviour of the reward, but also of the the effective frame rate ω_e, the total response time d_t and the lag time τ is similar to the single cluster setting, for this reason the chart of these last three parameters has been omitted. However, in this setting, is relevant to notice the behaviour of the decisions taken by the schedulers, shown in the last three lines of Figure 4.3.11. As the first cluster setting, the agents are able to derive the speeds of the worker nodes and pick the best allocation but now part of the traffic goes to the nearby cluster and more than of the one that is forwarded to the slower nodes. This means that agents prefer to forward some tasks to other clusters, instead of executing them in the current one if the worker are slower, and this was perfectly expectedwe remind that the agents have no information about the other clusters, the representation of the state is always the same of the one presented in

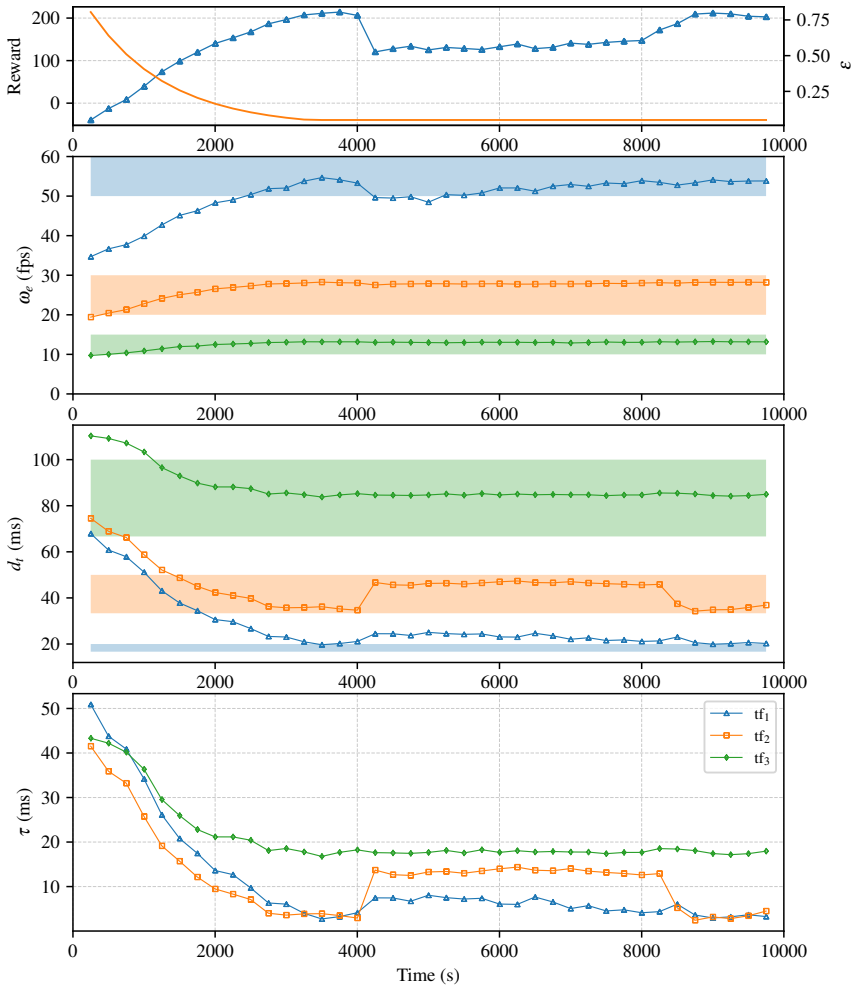


Figure 4.3.9: Results of the simulation of a single cluster and three worker nodes (Section 4.3.3.1), regarding, from top to bottom, the reward, the effective frame rate ω_e, the total response time d_t and the lag time τwe assume that node #1 fails at time 4000.

Section 4.3.1.

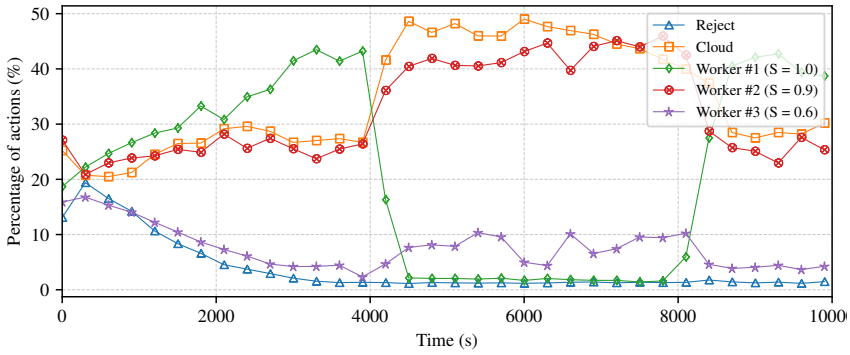


Figure 4.3.10: The distribution of the actions made by the agent on the scheduler node in the experiments with a single cluster and three worker nodes (Section 4.3.3.1)we assume that node #1 fails at time 4000.

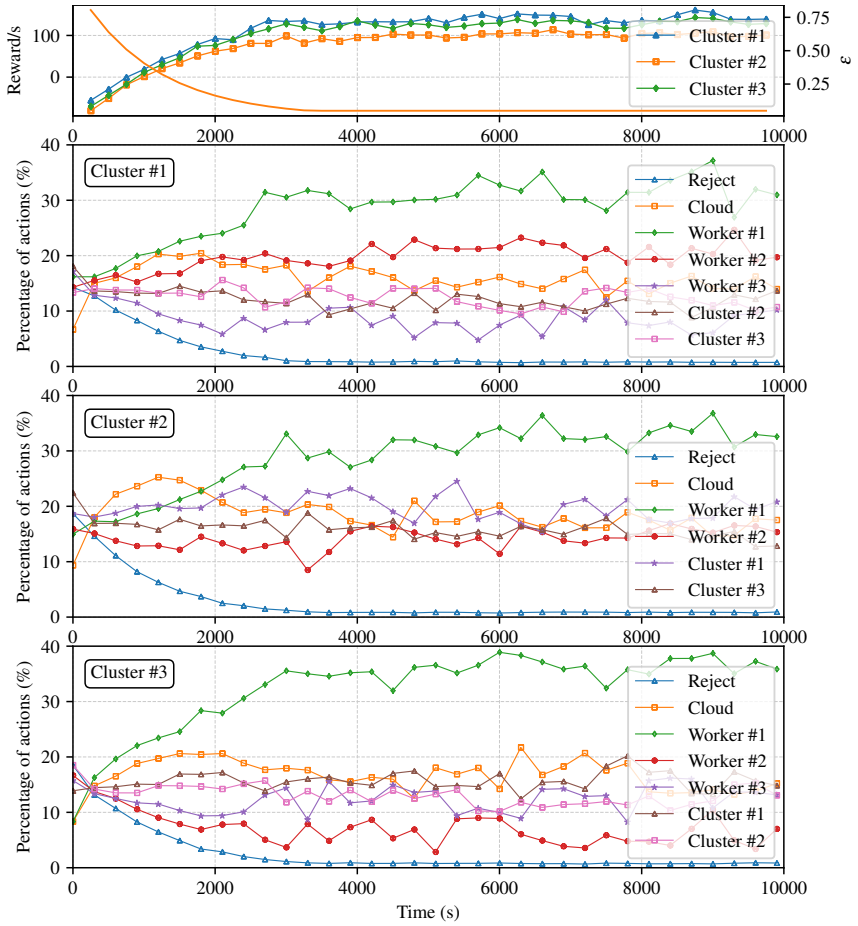


Figure 4.3.11: Results of the simulation in a three clusters setting, regarding, from top to bottom, the reward per second, the percentage of the chosen action over time for the three clusters, in order.

Chapter 5

Energy-oriented Studies

> If you want to find the secrets of the universe, think in terms of energy, frequency and vibration.
>
> Nikola Tesla

The energy aspect of devices has particular relevance, mainly in Green Edge Computing environments, where devices use energy from solar panels but also in smartphones which are particularly sensitive to the battery duration. In this Chapter, we gathered two works whose principal purpose is analysing and managing the power consumption of the devices and studying possible offloading mechanisms. The first work, presented in Section 5.2, is a study of the energy/latency tradeoff which exists when a smartphone off-loads intensive ML tasks to a Fog or an Edge node. The work presents the study of different mobile computer vision set-ups for evaluating the energy requirement that they need to be performed and the improvements that it is possible to achieve if the inference is run on the edge or in the cloud. In particular, the experiments show that offloading the task (in my case a real-time object recognition) to a possible next-to-the-user node allows saving about the 70% of battery consumption while maintaining the same framerate (fps) that local processing can achieve. Then, in Section 5.3 we report a study of possible energy-oriented load balancing policy which specifically targets Green Edge Computing. In particular, we consider the case of a group of communicating Edge nodes only supplied with photovoltaic (PV) panels. Since the amount of storable energy is limited by the battery capacity, the solar energy at an Edge node with fully charged battery cannot be further accumulated, which is an indirect source of inefficiency. As a consequence, an Edge node which is running out of green energy can conveniently offload its computations to energy-richer

nodes in idle states. The work reports some preliminary results of this form of green cooperation.

The solutions presented in Section 5.2 and Section 5.3 has been respectively published in [10] and [5].

5.1 Related Work

Mobile neural networks Different works investigated the energy efficiency of neural networks or tried to provide solutions for assisting mobile devices in computer vision tasks. Neural networks represent the core of deep learning [146], they allow performing various machine learning tasks, starting from the simple classification to the generation of new data [147] by using any kind of input data, like numbers, images and audio. The major challenge, that mobile devices open, is the one of making the network most efficient as possible in order to drastically reduce power consumption without losing accuracy. This can be done in different ways [148] as pruning some parts of the network, decomposing tensors and quantising the weights. Every optimisation is done to reduce the number of operations that needs to be performed for running the network.

In [148] a method for compressing a CNN is proposed. The method is composed of 3 steps and it is a one-shot process that can be easily implemented by using publicly available tools. In [149] are presented a series of lightweight neural networks, called *MobileNets*, that are optimised for running in mobile devices, in particular, they use the "depthwise separable convolution", a form of factorised convolution which is able to drastically reduce the computation load and the size of the model. Finally, in [150] an example of an efficient neural network for mobile application is designed and evaluated. The network uses "pointwise group convolution" as for MobileNets, and "channel shuffling", a technique which allows exploiting the convolution in a more computation-wise manner.

Frameworks for mobile neural networks Another set of works is not just focused on the design of lightweight neural networks but they are aimed to provide a complete solution for allowing deep learning tasks on mobile, both on their own and with the assistance of a backend server. Frameworks like the ones presented in [151]–[153] support computer vision tasks on mobiles by using the GPU and for supporting real-time applications, in particular, they implement an engine that is able to load and run CNN models in a fast and energy-efficient manner. Otherwise, many solutions allow offloading the deep learning tasks to the Edge, for example, in [154] shows how is possible to have a synergy between the mobile and the Edge taking into account parameters like video quality, battery consumption, accuracy and latency in the case of an AR application. A similar proposal is presented in [155] but not designed for real-time applications.

In all of these works what is missing is a clear depiction of the actual energy consumption gain of the offloading task, mainly when we want to use open-source frameworks that are freely available on the web.

Green Energy for Edge Computing Several papers address the problem of energy efficiency in Fog Computing via load balancing or proper resource allocation. [156] proposes an energy-aware load balancing and scheduling (ELBS) method based on Fog computing. The work reports an energy

consumption model of the workload on the Fog node. [157] designs a novel Energy-aware Data Offloading (EaDO) technique to minimise energy consumption and latency in the industrial environment. [158] studied a sustainable infrastructure in Fog-Cloud environment for processing delay-intensive and resource-intensive applications with an optimal task offloading strategy. The proposed offloading strategy optimises two Quality-of-Service (QoS) parameters such as energy consumption and computational time. The model in these papers includes a cloud layer where the computation is eventually performed. [159] presented an energy-efficient Fog architecture considering the integration of renewable energy. Three resource allocation algorithms and three consolidation policies were studied.

5.2 Energy and latency tradeoff in local and remote offloading of ML tasks

Image-processing-based applications for smartphones and mobile devices are growing at an extraordinary rate due to the level of maturity achieved by most support technologies (e.g. computer vision, hardware, AI). Application domains range from immersive multi-gaming, online shopping, virtual browsing, remote assistance, etc. Experts predict that the Augmented/Virtual Reality (AR/VR) industry will reach over $25 billion by 2025 and that growth will continue steadily.

Cloud VR/AR services are currently offered by the main cloud providers, but they will hardly meet both the above conditions. Edge/Fog computation capability will likely complement and improve these services, however, running these tasks directly in mobile devices is progressively possible with inference latencies, for example in the case of Convolutional Neural Networks (CNN), that are comparable to the ones that before was only expected on high-end GPUs [150] – obviously with less but still acceptable accuracy. These results can be achieved, not only by using the CPU and GPU resources of the device but also by directly implementing particular chips, also called coprocessors [160], that are designed for executing the most recurring machine learning operations. For example, Google, starting with the Pixel 4 smartphone, implemented the Pixel Neural Core, a dedicated core for efficiently performing image-related machine learning (ML) tasks. Indeed, the most common applications for having Deep Neural Networks (DNN) directly running in the device regard [161]: photo improvement [162], activity recognition and tracking [163], [164], image classification [165], face recognition [166], real-time object recognition especially for supporting AR (Augmented Reality) applications [167] (like for example the landmark recognition). All of these applications can now easily run on smartphones, even with very low processing latency but this strategy has a clear downside. Indeed, running a DNN requires a non-negligible number of operations (that are often referred as FLOPs – floating-point operations), as a consequence, we expect a significant impact on the power consumption, that is a critical aspect for mobile applications. The approach for reducing this problem is to compress the DNN model [168], [169] by pruning the network, decomposing layers or use quantised weights for the model. Obviously, by losing information and model complexity, we lose something in the accuracy of the neural network, but in some cases, this loss is so minimal that the performances of the network are still acceptable.

The second approach to solve the problem is the offloading [154], namely making the device to execute the inference on an external server and then locally parsing the results. Task offloading consists of delegating part of the image processing to a remote server. The convenience of this technique presupposes three conditions: (1) the energy cost of offloading operations at the device is less than the energy cost required if the delegated calculation is performed on-board; (2) the response time and more generally the processing latency should guarantee at least the frame rate measured when processing is all local, and does not produce a lag in the rendering of the video; (3) the image processing precision metrics, e.g., accuracy, does not deteriorate.

For example, if we use a 'classic' cloud provider we can obtain very low inference latencies, but the network latency can reach values of 100ms or more, and therefore this cannot be suitable for performing VR applications. In addition, some AR applications require shared virtual objects (also called "cloud anchors") that must be updated in real-time by multiple users, which make the adoption of a

low latency and a high throughput cloud service mandatory. In these application scenarios, Edge/Fog computing can provide a solution[1].

In this Section, we report the results of the measurements of some experiments conducted on real mobile devices and using state-of-the-art and open-source video and image processing libraries, with the aim of concretely verifying what are the energy consumption and processing capabilities measured in frames per second. The useful use of this study lies in acquiring concrete data that can be used as criteria of choice in the design of offloading algorithms.

In this study, we investigated the energy compromise that a deep learning task for object recognition imposes to a mobile device and how much we can gain in terms of power consumption when the task is completely offloaded to a backend server equipped with a good GPU as well as the achievable frame rate.

For doing this, we set up an environment by using the most common libraries that allow doing inference with CNNs, like TensorFlow[2], OpenCV[3] and Darknet[4], and a set of pre-trained networks. From this, by using a sample video, we run the object recognition frame by frame, testing different set-ups, libraries and neural networks by using a custom Android application (Figure 5.2.1) and a Python Flask backend. We have conducted several benchmarks for deeply understanding the energy impact of a neural network deployed in a mobile system but also for evaluating how the offloading can have a beneficial effect on power consumption though preserving inference latencies. As the main source for power consumption data, we used the values offered by the Android OS that have been cross-checked by using a USB power meter.

The main contributions of this study can be summarised as follows:

- we present which are the strategies for implementing mobile object recognition task and for offloading it to a possible Edge device by using publicly available open-source frameworks and tools;
- we measure and analyse which is the energy impact of a mobile neural network directly running in a mobile smartphone and the consumption gain that we can obtain when the neural network recognition task is offloaded to an Edge device;
- we quantify the beneficial effect of the task offloading.

5.2.1 Background

The main purpose of this study is to find out when and how offloading a deep neural network inference task is convenient, especially under a power consumption point of view. In particular, we focus on the object recognition, namely the task of recognising the highest number of objects present in a photo and classify them by also giving the coordinates of where they can be found. The most used pattern for classifying objects is the one of setting up a Convolutional Neural Network (CNN), that is a neural network which takes grid-like data as input, and instead of performing matrix multiplications for describing the interaction between neurons' input and output (like in the classic Artificial Neural

[1] Google is experimenting/suggesting solutions in this sense for its AR core SDK
[2] `https://www.tensorflow.org`
[3] `https://opencv.org`
[4] `https://pjreddie.com/darknet/`

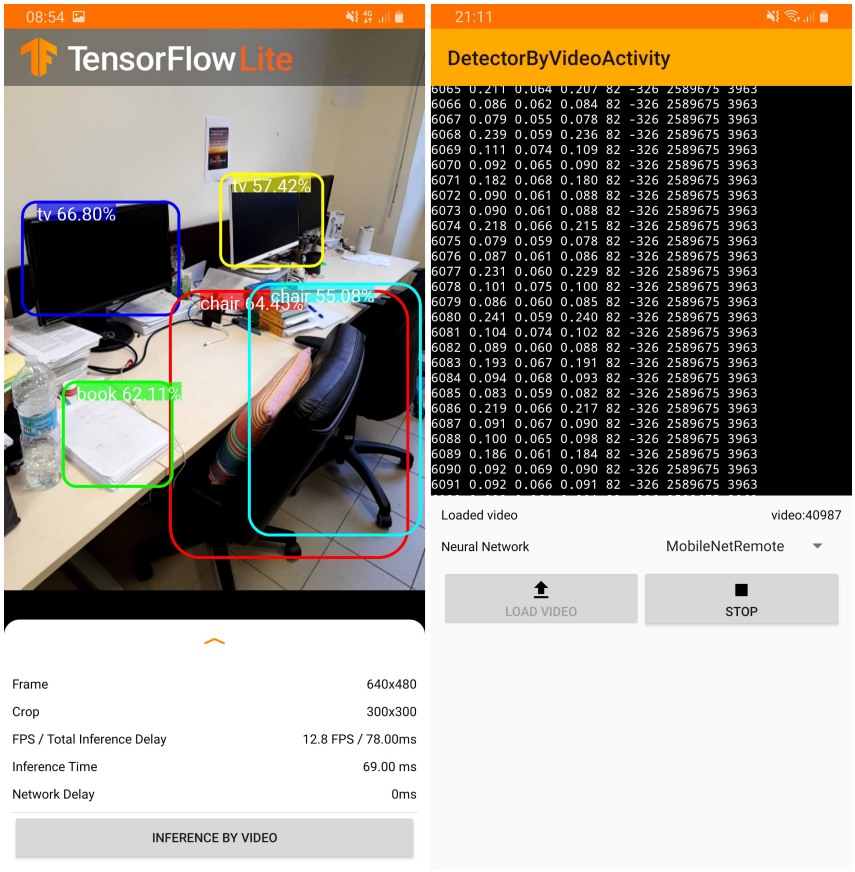


Figure 5.2.1: TensorFlow Lite Application for mobile object recognition.

Network model), they use the convolution operation [146]. The convolution has particular properties that make CNNs very successful with images.

When a CNN is used for classification, the final neurons layer that is used is the *fully connected layer*, which has a number of neurons equal to the number of categories that we are classifying. Instead, performing object detection requires that the final layer is generally replaced with a *detection network*, namely another set of *hidden layers* which are able to localize and classify objects inside the photography given as input the features extracted by the CNN. The two major examples of detection networks are Fast R-CNN [170] and SSD [171]. This is one approach to the object detection, but it is not the only one, indeed CNNs like YOLO [172] do not use a final detection network, but they try to do all at once.

The performance of a neural network of the kind mentioned above is described by:

- the mean average precision (mAP), that is the mean average precision across all the categories of objects that the network can recognize;
- the FLOPS, as already mentioned, that is the number of operations that the network requires for generating the output.

5.2.2 Experimental Setup

The purpose of this study is, first of all, to set up a complete environment which allows recognising objects that are framed by the mobile camera. Once the environment is ready, as a first case we consider that the mobile acts autonomously, then the device will be "assisted" by a backend which is characterised by a medium-level GPU and offers a very low latency communication with the device. This is the common environment that is provided, for example, in a Fog/Edge infrastructure [173].

In my setup, we consider that we are offering a real-time application, and therefore the maximum inference latency per-frame that we can expect is $\approx 50\text{ms}$. If we consider that the device camera is able to provide 30fps, then we are allowing that the recognition does not fall under the 20fps limit. This is a relaxed limit since here we did not test the most advanced hardware, but it is still acceptable for a subset of AR applications.

In the tests that we conducted, we used two different libraries for setting up the task environment. The two environments make use of the following libraries for running a CNN: Tensorflow, OpenCV and Darknet.

5.2.2.1 Mobile

The first step for running a mobile object recognition task is to set up an application which is able to capture the camera frames and process them one by one. Once the frame has been captured, this must be passed to a neural network that tries to find the objects and then returns their class and their position within a box. According to the library that is used, there are different strategies that we can follow.

OpenCV The OpenCV library is an open-source library which implements a very high number of features that are specifically related to computer vision: image/video processing and machine learning tasks. The other core feature of the library is that it is available for almost any computing platform, and in particular the Android version comes with some pre-written solutions for capturing the camera frames and running a callback function for each of them. In particular, all the frames that pass when we are processing one frame are lost, since a new frame is processed only when the callback function returns.

The first version of the mobile application, used for running the experiments, has been built completely with OpenCV 4.2.0 (Figure 5.2.2). The DNN subpackage of the library allows loading the main model formats for neural networks[5] like TensorFlow, Caffe, Torch, ONNX and Darknet. In presented experiments, as shown in Figure 5.2.2, we run the *TinyYoloV3* [172] CNN a reduced version of the YOLO network. This first kind of experiment showed a big drawback of the OpenCV library: the lack of GPU support. The obtained values are in line with the ones obtained in other works [174].

The complete inference process that has been built is the following:

1. when a frame is captured by the camera the callback function `OnCameraFrame` is called by passing as a parameter the frame in the OpenCV `Mat` format;
2. the frame is converted to RGB (from RGBA if we use the OpenCV camera object or YUV if we

[5] `https://docs.opencv.org/master/d6/d0f/group__dnn.html`

use the native camera object), scaled to fit the neural network input size (for example 416x416 for TinyYOLOV3, or 300x300 for MobileNet) and set as input to the neural network;

3. the neural network is run with the `forward` command;
4. the output matrix is parsed and non-maximum suppression [175] is applied to the output boxes;
5. boxes are drawn in the input frame;

Figure 5.2.2: OpenCV object detection with YOLO

TensorFlow Lite TensorFlow Lite is a completely different library with respect to TensorFlow, it supports a reduced set of its features, but it is optimised for running with low-power devices like smartphones or Raspberry Pi[6]. These optimisations require that the TF models must be converted in a format that is readable by the library, the ".tflite" format. Differently from the OpenCV library, TensorFlow Lite natively allows using the device's GPU, thus allowing decreasing the inference time drastically.

The TensorFlow team provides many examples of how using their libraries, in particular for TensorFlow Lite there is an example application which implements the real-time object recognition by using MobileNet neural network [7] and the mobile camera. The main difference with the OpenCV solution is that the application offers two image layers, in the lower one the camera frames are directly displayed, even if we are doing inference on them, in the upper one only the object boxes are displayed; this

[6] `https://www.raspberrypi.org/`
[7] `https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/android`

strategy allows making the camera stream fluid during the inference process.

The second version of the experimental application has been built upon the example mentioned above, some core parts have been rewritten to implement a benchmark process that takes as input a video file and allows analysing it frame by frame. Figure 5.2.1 shows two screenshots of the application. On the left, there is the real-time object recognition with the essential parameters displayed on the screen, and on the right, there is the activity which we implemented for loading videos. As we will see in Section 5.2.3, for every frame we log the frame number, the inference and network latency, the battery percentage, the instantaneous current (mA), residual battery capacity (mAh) and the battery voltage (mV). The inference process is precisely equal to the OpenCV version with the only difference that the inference result is not directly drawn to the frame but is passed to a layer that draws boxes on top of the camera frame. Within the application, the neural network that is implemented is MobileNet [149], which is a class of CNN specifically designed for mobile and embedded devices deployment.

5.2.2.2 Edge

We implemented the object recognition service by firstly using TensorFlow and then Darknet by wrapping them with the Python Flask[8] library. Finally, we attached the mobile device to a Wi-Fi hotspot created from the PC, thus simulating an Edge offloading infrastructure. The working paradigm is the following:

1. the device captures the frame and scales it to match the neural network input size – in this phase, the mobile also choose the compression level of the image, that is a critical parameter since it determines the network latency;
2. the Edge device that already loaded the neural network performs the inference and returns the result as a response;
3. the mobile device visualizes the results on the screen;

To natively exploit the GPU, before installing any neural network library, we need to obtain the CUDA toolkit and the cuDNN SDK manually. In the experiments, with the latest nVidia drivers 440.33, we used the CUDA toolkit 10.1 and the cuDNN 7.6.5 (compatible with TensorFlow 2.1[9]).

TensorFlow TensorFlow is an open-source library for performing machine learning tasks. As in the mobile case, we used neural pre-trained networks model available at the TensorFlow website[10].

Darknet We used Darknet [176] to run the YOLO neural network. Darknet is a neural network runtime environment written in C, and it is from the same authors of YOLO. The library must be compiled and then imported it in the Python web server script.

5.2.2.3 Equipment

In the experiments, we used a Samsung Galaxy Note8, a smartphone equipped with Exynos Octa 8895 @ 2.31Ghz processor, 6GB of RAM and a Mali G71 MP20 GPU with a computing capability of 374GFlops

[8] https://www.fullstackpython.com/flask.html

[9] https://www.tensorflow.org/install/source#tested_build_configurations

[10] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

and 29.80GB/s of memory bandwidth. The OS of the device is Android 9.0 Pie.

As Edge device which allows performing object detection, we used a PC with 16GB RAM, AMD FX-8350 processor and an nVidia GTX 1070 GPU with a computing capability of 5.73TFlops and a memory bandwidth of 256.3GB/s. All the neural network frameworks have been installed on Ubuntu 18.04 LTS.

5.2.3 Measurements and Results

The experiments have not been conducted by using the device camera but by analysing a video. The video that we used is a view of the Warsaw city from a car[11]. The original video has been converted from the resolution of 3840x2160 to 720x576, cut to 5 minutes (or 9000 frames) and then analysed in the device by using the JavaCV library[12], that binds together OpenCV and other media tools, like FFmpeg, the most common library for processing videos. The main reason for this conversion is due to the memory and the time required to load the video. During the video processing, as introduced in Section 5.2.2, we logged for every frame not only the timings but also the battery data that comes from the *BatteryManager* service of the Android OS. These values have been then cross-validated with a USB power meter, as shown in Figure 5.2.5. As far as regards the OS battery values, from the experiments emerged that they are not returned by every device, but every vendor decides whether log or not to log them. Specifically, the Samsung Galaxy Note8 device that we used for tests, the values do not change every time they are requested, but they are updated within a specific interval of time, in particular, the residual battery capacity is updated in multiple of 3.139mAh.

Results of the experiments are summarised in Table 5.1, which describes the environment used along with the timings and battery data. The neural networks tested are:

- *FakeNet*, that is a placeholder of processing frame by frame by doing nothing, this kind of test is done to understand which is the baseline consumption of the device;
- *MobileNet*, in the specific test we used a quantised MobileNet v1.0 with SSD for the TF Lite framework[13], and for the TensorFlow framework MobileNet v2.0 with SSD[14];
- *YOLOTinyV3*, that is a smaller version of the YOLO neural network [172][15].

All the neural networks that we used are pre-trained on the COCO dataset[16], a very large dataset of segmented images with 80 objects categories.

What emerges from the experiments, as we can see in Figure 5.2.3a and Figure 5.2.3b, is that running the object detection remotely allows to save about the 70% of the battery energy, in particular, about $45J$ are saved when the object recognition is offloaded to the Edge. Moreover, across the entire test, the recognition task has a constant energy consumption, this is justified by the fact that the number of objects recognised has not a great impact on the number of operation executed by the neural network, we assume that the oscillations are due to the underlying operating system of the device. As far as

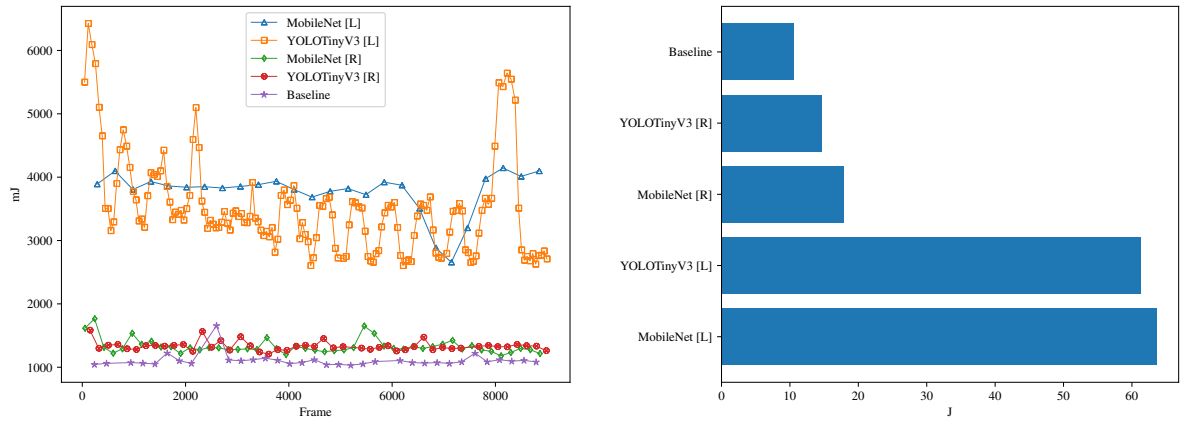
[11] `https://archive.org/details/0002201705192`
[12] `https://github.com/bytedeco/javacv`
[13] The codename of the network, available at the TF repository, is `coco_ssd_mobilenet_v1_1.0_quant_2018_06_29`
[14] The codename of the network, available at the TF repository, is `ssd_mobilenet_v1_coco_2018_01_28`
[15] https://pjreddie.com/darknet/yolo/
[16] `https://cocodataset.org`

		DNN Framework		*Average Time (ms)*			FPS	*Energy Consumption*	
	Neural Network	Mobile	Edge	Inference	Network	Total	Average	Instant (mA)	Cumulative (J)
Local	FakeNet	-	-	-	-	-	-	264.31	10.67
	MobileNet	TF Lite	-	46.1	-	46.1	21.70	918.32	63.65
	YOLOTinyV3	OpenCV	-	423.9	-	423.9	2.36	950.97	61.28
Remote	MobileNet	-	TensorFlow	42.1	25.6	67.7	14.80	330.45	17.91
	YOLOTinyV3	-	Darknet	21.9	27.0	48.9	20.44	360.25	14.70

Table 5.1: Summary of the main results of the experiments.

regards the inference latencies, in the tests, the remotely deployed TinyYOLOV3 allowed to reach about 20FPS, that is slightly less than the locally deployed MobileNet. These values are justified by two factor, first of all, the computational power of the nVidia GPU and, secondly, by the network latency that is in the order of 27ms – a value that depends on the specific Wi-Fi protocol used by the network adapter. In Figure 5.2.4 the behaviour of the inference latency is shown, and as we can observe the offloading scheme of the recognition framework allows for a more stable inference time, but this is essentially justified by Android operating system and all the background services that have been unpredictably activated during the test. What emerges is also that YOLOTinyV3 in remote reaches the same performances of the local MobileNet but with no impact on energy, this is a clear example of the beneficial effect of the offloading mechanism.

Summarising, offloading the object recognition has almost a zero-impact on the energy consumption, and in real Fog/Edge deployment with latest "Wi-Fi 6" [177] technology and more powerful GPUs could also allow reaching real-time inference latencies, i.e. 30FPS.



(a) Power consumption per-frame

(b) Total power consumption comparison

Figure 5.2.3: Power consumption during the experiment

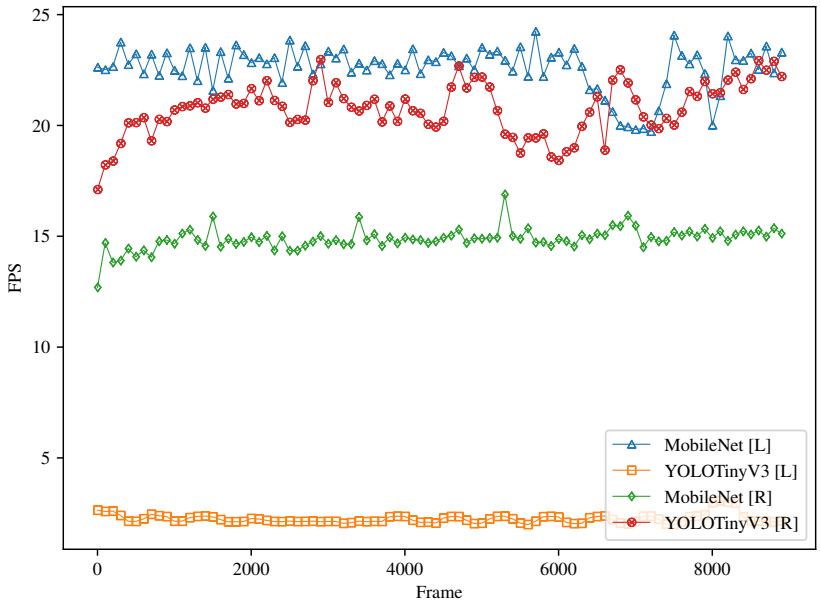


Figure 5.2.4: Inference latency behaviour in FPS.

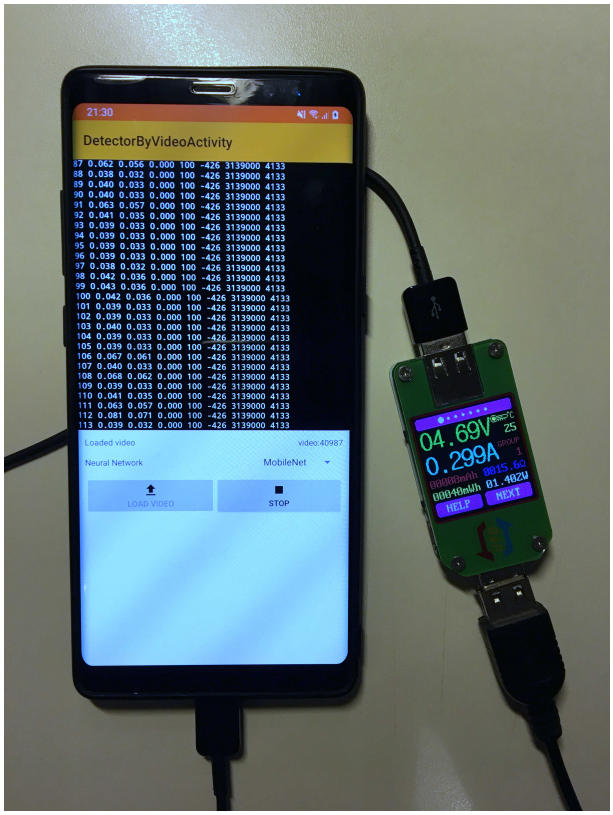


Figure 5.2.5: The USB power meter and the mobile device used for experiments

5.3 Energy-oriented load balancing for Green Edge computing

The amount of data generated by IoT devices is expected to increase exponentially over the next years. Edge computing [2] may alleviate the need to transfer data to a cloud server by performing data elaboration closer to the data sources. With the improvement in their performances, complex elaborations are also possible at the Edge. For example, accelerator-based single-board computers (SBCs) showing high performance are being used as Edge devices to run the inferencing part of the artificial intelligence (AI) model to deploy intelligent applications [178]. Besides being valuable for performance reasons, local elaboration is also the first step towards a reduction of pollution. Moving data to remote servers has, in fact, a non-negligible carbon footprint due to all the telecommunication equipment involved in data transmission and, of course, in the use of the data centres [179].

During the idle state, the energy consumed by an Edge device is one-third or less of the power needed when executing code. For example, NVIDIA Jetson Xavier NX consumes 15.2W when running object detection with YOLOv3, and only 3.6W when idle [178]. Another element to add to the picture is the cost reduction of the Wh [Watt per hour] of batteries and the increasing efficiency of photovoltaics (PV) panels, which makes it possible to design green Edge computing systems, that are energy self-sustainable (and to ensure service continuity this deployment can run side-by-side with classic grid powered architectures).

In this section, we focus on a use-case scenario where a set of off-grid PV supplied Edge nodes with their own camera implement a common service of image-based object detection, so that a node can delegate the inference done on an image acquired by its own camera, to another node in the set (thus relying on the task offloading strategy). This can be seen as a typical scenario of traffic monitoring in a smart city.

Concept and motivation The distributed nature of Edge/Fog resources makes it harder to exploit locally-produced green energy as more sites are considered [159].

In general, the amount of solar radiation converted by a PV panel into electrical energy depends, besides exogenous variables, e.g. weather conditions, on the orientation of the panels, and on the specific day of the year. It is reasonable to assume that due to mechanical or other constraints, the orientation of panels in this set is different, so that nodes have different green energy production. In addition, the state of these nodes (idle or working) may also differ.

Since the amount of storable energy is limited by the battery capacity, the solar energy at an Edge node with fully charged battery cannot be further accumulated, which is an indirect source of inefficiency. We call this not-accumulated energy, *energy surplus* loss. In order to consume as much green energy as possible, it is worth it for an Edge node that is running out of green energy to offload image detection tasks to nodes with (almost) full battery at risk of energy surplus loss. This offload is globally energy efficient since the energy consumed to move (send) the image data from one node to another is usually less than to do computation locally.

The contribution of this study is (i) the characterisation of an off-grid green Edge computing model and (ii) an initial evaluation of the benefit of task green energy aware task offloading.

5.3.1 Background: the Solar geometry

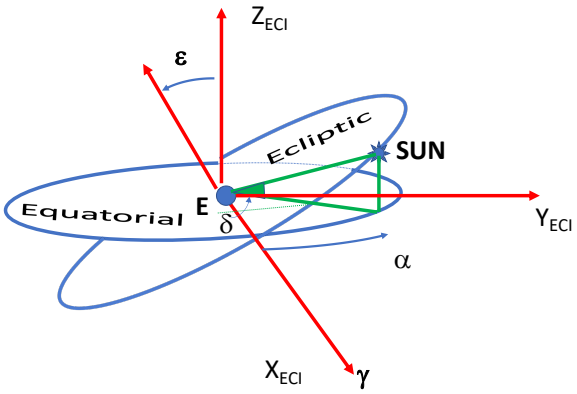


Figure 5.3.1: The position of the sun in the sky is identified through two angles: the Right Ascension (α) and the declination (δ). The movement of the sun is due to the earth rotation and revolution.

The apparent movement of the sun in the sky is due to two rotations: the earth's rotation around the earth's axis with a period of one sidereal day and the earth's revolution around the centre of gravity of the earth-sun system with a period of one sidereal year. The two rotations take place on two planes, respectively called equatorial and ecliptic planes, which are inclined by $\epsilon = 23.4°$. The intersection of the planes defines a line called the equinox line. To describe the motion, it is convenient to use a geocentric coordinate system (see Figure 5.3.1. We can use a simple model to derive the position of the sun $\mathbf{d}$, in which the orbit around the sun is circular, and the speed of the earth is constant. The position depends not only on the time but also on the geographic latitude of the earth and on the day of the year. The motion is firstly described in the so-called ECI frame and then transformed into the horizontal frame, which is used to orient solar panels.

The Earth-Centred-Inertial (ECI) frame has its origin at the centre of mass of the earth, the X axis on the equinox line pointing towards the vernal equinox (denoted as point γ, Z on the (mean) earth's rotation axis, $Y = Z \times X$[17]. the Earth-Centred-Earth Fixed (ECEF) frame that co-rotates with the earth. The ECEF and ECI frames are overlapped every sidereal day. The X and Y axis point towards longitude $0°$ (Greenwich meridian) and e $90°(east)$.

The position of the sun in the ECI frame can be expressed in Cartesian coordinates with following unit length vector:

$$\mathbf{d}_{ECI} = \begin{bmatrix} x_S \\ y_S \\ z_S \end{bmatrix} = \begin{bmatrix} cos\delta cos\alpha \\ cos\delta sin\alpha \\ sin\delta \end{bmatrix} \tag{5.1}$$

[17]Due to slight changes of the earth's axis rotation over time, it is common to adopt for the definition the orientation of the axis at a given date. This sync point is called an epoch. ECI J2000 refers to the orientation at the 12:00 Terrestrial Time on 1 January 2000.

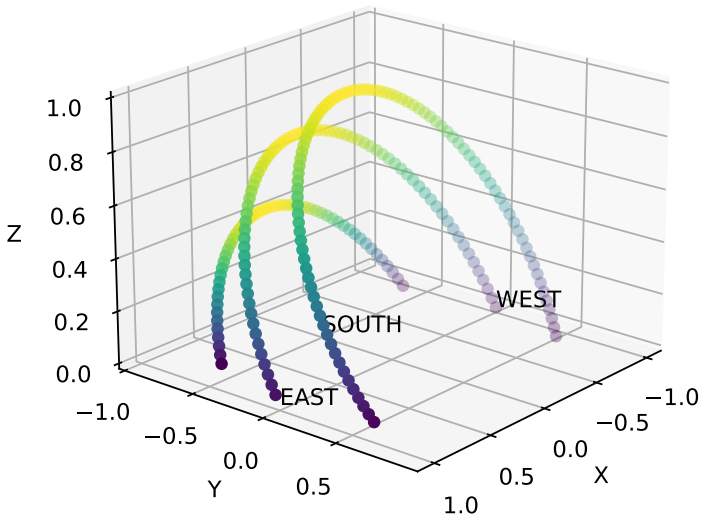


Figure 5.3.2: Example of sun path in the NEU frame at the equinox, winter and summer solstice, latitude $42°$ derived from Equation 5.6. At the equinox, the sun rises due east and sets due west. The duration of the day is the longest at summer solstice and the shortest at the winter solstice.

from which:

$$\alpha = tg^{-1}\left(\frac{y_S}{x_S}\right) \tag{5.2}$$

$$\delta = sin^{-1} z_S \tag{5.3}$$

where α and δ are two angles: the Right Ascension (RA), which is the angle from the vernal equinox measured along the equator and the declination angle with the equatorial plane, see Figure 5.3.1.

By definition, at vernal equinox (21 March), d_{ECI}=(1,0,0) so that $\alpha = 0$ and $\delta = 0$. A simple relationship between α and δ is the following. As the ecliptic plane is tilted of ϵ, its normal in the ECI plane is $\mathbf{n} = (0, -sin\epsilon, cos\epsilon)$. The points on the vertical line (parallel to Z_{ECI}) passing from the sun with given α and δ, have coordinates $\mathbf{p} = (cos\alpha, sin\alpha, tg\delta)$. Hence, the intersection with the ecliptic plane is the point $\mathbf{p} \cdot \mathbf{n} = 0$, so that the relationship between α and δ is:

$$\delta = tg^{-1}(tg\epsilon \times sin\alpha) \approx \epsilon \times sin\alpha \tag{5.4}$$

which the relationship in [180]. Days are numbered from January first and during day N the α is fixed:

$$\alpha(N) = \frac{360}{365}(N + \Delta_\gamma) \tag{5.5}$$

where $\Delta\gamma$ is the number of days from the last vernal equinox (Mach 20) from December 31, $\Delta\gamma = 286$. The value α is the right ascension of the sun during day N, which corresponds to the sun on the local meridian of the observer.

The NEU frame has the origin on the surface of the earth, the Z that points up to the sky towards

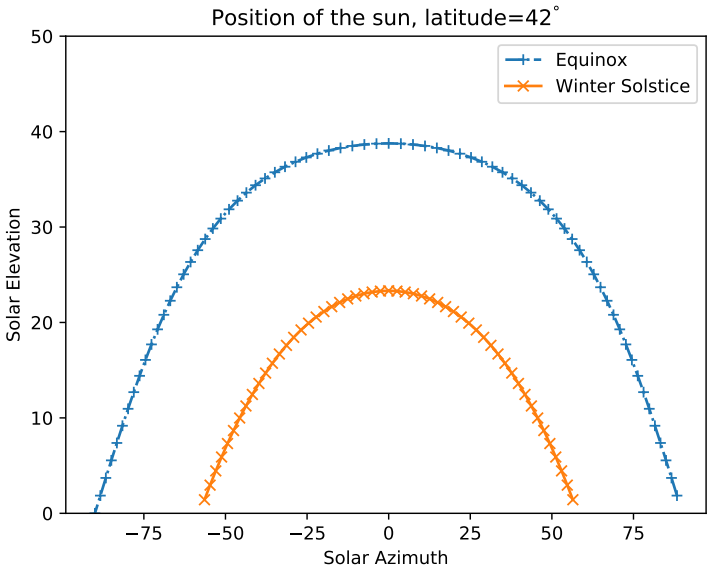


Figure 5.3.3: Example of solar diagram for the winter solstice and equinox.

the zenith (opposite to the plumb line), Y points towards the true North and X points towards East. The XY axis is tangent to the earth (assumed sphere). For this reason, the frame is also called the horizontal frame, and it is used to define the orientation of the panel on the earth.

The position of the sun $\mathbf{d}$ in the NEU frame is found through the following linear mapping:

$$\mathbf{d} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} c_{\beta_z} & -s_{\beta_z} & 0 \\ c_{\beta_x} s_{\beta_z} & c_{\beta_x} c_{\beta_z} & -s_{\beta_x} \\ s_{\beta_x} s_{\beta_z} & c_{\beta_z} s_{\beta_x} & c_{\beta_x} \end{pmatrix} \begin{pmatrix} cos\delta cos\alpha \\ cos\delta sin\alpha \\ sin\delta \end{pmatrix} \tag{5.6}$$

where $c_\beta = cos\beta$ and $s_\beta = sin\beta$, with $\beta_z = -\lambda - \pi/2 - \omega t$ and $\beta_x = \phi - \pi/2$. The meaning of these parameters are: $\omega = 360/24 = 15[^\circ/h]$ is the angular rotation speed of the earth, t is the time elapsed from when the ECI and ECEF frames were aligned, ϕ is the latitude and λ the longitude of the place on the earth. See [5] for details.

Horizontal	**ECI**	**ECEF**
Elevation (a) $[-90°, 90°]$	Declination (δ) $[-90°, 90°]$	Declination (δ) $[-90°, 90°]$
Azimuth (A) $[0, 24]$h clockwise+	right ascension (α) [0,24h] clockwise- from γ	hour angle (h) clockwise+ $[0, 24h]$ from south meridian

Table 5.2: Position angles in definitions in different reference frames.

5.3.2 Energy produced by solar panels

5.3.2.1 Panel orientation

The orientation of the panel is assigned via the normal to the surface of the panel $\mathbf{n}$, which is specified through a pair of angles: the altitude a - also called elevation or tilt angle, and the Azimuth (A). Elevation is measured from the horizon along the vertical, while the azimuth is positive clockwise. Instead of the altitude, it may be sometimes convenient to specify the zenith angle $\xi = 90 - a$.

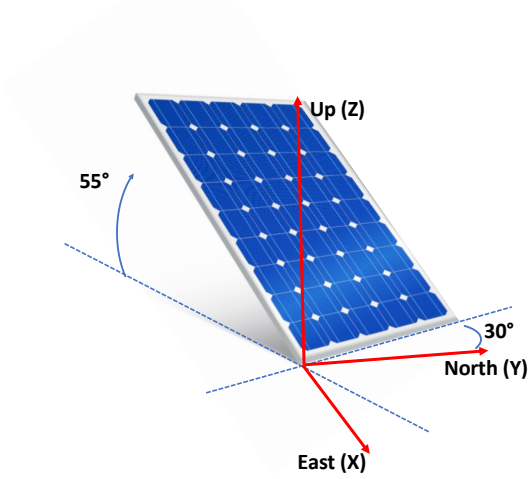


Figure 5.3.4: Example of panel orientated towards the west ($A = 330°$) and titled of $a = 55°$ (Creative Commons, authors unknown).

For example, a panel oriented towards west of $30°$ and tilt angle $a = 55°$ is specified as $A = -30°, a = 55°$ (see Figure). Clearly, the Cartesian coordinates of the normal are $\mathbf{n} = (cos\xi, sin\xi cosA, sin\xi sinA)$.

The global solar irradiance (energy per unit time and per unit area) hitting the surface of the PV panel is the sum of three contributions: direct bean, diffuse and reflected irradiance [181]. The amount of these contributions depends on three main elements: (i) panel orientation, (ii) the path of the sun in the sky and (iii) weather conditions. To keep the analysis simple and since we want to characterise the difference among nearby solar panels, we will focus only on the first contribution due to direct beam irradiation in clear days.

The value of solar irradiation at the mean distance of the earth from the sun on a surface normal to the sun is called the solar constant G_{sc}, and its current estimation is $G_{sc} = 1360 W/m^2$. The maximum value of the direct irradiation is approximately $1000 W/m^2$ at sea level on a clear day. With the current technologies, it is about 10% to $\approx 23\%$ of the solar energy is converted into electrical energy by a solar panel. We will then assume that the maximum power generated by the panels is $G_0 = 100 W/m^2$.

5.3.2.2 Energy production

The power produced at a given time by a unit square surface can be computed as:

$$G_f = G_0 cos\Theta = G_0(\mathbf{d} \cdot \mathbf{n}) \tag{5.7}$$

where Θ is the angle between the position of the sun in the sky (and hence the direction of incidence ray on the surface of the PV panel) **d**, and the normal to the surface of the panel **n**. The energy that recharges the battery during an infinitesimal time interval dt is:

$$dE_C(t) = \begin{cases} G_f dt & b(t) < B \\ 0 & b(t) = B \end{cases} \tag{5.8}$$

where $b(t)$ is the battery level at time t and B the battery capacity. The infinitesimal energy discharged from the battery is:

$$dE_D(t) = \begin{cases} P(t)dt & b(t) > 0 \\ 0 & b(t) = 0 \end{cases} \tag{5.9}$$

where $P(t)$ is the power required by the attached operating Edge node. The energy stored int the battery at time t is:

$$b(t) = \int dE_C(t) - dE_D(t) \tag{5.10}$$

The amount of energy produced over a period t_0, t_1 by the panel that cannot be stored nor used is called the *energy* loss, E_L. This quantity can be computed as the difference of the energy entering the battery of infinity capacity and into a battery of finite capacity is:

$$E_L = \int_{t_0}^{t_1} G_f(t)dt - \int_{t_0}^{t_1} P(t)dt - \int_{t_0}^{t_1} dE_C(t) \tag{5.11}$$

5.3.3 Baseline results

Parameter	Symbol	Range
Idle power consumption	P_I	2.5 W
Working power consumption	P_W	7.0 W
Power required to send tasks	P_S	0.5 W
PV power generation	P	25 W, 35W
Battery capacity	B	80 Wh,120 Wh
Time interval with no sun	T_{NS}	

Table 5.3: Parameters definition and value ranges

We report here some general results useful to motivate the proposal. The main parameters and their range are reported in table 5.3.

5.3.3.1 Charge-discharging cycles

The energy accumulated in a battery of a single Edge station follows a charging-discharging cycle that depends on the generated/consumed energy and battery capacity. A well-defined system should be able to fully charge the battery in a sunny day and supply the device at its maximum computation

speed starting from a fully charged battery, i.e. $B \geq E > P_W \times T_{NS}$. Figure 5.3.5 shows $b(t)$ from Equation 5.10 for two days.

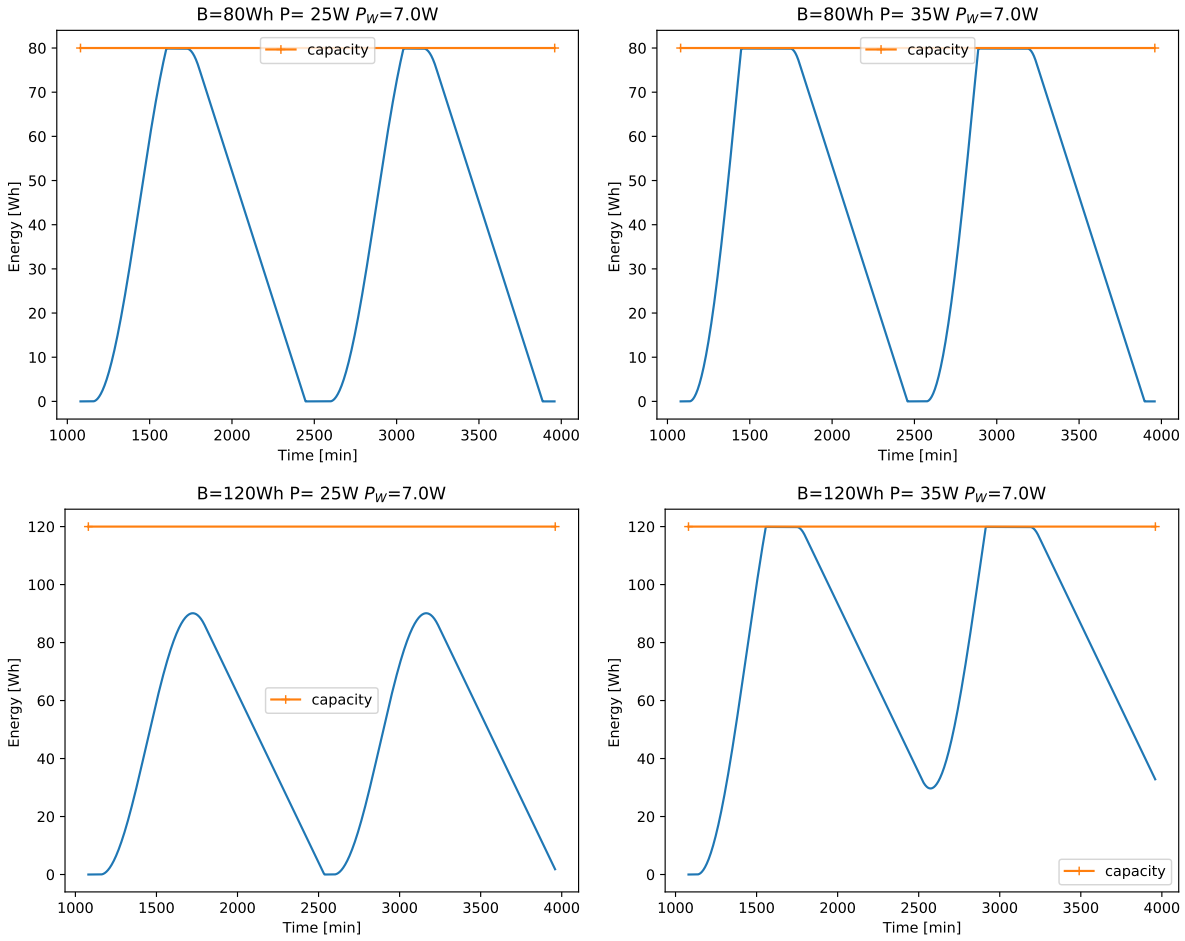


Figure 5.3.5: Example charge-discharge cycles over two days. Poor design due to insufficient battery (upper figures); insufficient PV performance (bottom left). The bottom rightmost figures show a correct design that provides good behaviour.

5.3.3.2 Effect of PV orientation

Figure 5.3.6 shows how the tilt angle that produces the maximum power depends on the day. Rotation from the south (azimuth=180) towards the west (azimuth=200) shifts the time when the maximum power is generated and pushes production more towards sunset time.

5.3.4 Seeker-Giver: an algorithm for green cooperation.

The idea of the algorithm is based on the observation that since the system is not connected to a grid when a battery is fully charged, further energy coming from the PV is not utilised (green *energy loss*). The proposed algorithm assumes a set of Edge nodes connected in a full mesh via a wireless communication channel. The algorithm divides the operating period of nodes into time slots of equal

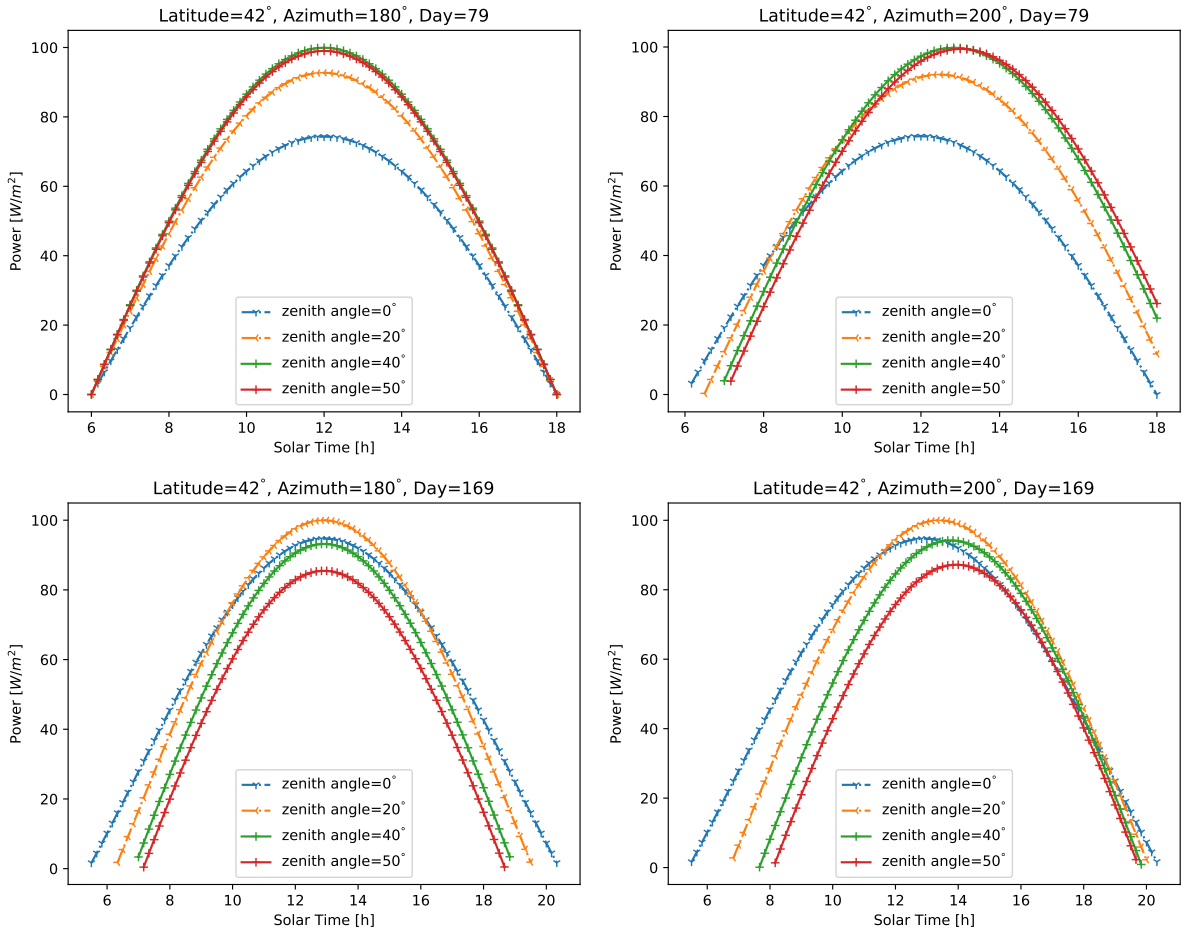


Figure 5.3.6: Example of power generated during the equinox (day 79), and summer solstice (day 169), for different elevation angles.

length, say of some minutes indexed as $k = 0, 1, \ldots$. Let e_i^k be the accumulated energy at the end of time slot k, expressed as a percentage of the capacity B. If the accumulated energy at node i is higher than a threshold value, say $e_i^k \geq T_1$ the node announces itself as a surplus or energy *giver* node, while the node whose energy is $e_n^k \leq T_2$ is called an energy *seeker* node. The nodes form two sets, named the Giver set GS and the Seeker Set ES. In order to match a giver with a seeker, nodes in the two sets are sorted according to their battery level to form two lists, ES' and EG'. The first node in the order list ES' is the one with the lowest stored energy, while the first in GS' has the highest accumulated value. Ties are broken based on ids. The value $n = min\{|ES|, |GS|\}$ is computed. Then the first node in the ES' set is matched with the first node in the GS' set, the second one with the second, and so on until n pairs are determined.

5.3.4.1 Results

We now present some preliminary numerical results of the above algorithm. Results are obtained numerically with time steps of 1 min at equinox day, sunny clear day, and latitude $\phi = 42°$.

Experiment 1 This experiment considers two nodes with same orientation, $A = 180°$ and $\xi = 42°$. Figure 5.3.7 shows the charge-discharge cycle among an energy giver and energy seeker node over four days. In this experiment, node 1 requires full power P_W for all days and eventually becomes a seeker, while the other node (node 0) is in idle state and it is a giver. The top graph shows how node 0 is subject to energy loss events, while node 1 basically discharges the whole battery. This means in the off-grid assumption that node 1 needs to move into a frozen state and be rebooted when energy is available again. The bottom graph shows the cycles for $T_1 = 0.5$ and $T_2 = 0.2$. Task offloading costs $0.5W$, i.e., when offloading the sender node (seeker) consumes $P_I + P_S$ while the giver node $P_W + P_S$. We can see how node 1 never runs out of energy. The difference in the slope of the energy demarcates the time when offloading occurs.

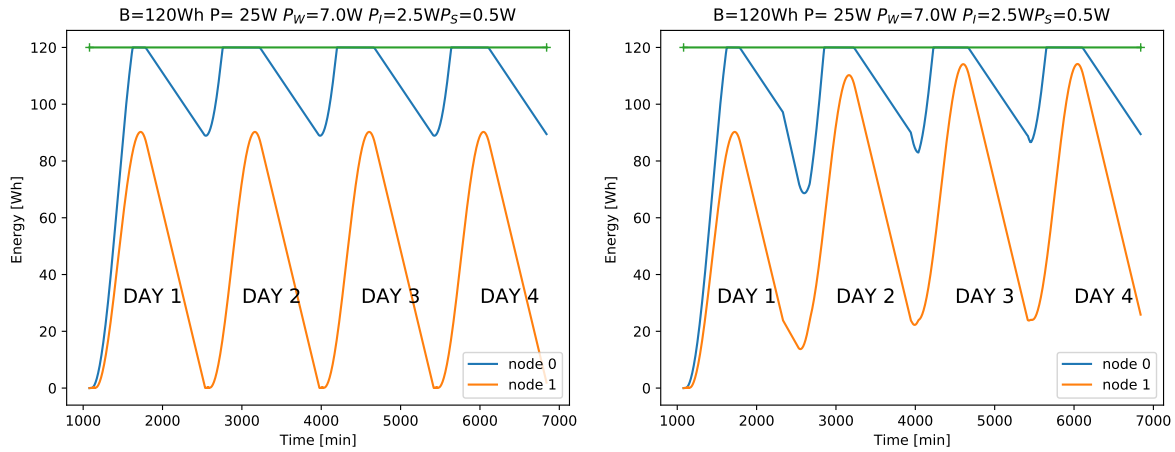


Figure 5.3.7: Example of algorithm with $T_1 = 0.5$ and $T_2 = 0.2$. No cooperation (left), with cooperation (right)

Experiment 2 In the second experiment we considered four Edge nodes with the following orientations: $\xi_1 = 20°, A_1 = 120°$,$\xi_2 = 35°, A_2 = 120°$,$\xi_3 = 35°, A_3 = 130°$,$\xi_1 = 42°, A_3 = 180°$. A trace over four days is reported in Figure 5.3.8. The first two nodes are idle all the days, while the other are working. Again, the cooperation shows how the working nodes never goes out of energy. The energy loss without cooperation was evaluated from Equation 5.11 to $E_L = 787.5Wh$, while under cooperation it was $E_L = 747.31Wh$. This reduction is due to offloading.

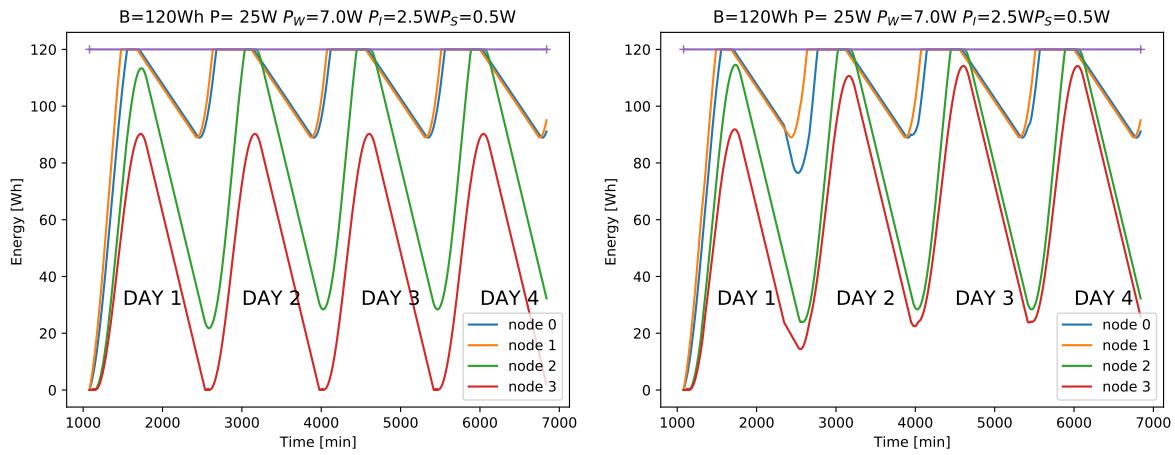


Figure 5.3.8: Example of algorithm with four nodes having slightly different PV orientation, with $T_1 = 0.5$ and $T_2 = 0.2$. No cooperation (left), with cooperation (right).

Chapter 6

Infrastuctural Studies

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> C.A.R. Hoare

While the previous Chapters focused essentially on system modelling and algorithm design and development, in this chapter instead, the approach is studying the infrastructural conditions, requirements and software which are needed for running the scheduling and load balancing algorithms. In particular, in Section 6.2, we formulate a series of conditions which are needed for building an unattended cluster of SBCs (Single Board Computers), moreover, we provide a hardware solution for exploiting a standard desktop PC case and the annexed ATX power supply. Instead, in Section 6.3 we illustrate the design of a framework, called "P2PFaaS", that allows the implementation of the aforementioned algorithms by relying on the Function-as-a-Service (FaaS) task model. The framework, based on Docker containers, has been published as open source.

The solutions and the concepts presented in Sections 6.2 and 6.3 have been published respectively in [12] and [3].

6.1 Related Work

Raspberry Pi and SBCs (Single Board Computers) clusters There are different approaches in literature for building a Raspberry Pi cluster, indeed the low realisation and operational costs make it usable for learning parallel computing [182], [183], running distributed algorithms [184], [185] or just for studying the energy and the computation power system [186]–[188].

In particular, [182] and [183] describe a Raspberry Pi cluster that is built for learning purposes. The devices are arranged in a cart and they make use of the MPI messaging protocol for intercommunication. However, this solution is not self-contained and it does not provide a practical solution to power management.

Works [189] and [184] show an implementation of a cluster of Raspberry Pi that is assembled with Lego bricks, the former also introduces a software management system called PiCloud and the latter uses the cluster for running a tourism data aggregation application. In both cases, a self-enclosing and an unattended design and structure are not considered.

"Iridis-Pi" is a cluster of 64 Raspberry Pi presented in [190]. The authors perform a benchmark of the cluster trying to derive the total computing power of the entire architecture, but for doing that different issues are addressed, like the power supply, the network capability and shared and distributed storage.

A series of challenges when building a Raspberry Pi cluster is listed in [191]. The paper, after describing the design and the setup of the cluster, performs a series of benchmarks regarding the total computational power of the system.

Concluding, [186] studies a Raspberry Pi cluster as a high-performance computing (HPC) cluster, considering the computing and the electric power the work provides performance results regarding the number of cores and the number of computing nodes in a cluster.

Frameworks for Fog and Edge Computing The idea of constructing a framework for Fog or Edge computing is quite well addressed in literature. Indeed, similarly to the work presented in this chapter, [129] proposes a platform for performing online machine learning with IoT data streams by leveraging Kubernetes for managing the containers that compose the framework. However, the operating model studied in this chapter is different since the focus is on the scheduling of FaaS execution requests and it is done in an online manner. Then, OpenFaaS [192] is an open-source software framework which allows to easily implement FaaS functions but the software does not allow the customisation of the internal scheduler, which is left to the underlying Kubernetes framework. From this framework, only the FaaS creation process has been taken into consideration. In [193], the authors propose an extension of the OpenFaaS framework addressing the scheduling of the task in nodes that are distributed geographically, however, the work focuses on the scheduling of the services and not of the single tasks and the approach is more oriented to the Cloud computing environment than the Fog or the Edge one.

Finally, other works instead are still focused on the implementation of scheduling and load balancing algorithms but differently from this study, they only simulate the computing nodes, these simulators are iFogSim [194], FogWorkflowSim [195], YAFS [196], xFogSim [197] and FogNetSim++ [198]. Simulations can have advantages during the design of the algorithm but do not consider real environments' issues

and parameters. Indeed, with the proposed framework, researchers can assess the efficacy of the algorithms in real environments.

6.2 Raspberry Pi testbed design

A crucial aspect of designing distributed algorithms for Fog or Edge computing is to provide results that efficiently run, aside from a simulation, in a real setup with real hardware and software. Most of the time trying to set up such kind of testbed could appear costly and time-consuming, but it is not always the case. Indeed, thanks to the current hardware technologies, there are many types of single-board computers (SBCs) which have a non-negligible computing power and they are available at a very low cost, like, for example Raspberry Pi[1]. Building a cluster with this kind of device is reasonable but if we want to build a stable, long-term, and stand-alone solution there are different issues that must be addressed: (i) decide a proper enclosure or chassis which will hold all the components of the cluster, and this must bring with it a suitable solution for managing the power supply of the entire system since it can be unfeasible to have a single power adapter for every single-board PC; (ii) the entire cluster should be managed, under the hardware point of view, like a server rack component which can be easily added or removed for example from a server cabinet; (iii) the ability to power on and off the cluster remotely, and this feature introduces an entire new set of problems that regards software management; (iv) the system is to be unattended or there is a low probability of the intervention of an operator who manually has to remove the single-board PCs from the cluster enclosure and reinstall the operating system, indeed in this kind of low power devices, the software resides on external storage, like a microSD, therefore installing the operating system requires the software to be flashed in the storage support. All these needs are the prerequisite to set up a general cluster that can be used remotely to run experiments, i.e., a concept that we dubbed *Testbed as-a-Service*.

This study is the result of an experience done to realise the envisioned cluster by using Raspberry Pi SBCs. The main contributions of this section can be summarised as follows:

- delineation of hardware and software requirements for a long-term, unattended and remote controllable solution for implementing a Raspberry Pi cluster;
- design of a power supply board for using a desktop computer power supply (called ATX) for powering up to eight Raspberry Pi boards;
- design of a remote Ethernet switch system for remote controlling the power of the cluster to be associated with the power supply board;
- propose a way to define the testbed configuration and an experiment via JSON configuration files, towards a Testbed-as-a-Service paradigm;
- show the results of the benchmark of a distributed scheduling algorithm installed in the cluster.

6.2.1 Hardware

6.2.1.1 Enclosure

The first decision that is needed to take when building the cluster regards a suitable physical structure that is able to hold the essential components, namely the SBCs and the power supply unit. There are many solutions that can satisfy our needs but they are often offered at a very high cost, relatively to

[1] `https://www.raspberrypi.org/`

the cost of the boards. For this reason, in this study, we tried to re-use a desktop PC case and its power supply unit. Figure 6.2.1 shows a preliminary set up of the cluster enclosure. We can observe that the single Raspberry Pis are arranged in (black) cases which have a single screw on the side, then all of these are attached to a rigid plastic structure that fits the standard holes of a Standard-ATX motherboard, which are specified by Intel [199]. The case has also been enriched with two fans for favouring the airflow.

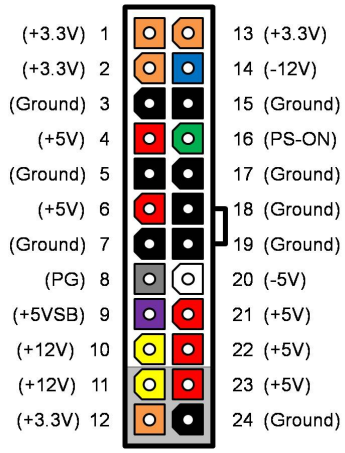
The case that has been used is a standard ATX case but it is not rack-able (i.e. it cannot be arranged in a server rack). For better space management, there are also available rack-able ATX cases of 4 units that could be easily installed in a server cabinet. However, switching to this kind of case does not alter anything of the study presented here.

Figure 6.2.1: Preliminary set up of the cluster

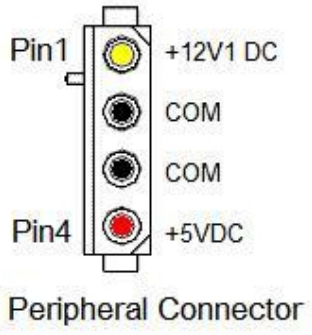
6.2.1.2 Power board design

The main issue of using a PC case as an enclosure is the usage of the ATX power supply unit, this because having eight power adapters for eight Raspberry Pis is not a feasible solution, especially in terms of space. A standard ATX power supply has different connectors that are usually attached to the motherboard and to all the peripherals. There are many vendors and types (we used a "Trustech TR-20787", whose specifications are listed in Table 6.1) but in general all of the connectors are cascade arranged in at least in 4 lines, in the following way:

L1) this line has one 24-pin port (Figure 6.2.2a) for the motherboard power;



(a) ATX 24pin motherboard connector



(b) 4pin peripheral connector

Figure 6.2.2: Desktop PC power supply unit connectors

L2) this line has two 4-pin ports for powering the peripherals (Figure 6.2.2b) and two ports for powering SATA disks;

L3) same as L2);

L4) this line has one 4 (or 4+4) pins port powering the CPU.

All of these lines are made up of 20 AWG cables (which corresponds to 0.50mm^2 of section) and they have 20 cores. In normal conditions, each of these cables can bring up to 3.5A[2], considering 5V that is the voltage needed for the Raspberry Pi. Moreover, a Raspberry Pi 4 needs 3A for operating correctly[3], this means that for powering up to 8 Raspberry Pi 4 we need 24A and 7 cables from the +5V rail of the PSU (whose colour is red) and at least 7 cables for the negative pole (whose colour is black). We can gather all the needed cables from the PSU connectors, in particular we can:

- pick 5 red +5V cables and 8 black negative cables from the 24pin connector;
- pick 2 red +5V cables and 2 black negative cables from the two lines in which we have the peripheral 4pin connector.

This configuration would allow powering up 8 Raspberry Pi since the PSU can support at maximum 28A on the +5V rail (Table 6.1).

Figure 6.2.3 shows the circuit diagram of "ATX2RPi8", a printed circuit board (PCB) that we designed for powering eight Raspberry Pi using a desktop ATX power supply unit. The red tracks are printed at the front face of the board and the blue ones instead to the rear face. The diagram has been designed with EasyEDA[4] and then submitted to the factory for printing[5]. Aside from gathering the red and black cables as described, also considering the thickness of the tracks proportional to the load, the board also has the following features:

[2] https://www.engineeringtoolbox.com/wire-gauges-d_419.html
[3] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/
[4] https://easyeda.com/
[5] https://jlcpcb.com/

Trustech TR-20787						
AC Input	**Voltage** 230Vac		**Current** 6A Max		**Frequency** 50Hz	
Max DC Output	+3.3V	+5V	+12V	-12V	-5V	+5VSB
	22A	28A	28A	0.8A	0.6A	2.5A

Table 6.1: The specification label on the PSU (Power Supply Unit)

- it distributes the 5V+ current to eight clamps, to which will be attached the cables towards the Raspberry Pis. These cables have been assembled with a Type C connector;
- it offers two fan ports, one at 5V and one at 12V;
- it offers a clamp for the PS_ON rail that if connected to ground cause the switching on of the PSU. This clamp will be attached to an ethernet switch for remotely turning on and off the cluster;
- it offers a clamp for the +5VSB rail, that is a line always powered, even if the PSU is turned off. This line is used for powering an ethernet relay;
- it offers an additional clamp for 12V, for general purposes.

Table 6.2 shows the Bill of Materials (BOM) of the board, it can be used for ordering the components from the supplier[6]. The components are: the 24pin female connector, two fan ports, two 4pin peripheral power connectors and eleven generic bipole clamps. The PCB printing and the components cost for a single board, including the shipping fees, is about 10$.

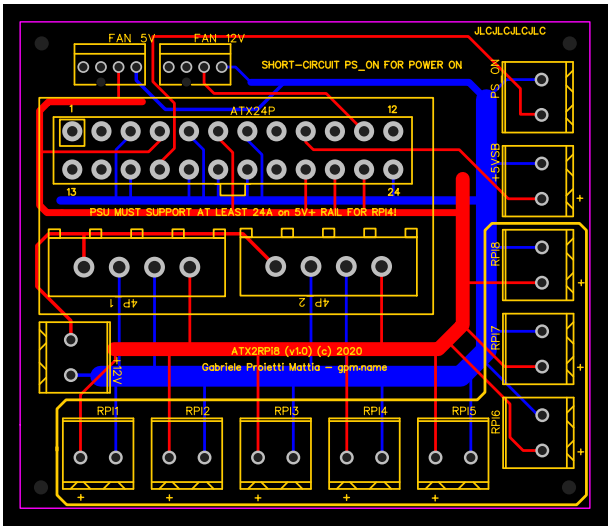


Figure 6.2.3: Power supply board (ATX2RPi) (82x70mm)

[6] https://lcsc.com

ID	Name	Designator	Quantity	Manufacturer Part	Manufacturer	Supplier	Supplier Part
1	CONN-TH_39281243	ATX24P	1	39281243	MOLEX	LCSC	C114088
2	CONN-TH_47053-1000	FAN_12V,FAN_5V	2	47053-1000	MOLEX	LCSC	C240840
3	CONN-TH_350211-1	4P_1,4P_2	2	350211-1	TE Connectivity	LCSC	C305826
4	CONN-TH_2P-P5.00_WJ500V-5.08-2P	RPI*	11	WJ500V-5.08-2P-14-00A	ReliaPro	LCSC	C8465

Table 6.2: BOM for ATX2RPi(8) board

6.2.1.3 Power control board

A relevant feature that an unattended cluster needs to have, especially if composed by Raspberry Pi, is the ability to remotely switching on and off the power in case there is the necessity of force restart the devices. This capability is included in the design of ATX2RPi8, but it must be completed with an ethernet or wireless relay which must be connected to the +5VSB port for being powered and to the PS_ON port for switching on and off the PSU.

In proposed set-up, we used a "HW-584 Web_Relay_Con V2.0" that is a relay control board and it can manage up to 16 channels. we only used one channel that has been connected to a high/low-level trigger relay, and the relay has been connected to the PS_ON port of the ATX2RPi8 board. The controller exposes a web dashboard from which we can switch on and off the relay and therefore the entire cluster.

6.2.1.4 Final Setup

Figure 6.2.4 shows the final configuration of the cluster that is currently operational. As we can observe, the cluster with eight Raspberry Pis 4B (with a quad-core Cortex-A72 CPU, 4GB of RAM and gigabit port) have been attached to a WiFi router (ASUS RT-AX88U, that supports up the 802.11ax standard) which allows experiments that make use of the wireless network, for example, smartphones or IoT. Then another two external Raspberry Pis 3B (with a quad-core Cortex-A53 CPU, 1GB of RAM and gigabit ethernet capped to 300Mbps due to the internal design of the RPi) have been added: the former is used as traffic generator which can simulate a background or noise traffic to the other SBCs, the latter is instead an entry point, since the cluster lives in a department network, it is necessary to have an SSH or VPN entry point from which we can have the control of all the components of the cluster, in this way we expose only a single node to the department's internal network and the cluster traffic remains confined in the subnetwork. In particular, this Raspberry Pi has been equipped with a USB ethernet (eth1), beyond the embedded ethernet port (eth0) in this way the RPi can be reachable from both the subnetworks.

6.2.2 Software

A crucial feature that a cluster of Raspberry Pi should have, regards the ability of easily deploy and un-deploy software when it is needed. This process could be the easiest and cleanest as possible, we cannot allow an uncontrolled installation of libraries, dependencies and configurations. Therefore the use of some container management system is essential. We chose to install within each Raspberry Pi board a software distribution that already has included Docker and cloud-init that is a utility used in cloud contexts for auto-configuring nodes at boot. This distribution is open-source and it is called

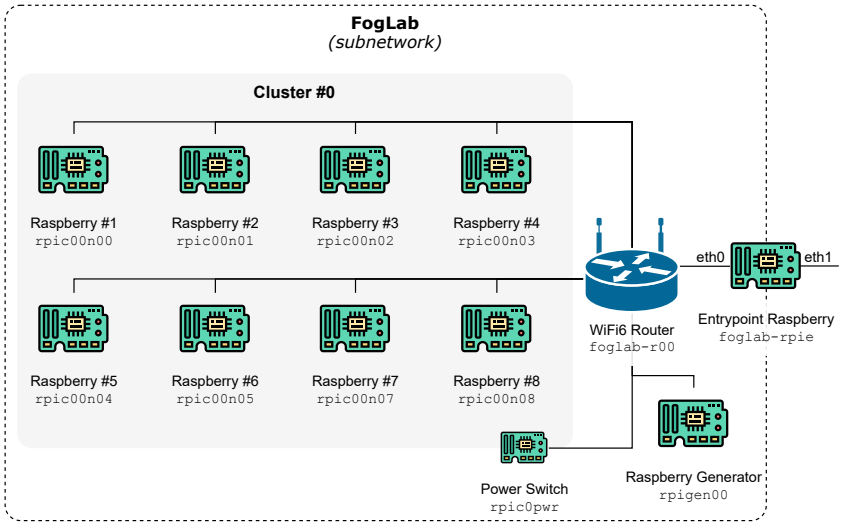


Figure 6.2.4: Final set up of the cluster

Hypriot OS[7]. But we did not use the software as it is, we adapted it to the following guidelines that we defined to make the cluster unattended.

1. in order to limit a human operator intervention we need to drastically reduce the possibility of OS corruption, for this reason the root partition should be read-only while all the persistent data (e.g. the Docker images and containers data) should be moved to a new writable partition;
2. the Raspberry Pi should reboot automatically if the system hangs or freezes (for example due to a kernel panic) therefore the watchdog kernel module (which is natively supported by hardware backend in RPi) must be enabled;
3. every node must auto-configure itself at boot, in particular every node must be reachable with SSH without manually typing username and password, this for facilitating any set-up or benchmarking script.

These code changes are published as open-source[8]. The final OS has been built and installed to all the Raspberry Pis and configured to have fixed IPs under a WiFi6 router which completed the cluster deployment.

6.2.2.1 Testbed-as-a-Service

For executing experiments, we installed in the cluster a FaaS scheduling framework called P2PFaaS [13] that allows implementing distributed scheduling algorithms for the FaaS job unit model. Since the framework is fully configurable via REST API, we can envision a JSON configuration file which sets up the testbed environment, like the needed nodes, the topology, the chosen scheduling algorithms and other parameters. This JSON, which can be accompanied by another configuration file which regards the specific experiment parameters, should be taken as input to a hypothetical master node which is in charge to actuate the passed configuration constraints.

[7] `https://blog.hypriot.com/downloads/`
[8] `https://github.com/rpi-cluster`

Listing 1 shows the full description of a possible JSON file for describing the testbed environment. As we can see, the capabilities that are offered by the configuration regards:

1. the definition of the `infrastructure`, namely the number of nodes to start and their specific topology which is expressed in terms of neighbours nodes;
2. the configuration of the `scheduler` service envisioned as a Docker container; therefore we need to pass the address of the Docker image, the name of the scheduler that will be used (to choose among a set of schedulers implemented in the framework), the arguments of the scheduler and other basic parameters like the maximum number of parallel jobs that can be executed and the maximum job queue length;
3. the configuration of the `discovery` service, which is again a Docker container in charge of making nodes aware of their neighbours; here we could configure, for instance, the Docker image and the delay between the heartbeats;
4. the configuration of the `functions` that will be made available for testing, again envisioned as Docker containers; therefore we need to specify the Docker image address of the function, a name, the API address that will be call-able by clients and a set of specific fixed deployment arguments, if needed. The list of functions is expressed as a JSON array of JSON objects.

The proposed cluster is implementing a Fog environment, therefore we emphasize that the scheduler and the discovery service (and therefore the P2PFaaS framework) with the specified functions will be spawned in every node, since every node will be able to execute that functions by calling the respective API addresses and to schedule the execution in other neighbours nodes.

Listing 2 represents the configuration file for running an experiment. As in the previous case, this file should be passed to a hypothetical master node that is in charge of executing parallel flows of REST API calls to all the nodes in the cluster. The configuration file should allow to properly set:

- the `api` address of the function to test (that has previously configured in Listing 1);
- the `payload` path to associate to every REST API call;
- the path (`log_path`) to the log directory where the test results can be collected;
- the job `arrivals` configuration, that comprehends the rate (requests/s, also referred as λ) to every specific node and the distribution according to which the requests will be generated, for example as a Poisson distribution; for the sake of simplicity, we assume a fixed distribution but we could also envision to use here a distribution that comes from real user traffic, properly defined in a plain text file;
- the total number of request after that the experiment can stop, with field `max_num_requests`.

By having defined both the configuration files for the testbed itself and the experiment to carry out, we envision that the testbed usage can follow a Testbed-as-a-Service paradigm.

Listing 1 Testbed JSON configuration file

```
{
  "infrastructure": {
    "nr_nodes": "3",
    "topology": {
      "0": ["1","2"],
      "1": ["0","2"],
      "2": ["0","1"]
    }
  },
  "scheduler": {
    "image": "https://...",
    "name": "SchedulerIdentificator",
    "args": ["arg1", "arg2", "arg3" ],
    "max_parallel_jobs": 4,
    "max_queue_length": 2,
  },
  "discovery": {
    "image": "https://...",
    "heartbeat": "30s"
  },
  "functions": [
    {
      "name": "My Service",
      "api": "my_service",
      "image": "https://...",
      "args": ["arg1", "arg2", "arg3"]
    },
    {
      "name": "My Service #2",
      "api": "my_service_2",
      "image": "https://...",
      "args": ["arg1", "arg2", "arg3"]
    },
  ]
}
```

Listing 2 Testbed experiment JSON configuration file

```
{
  "api": "/my_service",
  "payload": "/path/to/payload",
  "log_path": "/path/to/log",
  "arrivals": {
    "distribution": "poisson",
    "rate": 1.0,
    "rates": {
      "0": 1.0,
      "1": 2.0,
      "2": 1.5,
    },
    "max_num_request": 2000,
  },
```

6.2.3 Experiments and Results

6.2.3.1 Experiments

We performed two experiments by using the presented cluster:

1. the first experiment has been run configuring the testbed to use a scheduler that requested no cooperation between nodes: upon a job arrival the job is always executed locally in the node, if there are available resources, otherwise it is rejected. Then we chose an arrivals scheme with no distribution behind, thus with fixed jobs interarrival time. This has been done for understanding the computational power of a single node in the cluster and to properly choose a reasonable arrival rate for the next experiment;
2. the second experiment is based on a power-of random choice cooperation algorithm for scheduling (called `PowerOfN` within the framework), in particular, the one presented in [13]. We executed a benchmark for eight different payload sizes.

For both the experiments, we tested the same function, namely a face detection task based on the PiCo algorithm[9]. The same image has been used for every job request: a 640x480 JPG with exactly four faces.

6.2.3.2 Results

Experiment 1 With this experiment, we collected the response time of 2000 requests sent in series. We obtained an average of $0.696s$ ($\sigma = 0.01848$) and the distribution depicted in Figure 6.2.5. This means that the service rate μ for a single node in the cluster is about 1.43 jobs/s, but this must be multiplied by the four processing cores of the CPU, therefore we estimated a total service rate of $1.43 \times 4 = 5.72$ jobs/s. This value can be considered as a good estimation because we assume that the kernel scheduler is fair and therefore four parallel jobs will be approximately scheduled to four different cores for the most of the execution time. This is also the reason why we set the P2PFaaS framework to be allowed to execute only four jobs in parallel ($K = 4$) in each node and in each benchmark.

Experiment 2 Listing 3 shows the testbed configuration file for this experiment. As we can observe, the topology has been declared as a `fullyConnected` graph, and we set the `PowerOfN` scheduler which takes as input: (i) the fanout, that is the number of random probed nodes, set to one; (ii) the threshold (Θ), that is the limit above which cooperation is started, set to two; (iii) if the job that cannot be executed is discarded and not put in a waiting queue, set to true; (iv) the maximum number of hops that job can perform before being executed, set to one. Then we also set the maximum number of parallel jobs to four, as four is the number of cores of the Raspberry Pi 4B, and we set to use no additional queue, since `max_queue_length` is set to zero. As far as regard the function, the parameters `input_mode` and `output_mode` that refers to the input and output type of the function are set to `image`, therefore the input will be a binary file representing a JPG image and the output a binary file that is the same image with the highlighted faces.

[9] `https://github.com/esimov/pigo`

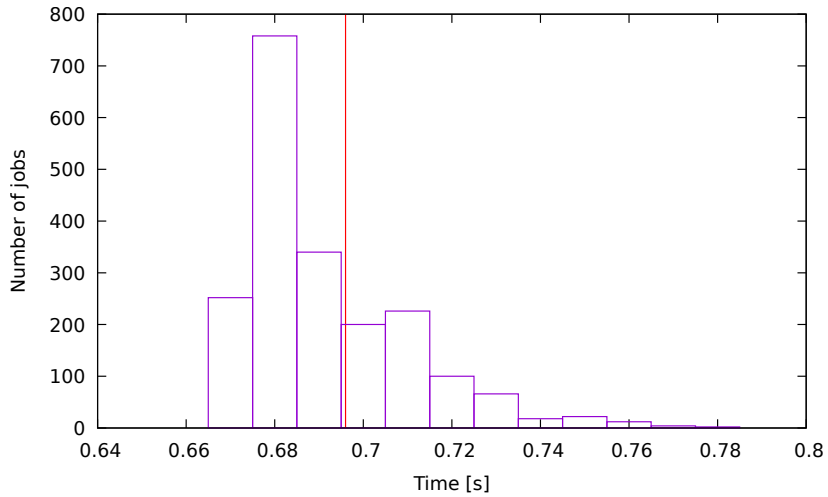


Figure 6.2.5: Job duration distribution for a single Raspberry Pi 4B with no cooperation scheme

Listing 3 Testbed JSON configuration file for Experiment 2.

```
{
  "infrastructure": {
    "nr_nodes": "6",
    "topology": "fullyConnected"
  },
  "scheduler": {
    "image": "p2pfaas/scheduler",
    "name": "PowerOfN",
    "args": [1, 2, true, 1],
    "max_parallel_jobs": 4,
    "max_queue_length": 0
  },
  "discovery": {
    "image": "p2pfaas/discovery",
    "heartbeat": "30s"
  },
  "functions": [
    {
      "name": "Pigo Face Detector",
      "api": "pigo-face-detector",
      "image": "esimov/pigo-openfaas",
      "args": {
        "input_mode": "image",
        "output_mode": "image",
        "write_timeout": 100,
        "read_timeout": 100
      }
    }
  ]
}
```

Metrics collected and analysed during the benchmarks are of two types. The first set of metrics regards the jobs' execution, and it has been collected by using the data reported by P2PFaaS. We have:

- *drop rate* (P_B), the percentage of jobs that have been rejected because they find the node to which they have been assigned at full load ($K = 4$);
- *total delay* (d_t), the total elapsed time for completing the request as seen by the client, in the present case the Raspberry Pi Generator;
- *probing delay* (d_p), the total elapsed time for asking another node its current load: it comprises the time for transmitting the request and for receiving the response;
- *forwarding delay* (d_f), that is the total time required for transmitting the job (only a few bytes of metadata) and its payload to another node;
- τ_e that is the time between the decision to forward a job and the effective job arrival in the remote node and it is defined as $d_f + d_p/2$.

The second set of metrics regards the nodes operating system and have been collected using Telegraf[10] and InfluxDB[11]. They comprise:

- *network activity*, the bytes received and sent by the network adapter every second;
- *CPU load*, the CPU time used by the system, the user and for serving the interrupt requests (Soft IRQs);
- *system load*, the average load of the system as reported by Linux in the last 1, 5 and 15 minutes.

In this experiment, we tested different payload sizes but we used the same image in order to not change the computational time required to detect the faces. We indeed appended spare bits at the end of the payload to reach desired payload sizes. The following results focus to nine different payloads, from 50kB ($\tau_e = 0.021$s), the original size of the image, to 800kB ($\tau_e = 0.135$s) the maximum payload supported by the Raspberry Pi Generator (we experimented that a payload greater than this limit saturates the network adapter queue). Each test for each payload involved 2000 requests sent with Poisson distribution and has been repeated five times. Confidence intervals that are shown have been computed as $\bar{X} \pm t_{\frac{\alpha}{2},n-1}\frac{S}{\sqrt{n}}$ (where $\bar{X}$ is the sample mean, S the sample variance, and t the Student-t distribution) with $\alpha = 0.05$. All the sample means of the experiments are reported in table 6.3. Listing 4 shows the JSON configuration file for this experiment, notice that we need a configuration for each payload to use, for this reason the payload path is set as `payload-Xkb.jpg`.

Figures 6.2.6a and 6.2.6b show the average drop rate and the delay when the threshold $\Theta = 2$ and the job arrival rate $\lambda = 5.50$ jobs/s as a function of τ_e. We can observe that the increase of the job payload, and thus of the network delay that exists between the forwarding decision and the actual arrival of the job to the destination node, causes a twofold effect: (i) an increase of the percentage of jobs that are dropped and (ii) a growth of the total delay. In particular, when we deal with the original image of about 50kB the drop rate is 55.16%, the total delay is 754ms while $\tau_e = 21$ms; when raising the payload up to 800 kB we observed an additional 10% in the drop rate, an increase of the total delay to 1.14s and of τ_e to 135ms.

[10] `https://github.com/influxdata/telegraf`
[11] `https://github.com/influxdata/influxdb`

Listing 4 Testbed experiment JSON configuration file for Experiment 2.

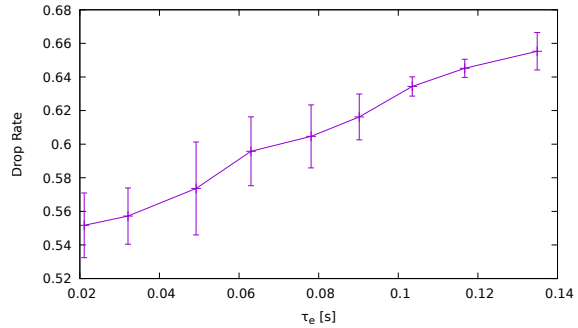
```
{
  "api": "/pigo-face-detector",
  "payload": "./payload-Xkb.jpg",
  "log_path": "./log",
  "arrivals": {
    "distribution": "poisson",
    "rate": 5.5,
    "max_num_requests": 2000,
  }
}
```

kB	P_B	d_t	d_f	d_p	τ_e
50	0.5517	0.7545	0.0174	0.0073	0.0211
100	0.5572	0.7751	0.0284	0.0075	0.0321
200	0.5736	0.8175	0.0455	0.0075	0.0492
300	0.5958	0.8740	0.0591	0.0078	0.0630
400	0.6046	0.9159	0.0744	0.0074	0.0781
500	0.6163	0.9486	0.0867	0.0072	0.0902
600	0.6344	1.0158	0.0998	0.0074	0.1035
700	0.6452	1.0712	0.1132	0.0070	0.1167
800	0.6553	1.1401	0.1314	0.0070	0.1349

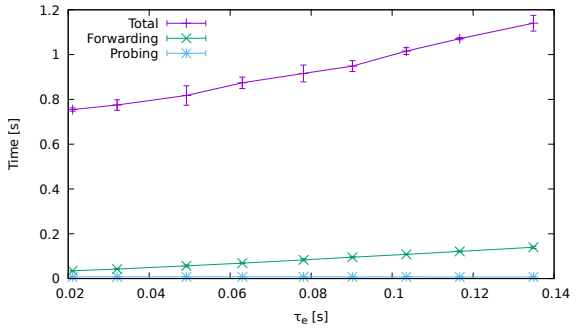
Table 6.3: Summary of experimental results ($\Theta = 2$, $\lambda = 5.50$) as a function of the payload size (kB), all times are expressed in seconds

The probing delay remains independent from the job payload, and it remains steady to about 7ms, and this reasonable since the probing does not use any relevant data transmission.

The linear relationship between the drop rate and the delay is reasonable and it shows the impact of the uncorrelation between the moment in which the node takes the forwarding decision and the moment in which the job actually arrives in the remote node



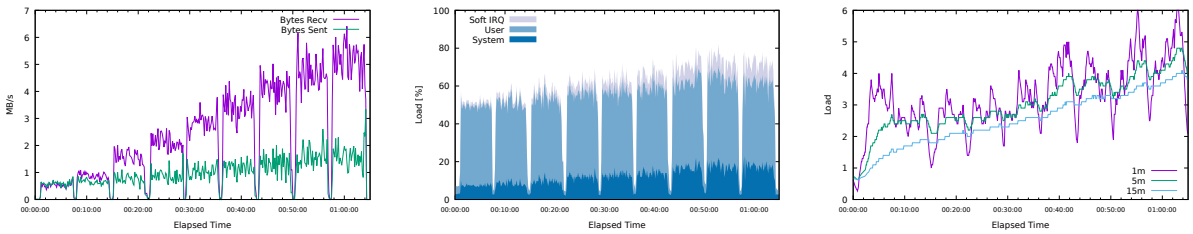
(a) Effect of τ_e on P_B ($\Theta = 2$, $\lambda = 5.50$)



(b) Effect of τ_e on d_t, d_f, d_p ($\Theta = 2$, $\lambda = 5.50$)

Figures 6.2.7a, 6.2.7b, and 6.2.7c shows Raspberry Pi 4B node OS statistics for a single benchmark from payload 50kB to 800kB with 2000 requests for payload size and using $\Theta = 2$ and $\lambda = 5.50$ images/s. In all of these figures, the x-axis represents the time elapsed during the benchmark, and we can notice a periodic fall of the y-axis values. Indeed, when switching the payload size, we observe a 60s gap during which the system is idle: this is voluntarily done to let the OS free all the resources and also to insert a clear recognition mark for the beginning each different test in the chart. analysing these results, we can observe the behaviour of network activity when the payload increases, and we can note that the bytes received and sent grow linearly with different slopes: this is justified by the fact that the bytes sent from the node regard only (i) the payload jobs that are forwarded and (ii) the response payload to the traffic generator (that is always the 50kB with the face highlighted independently from the payload of the request). Since we increase the payload while leaving constant the job arrival rate, the bytes sent rate also increases. In particular, we start with and an average of 0.5MB/s when the payload is 50kB to 1.5MB/s when the payload is 800kB. Focusing on the bytes received rate, we observe (iii) the payload of the jobs sent by the traffic generator and the (iv) payload of the job forwarded by other nodes: we start with 0.5MB/s when the payload is 50kB to 5MB/s when the payload is 800kB. The total network traffic estimation is reasonable considering nodes and router network capabilities.

We can conclude the analysis by observing the CPU usage (Figure 6.2.7b) and the system load (Figure 6.2.7c). In particular, we can note how the CPU usage in "user" space, which includes the job processing, remains constant during the experiment, while the IRQ processing and the system usage increases. This effect is explained because, by increasing the payload size, we require more packets to be received and to be sent: for this reason, there is more work to be done by the kernel with respect to the job processing. The system load, instead, reflects the overall load of the system, and due to the increase of the system and IRQ processing, it grows with the payload size. These last results again confirm the consistency of the experiments.



(a) Network activity **(b)** CPU usage **(c)** System load

Figure 6.2.7: Performance parameters for one Raspberry Pi 4B varying the payload size every 2000 requests (about 10 minutes)

6.3 The P2PFaaS Framework

6.3.1 Motivation and significance

The Edge and the Fog Computing paradigms [2] arise from the need to distribute the computation among a set of nodes. In general, this necessity is a natural consequence of the application's non-functional requirements, which can regard the latency experienced by the users and service availability. The classic use case often refers to a smart city where computing nodes can be positioned in precise locations and also attached to 5G antennas [200]. In this scenario, we suppose that users are able to request services to the nearest node available. An inexorable issue that arises in this context is that we can often observe a non-negligible variation of the traffic to the nodes during the day [7]. This leads to some nodes being overwhelmed by a consistent number of requests per second (req/s that we call λ) thus the latency seen by the users for executing the service increase, and the node itself also can start to reject requests. In the meanwhile, other nodes may receive no traffic and be completely unloaded. This situation creates the necessity of designing load balancing algorithms which are able to reach a balanced load situation by allowing the nodes to forward part of their traffic to others. In particular, we focus on cooperative strategies which allow no central entity or orchestrator, but every node, aware of its neighbours, can make decisions (that can also be based on Reinforcement Learning) independently from others by asking them for information that can regard their current load or other performance parameters. The only assumption that we make is that the scheduling decision is made per-single function execution request and therefore in an online manner. Different works in literature [82], [99], [102] are focused on the solution to this problem, but most of them only consider mathematical models and event-based simulations, and in general, real environments present many details and unexpected conditions that are very difficult to be grasped in a model of the system. For example, the operating system of the nodes may perform additional work in parallel to the execution of the service, the particular programming language used may add more execution latency due to the fact that it is compiled or interpreted and if the QoS requirement is tied to the latency this aspect can be crucial.

In this Section, we present P2PFaaS, a software framework whose objective is the practical implementation of cooperative online scheduling and load balancing algorithms generally studied only in mathematical and simulation prospectives. The key terms of the framework denomination are: peer-to-peer (P2P), which refers to the fact that each node can be considered a peer in the network who can share tasks with others without a central entity or orchestrator; and FaaS, which refers to the Function-as-a-Service paradigm that is chosen as the task model. The idea of the framework, which is built on different modules deployed as Docker containers, compensates for a lack of flexibility in modern orchestrators, like Kubernetes, which do not allow a custom definition of scheduler algorithm when multiple containers are deployed in different machines. This is essentially given by the fact that they are built for production and not for research.

The P2PFaaS framework has already been used in different works [7], [13] in order to perform benchmarks of distributed algorithms in real environments. These tests required the installation of the framework both in x86 virtual machines and in ARM devices, in particular the Raspberry Pi.

6.3.2 Experimental Setting

The framework is written to be fully portable, indeed, it uses languages like Go and Python, which are available for all of the main architectures. For running the starting up the framework, the user needs to have only Docker installed, then it will be built from the source. Regarding the hardware, it will suffice to have x86 machines or even ARM-based nodes, while in the former case we leave it to the user to clone the source and build the framework within every node, in the latter we instead suggest that the deployment can be efficiently done by using OpenBalena[12] framework. OpenBalena requires to prepare the devices with a custom OS (called BalenaOS[13]) and then the P2PFaaS can be built for ARM and deployed to all the nodes in the set by using the `balena-cli`[14] tool.

For setting up the framework it will be needed to clone the repository `stack`[15] and use Docker Compose for building and running all the needed containers. Once all the containers are running the discovery service must be configured only at the first running of the nodes. This can be done by using the API `/configuration` at port 19000.

6.3.3 Software description

The proposed framework consists of independently developed modules. Each module has associated a code repository and it is built as a Docker container. This means that an always-alive process is associated with it. In general, it is a web server which exposes APIs routes. However, in delay-sensitive operations, websocket pools are used. This has been shown to drastically reduce the time for creating the request since the setting up of the TCP socket and the handshaking are only done once.

The building of the framework can be done in any machine that supports Docker and even in ARM architectures for which `Dockerfile.aarch64` are given.

6.3.3.1 Software Architecture

The overall architecture of the framework is shown in Figure 6.3.1 which clearly depicts the main modules.

- The *scheduler service* listens at port 18080 and represents the entrypoint of the framework where the clients can request the execution of a function via REST API. The service usually contacts the discovery service for retrieving the list of the nodes and the learner service for retrieving the scheduling action to perform when a Reinforcement Learning based scheduler is used. The essential role of the service is forwarding the request to the correct FaaS after making a scheduling decision that can be to execute the function locally or remotely. In the latter case, the scheduler of another node is called with the same payload as the original request.
- The *learner service* listens at port 19020 and implements the training and the inference of Reinforcement Learning models used by the scheduler service for making the scheduling decision.
- The *discovery service* listens at port 19000 and implements the nodes discovery.

[12] `https://www.balena.io/open/`
[13] `https://www.balena.io/os/`
[14] `https://github.com/balena-io/balena-cli`
[15] `https://gitlab.com/p2p-faas/stack`

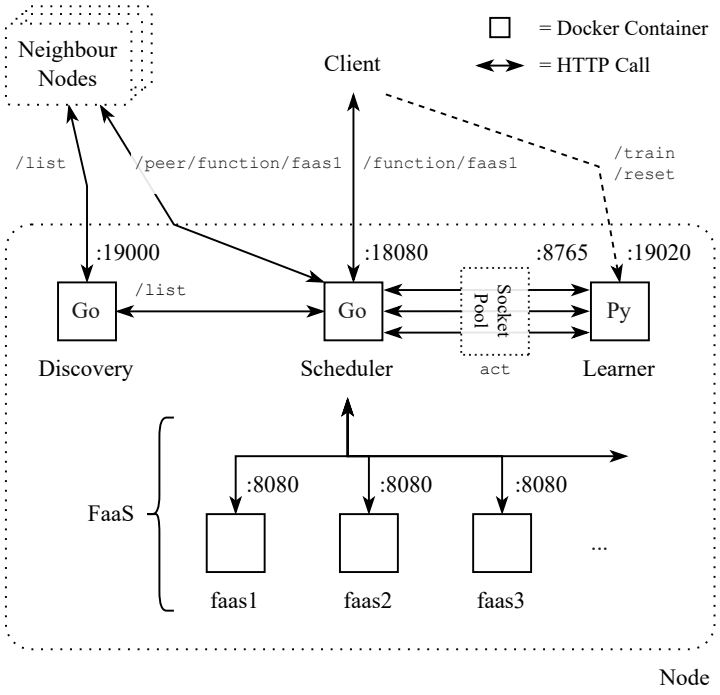


Figure 6.3.1: The P2PFaaS high-level software architecture. The framework comprises three core modules: the scheduler, the discovery and the learner services. Then a set of FaaS functions can be installed beside the framework. All of these components are deployed as Docker containers.

These three services compose the core of the framework, then the user must install one or more FaaS that implement the services offered by the node. For simplicity, we assume that every node implements the same set of functions and therefore the framework does not provide a way for the parallel deployment of the functions; indeed, this operation must be done manually or by using OpenBalena (see Section 6.3.2). The functions that can be used with the framework can be borrowed from the OpenFaaS[16] open source project. It will suffice to choose a function available and packaged with the `of-watchdog`[17] daemon and then build it with the tool `faas-cli`. However, the only requirement for the FaaS is that it must be deployed as a Docker container which implements a web server that executes the function when an HTTP call is issued at port 8080 and at the root '/' route. In [7], [13] the function that is used is the `pigo-openfaas`[18] function which implements a simple face recognition service.

Flow of operation The Figure 6.3.2 shows the flow of the operations that are carried out when the client (1) requests the execution of a function (called <fn> in the Figure). Once the framework is set up in a set of nodes, the flow of usage starts from a client which makes a request to a node, in particular to the `scheduler` service exposes at port 18080. The URL which must be called by the client is the following:

[16] `https://openfaas.com`
[17] `https://github.com/openfaas/of-watchdog`
[18] `https://github.com/esimov/pigo-openfaas`

```
http://ip:18080/function/<fn>
```

The placeholder <fn> must be replaced with the name of the function and it is mapped to the container name which implements the function. After making the request, the list of neighbours nodes is retrieved from the discovery service and cached. Then, a scheduling action is taken (2) and if a scheduler based on RL is configured, the current state is passed to the learner service which replies with the action to be taken. Once the action is known it is immediately executed (3) and this can require forwarding the request to another node. The request forwarding is implemented with an HTTP call to the URL:

```
http://remote-node-ip:18080/peer/function/<fn>
```

This HTTP will trigger the scheduler of the remote node and the task will be executed remotely or it can also be rejected. Otherwise, if the request has been marked as to be executed locally, the node will enqueue it and finally, it will be executed (4). The actual function execution is mapped to an HTTP call to the function's container. After the execution of the function, the output payload is finally forwarded to the client which will see its HTTP request to be concluded (5). At this point, there is an optional step that is executed only if the scheduler is RL based, that is the training of the model (6). Indeed, after the execution of the request, which is finished with the return of the output payload, we can derive the reward and forward it to the learner service which will update the weights of the model accordingly.

This concludes the operations that are needed for completing a FaaS execution request, we will now see in detail the core mechanisms of the three modules that we will call services in order to differentiate them from the sub-modules that compose them.

6.3.3.2 Scheduler Service

The scheduler module is written in Go. The language has been chosen because it is particularly tailored for the development of web servers. Figure 6.3.3 illustrates the architecture of the scheduler service with all the submodules that compose it.

As anticipated, the scheduler service can be seen as the entrypoint of the framework. This is because the client makes the function execution request directly to it. The handling of the APIs, declared in the root Go file of the service, is done by the *API* module that handles both the preparation of the function execution request that is then forwarded to the actual scheduler and the preparation of the payload for the configuration update.

Upon the booting of the container two threads are started, one is the web server thread which manages the HTTP API calls, the other is the worker thread which manages the internal task queue.

Scheduler The scheduler module has been designed in order to allow the interchangeability of the scheduling algorithms. For this reason, a `schedule` interface is declared (Listing 5). In this way, every scheduler algorithm can be easily instantiated and configured.

The `Schedule()` function implements the effective scheduling of the function execution request and three possible actions can be taken: the request is rejected, the request is executed in the current node or otherwise it can be forwarded to other nodes. When the request is rejected, the client HTTP

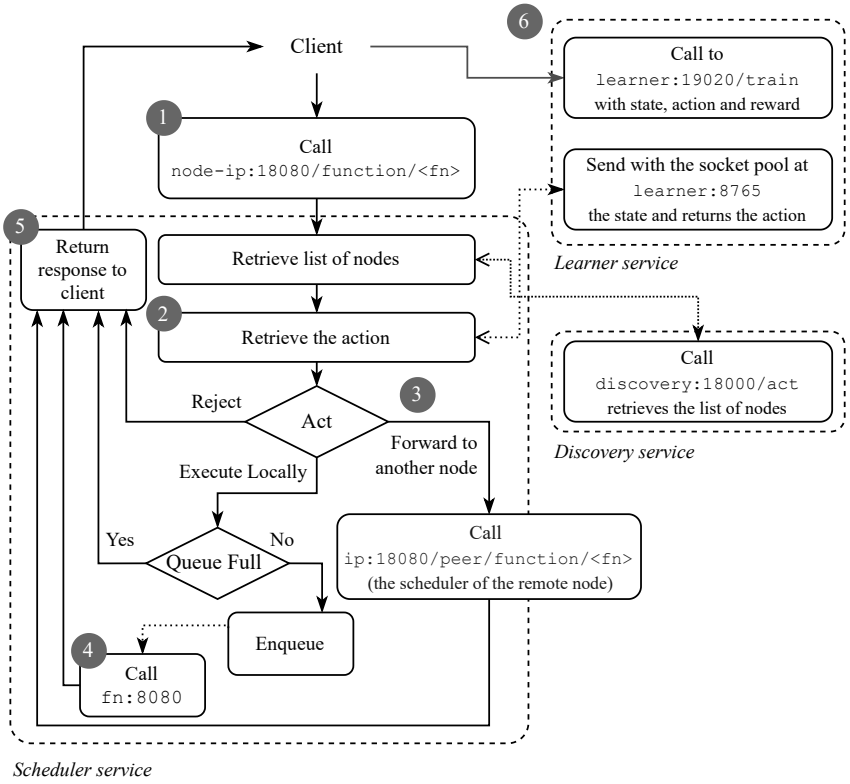


Figure 6.3.2: The flow of the operations that takes place after a function execution request is issued from the client to the scheduler service of the framework. The <fn> placeholder represents the name of the function and the calls are all HTTP calls. The example assumes the usage of Reinforcement Learning for making the scheduling decision.

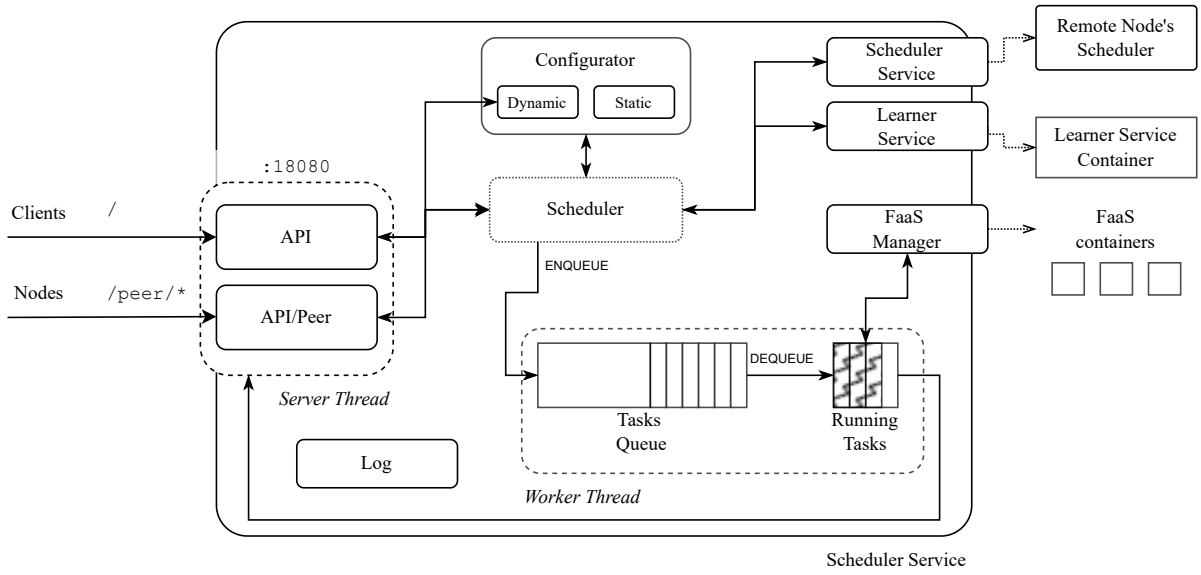


Figure 6.3.3: The architecture of the scheduler service.

Listing 5 The scheduler interface declares the `Schedule()` function which implements the scheduling algorithm.

```
type scheduler interface {
  GetFullName() string
  GetScheduler() *types.SchedulerDescriptor
  Schedule(req *types.ServiceRequest) (*JobResult, error)
```

request is immediately closed by returning the HTTP error code 500, otherwise, in other cases the request in enqueued locally or remotely. The forwarding of the request, which is done by using the "API/Peer" (Figure 6.3.3) module, again uses the function `Schedule()` for scheduling the request, but this time the same request will be marked as "External".

Queue The internal queue of the scheduler has been conceived with the idea of limiting the number of parallel running functions. The parameter that is often referred to as the number of parallel tasks that can be executed in a node is called K. When, for example, $K = 4$ we are assuming that the maximum number of parallel running FaaS functions is 4. The queue is managed by a thread which implements the producer-consumer scheme, in this way the functions are started only when at least one running slot is available. The queue can be also limited in size, and in this case, when it is full, the requests are automatically rejected. The queue that is implemented in the described way limits the parallelism and specifically targets Edge devices which do not have a relevant computational power and at the same time it matches with models which are based M/M/1/K and M/M/K/K queues.

Configurator The configurator module is in charge to read and write the configuration of the scheduler service. Configurations can be of two types: static and dynamic. The former regard parameters that are loaded upon the boot of the container through environment variables and the latter instead are passed via HTTP API calls and then saved to a JSON file. The scheme is applied to all of the other services. In particular, in this service, the static parameters regard the port and the host to which the server will listen, the logging verbosity and the path in which the dynamic configuration will be saved. The dynamic parameters are instead divided into two parts, the ones which refer to the service itself which can be set by using the API

`http://ip:18080/configurator`

and they regard, for example, the K parameter and the queue length; and the others instead to the scheduler algorithm and they can be set by using the API

`http://ip:18080/configurator/scheduler`

and these instead regard the name of the scheduler algorithm to be set and a variable array of parameters which set the algorithm's behaviour.

FaaS Manager The FaaS Manager module is in charge of forwarding the function execution request to the correct FaaS function container. The module is conceived for allowing a further level of decoupling

in order to allow different FaaS container technologies. The name resolution that translates the name of the container to the IP address of the FaaS container is done automatically by Docker.

Other modules The remaining modules are the "Log" module which is in charge to manage the logging, indeed extensive logging may slow down the service and increase the latency of the tasks, then we have the "Scheduler Service" module and the "Learner Service" module which both are in charge of allowing the interoperability between the Scheduler Service and the other services.

6.3.3.3 Learner Service

The learner module is written in Python. The language has been chosen because it is widely used for machine learning. The role of the module is of implementing Reinforcement Learning models which are used for making scheduling decisions. Figure 6.3.4 illustrates the overall structure of the service.

The learning process Reinforcement learning models need three fundamental entities for operating: the state, the actions and the reward. The state is encoded as a string and in general, it contains the current load of the node, the action is mapped to a scheduling action and can be to execute the task locally, reject it or forward it to another node. Finally, the reward drives the learning process and it can refer to the total task duration. For example, we may assign a positive reward if a task is completed within a certain deadline. For implementing this paradigm, we need to make the clients able to train the model, because the final delay is only known when the output payload of the function reaches the client. For this reason, the "API" module implements the `/train` and `/train_batch` routes, the first for the training of a single entity and the second for multiple entities at a time.

The learning entities The training of the model is carried out by passing to the learner thread blocks of learning information wrapped in structures called learning entities. A learning entity contains a progressive number, the state (as a string), the action (as a float) and the reward (as a float).

The learning thread The learning process is carried out by the learner thread which is in charge to defer the training upon the fact that all the needed entities are present. Indeed, the training process must follow the specific order according to which tasks are generated by the client but it may happen that tasks did not complete in the same order according to which they are generated. For this reason, to each arriving task to the Scheduler Service, a progressive number is attached, then after the action is taken, the number (called "EID"), the state and the action are transferred (through HTTP headers) to the client which finally triggers the training. The learning thread for deferring the learning implements a producer-consumer scheme has been implemented in such a way the training only starts when a W learning entries with consecutive EIDs are in the queue which is continuously sorted.

The learning model The weights associated with the learning model are updated by the "Value Function" module which implements the approximation of the $Q(s, a)$ [132] function. Both the Q-Table and the Tiling methods are implemented but the framework can easily be extended even with Deep Neural Networks (DNNs). The Value Function model updates the weights according to the error that is

computed by the "Bellman TD Form". This module, given the current state, the action, the next state, the next action and the reward, returns the δ. For example, the Sarsa learning strategy for the average the reward is [132]

$$q_*(s,a) = \sum_{r,s'} p(s',r|s,a)\left[r - \max_{\pi} r(\pi) + \max_{a'} q_*(s',a')\right] \tag{6.1}$$

the time differential form of the Equation 6.1 is

$$\Delta_t = [R_{t+1} - \bar{R}_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{6.2}$$

which is returned by the "Bellman TD Forms" module and applied by the Value Function module by using the Q-Table as in Equation 6.3.

$$Q(S_t, A_t) \leftarrow Q(S,a) + \alpha\Delta_t \tag{6.3}$$

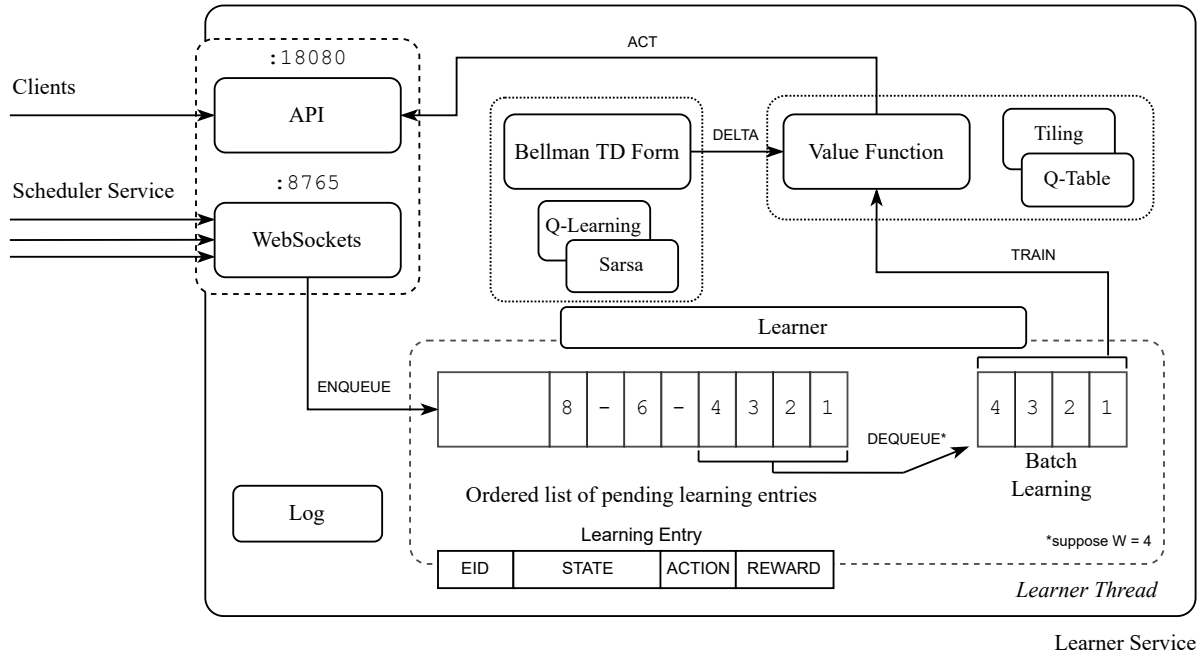


Figure 6.3.4: The architecture of the learner service. The learning entries, composed by a progressive EID, the state, the action and the reward are enqueued in a sorted list, then as soon as W consecutive entries are available the training of the batch is started. The training of the model is done through the "Value Function" module which retrieves the delta to be applied to the weights from the "Bellman TD Forms" which describes the learning model.

6.3.3.4 Discovery Service

The discovery module is written in Go, and its purpose is to allow the nodes to know which are their neighbours. The service is based on a gossip algorithm and must be configured at boot with the IP of another node (called "init server"); then, when another node, suppose B, requests to it, node A, the

list of all the nodes that it knows, the IP of B will be added list nodes known by A. For now, only fully connected topologies are supported by the framework and therefore, if every node is initialised with the same init server, then every node will eventually be aware of each other.

6.3.4 Illustrative Examples

Examples of the running framework have been illustrated in [13] and in [7]. In particular, [13] shows an early version of the framework running a benchmark on a power-of-n choices-based algorithm for distributed load balancing that follows a randomised approach. Instead, in [7] the framework has been used to show in practice how a Reinforcement Learning based approach for making the scheduling decision can be used on real devices. Indeed, after testing the solution in a simulated environment, the framework has been installed on 12 Raspberry Pi 4 and a Sarsa-based RL strategy has been used.

All the scripts used for running the benchmarks have been published as open source. They are available in the `experiments`[19] repository.

6.3.5 Impact

The P2PFaaS framework presented in this section is probably the first framework available as open source which allows the implementation of distributed scheduling and load balancing algorithms between nodes by following a fully decentralised (peer-to-peer) scheme. Indeed, its flexibility is the maximum possible achievable since the development of the framework started from the constraints imposed by well-known production frameworks. P2PFaaS does not have the same level of maturity as them but for researchers in the field, it can allow testing if the designed algorithms can have a possible implementation in real devices and under which conditions they can work. Moreover, after defining the FaaS function, the scheduler can be easily written within the core of the scheduler service and changed.

Due to the portability of the code, P2PFaaS is also easy to be deployed in multiple SoC computers (like Raspberry Pis) by leveraging OpenBalena and therefore avoid using virtual machines in order to test the algorithms on real computer devices which can be bought in bulk due to their affordable cost. Testing this kind of algorithms in real devices has a clear impact on the research and, in particular, on the algorithm design. A series of conditions and peculiar characteristics of real environments cannot be easily grasped by simulations and mathematical models. For example, in the original design of the Learner Service, the Scheduler Service had to ask for the action of the Learner by means of HTTP calls. However, these HTTP calls added a fixed delay of about 10ms to each request. When testing a deadline-based scheduling algorithm this is revealed to be a critical issue, indeed, the RL-based approach was not able to outperform even a simple randomised approach. This led to the replacement of the HTTP calls with a pool of 20 web sockets which are now used in parallel only for requesting the action to the Learner Service. Therefore, mathematical models and simulations can give a direction about the performance of the algorithms in a world that is simplified, but they are fundamental to study the algorithms.

[19] `https://gitlab.com/p2p-faas/experiments`

Chapter 7

Conclusions & Future Research Directions

In this thesis, different works have been presented and they were all focused on the design of algorithms and solutions for targeting load balancing, scheduling and offloading in Fog and Edge Computing environments. In this final section, we will draw conclusions about all the main topics covered and we will try to delineate future research directions.

Distributed algorithms for load balancing (and task offloading)

- Section 2.2 illustrated the performance of the randomised load balancing algorithm Least Loaded among d nodes ($LL(d)$) [201] when adapted to a Fog Computing deployment. This adaptation consisted of triggering the randomised search only when the workload of the current Fog node that receives a new job to execute is above a threshold value, T. The threshold is shown to be a simple way to reduce control delays without affecting the very nature of the power-of-random choices principle. Through a mathematical analysis we show that under Poisson arrivals and an exponential distributed service time, setting $T = K - 2$, where K is the number of servers of a node, achieves practically the same performance of the *power-of-n* choices classic implementation requiring a single global scheduler, but at much lower delay penalty and control overhead of up to one order of magnitude less. Simulation experiments and a real implementation corroborated the finding;
- in Section 2.3 two innovative algorithms, namely *Sequential Forwarding* and *Adaptive Forwarding* has been presented and they aim to provide load balancing in a Fog Computing infrastructure. The two algorithms are explicitly designed to be extremely simple and to be fully distributed to provide fair load sharing in highly heterogeneous scenarios with variable workload levels and high network delays which are typical characteristics of Fog computing. The results of the experiments suggest that the proposed algorithms clearly outperform the case where no load balancing is applied with a reduction in the drop rate by a factor of 19 and by a reduction in response time up to 19%. The results in a realistic scenario are even more impressive as we can nearly halve the response time and we can reduce the loss rate from 13% to less than 0.2%. However, additional

features such as managing jobs with different priorities and different dropping policies is an interesting space not considered in the present paper but that could be addressed in future works;

- Section 2.4 focused on the load balancing issue of distributing incoming jobs over the nodes of an Edge computing infrastructure. Specifically, the analysis concerned the impact of stale load information caused by network latency on the effectiveness of load balancing algorithms based on the randomisation of jobs dispatching over the nodes, showing that when this latency is comparable with the service time, the algorithm performs poorly. The study has been carried out from two different points of view: a mathematical model and a full-fledged simulator. The analysis revealed that taking schedule decisions based on state information received even with a small delay compared to the service time reduces the load balancing effectiveness considerably. In this setting, it is convenient to keep the randomisation principle incorporating it as a blind forward towards neighbouring nodes. The addition of a threshold to regulate the triggering of the algorithm is a valid method to reach high performance.
- Section 3.2 studied the mathematical modelling of a system of n Fog or Edge nodes for designing a dynamic which is able to level the service latency among all the nodes in a given topology. Then, even if from the model we are able to derive the solution, that is the migration ratios m_{ij} from any node i to a node j, we designed a fully decentralised and adaptive heuristic which is able to reach the same solution but without the need to have a centralised entity (which is able to run the model) and with potential capability to adapt when the load varies over time. The algorithm has been run both in simulations and in a real deployment of Raspberry Pi boards and we showed how the solution is very similar to the one predicted by the mathematical model. However, further research directions are needed to improve the proposed approach. First of all, the communication latency has to be included in the model while in the analysed case we only consider them in the final Raspberry Pi deployment which justifies the differences in the results, moreover, a more precise model for a real node must be studied since the M/M/1/K does not approximate exactly a real computer node, and this again justifies the discrepancy between the model and the final deployment results. Then, as the last improvements points, a load that varies over time can be introduced in the model, instead of having a fixed λ_i we can suppose to have a $\lambda_i(t)$ function and we can also consider to jointly level even other performance parameters beyond the single service latency;
- Section 5.2 presented a complete environment for allowing a real-time object recognition task, and it focused firstly on running it locally in the device and then offloading it on the Edge. we used free and open-source frameworks, we assessed their maturity and their ease of use. The experiments that we conducted show that, despite the fact that we can now rely on a big set of neural networks deeply optimised for mobile devices, it is far more convenient to offload a deep learning task to the Fog/Edge network. This strategy is even more corroborated by the fact that the Edge environments are able to offer very low latencies. However, this was only a first attempt to test the ground on this field. Indeed, we envision as future work to conduct more experiments by exploiting a wider range of neural networks and frameworks for assessing their convenience even with other types of computer vision tasks, and also by exploiting the WiFi6

technology that allows to drastically reduce network latencies. Moreover, we also envision using in Edge/Fog layer less powerful computing units, like for example Raspberry Pis equipped with specialised ML processing chips (e.g. Coral USB Accelerator[1]).

- Section 5.3 focused on the study of a deployment configuration of Edge nodes supplied by PV panels. Using numerical models we provided a first assessment of the advantage of performing cooperation among Edge nodes to maximise the use of solar energy. The algorithm is the first step towards a new class of energy-aware resource sharing and cooperation among nodes.

Distributed algorithms for scheduling

- Section 4.2 addressed the problem of extending the *power-of-n* choices distributed scheduling scheme with Reinforcement Learning in order to be able to efficiently schedule real time and deadline-constrained tasks. Starting from the simple approach in which the agent learns a known policy, we arrived to provide simulation-based results that the approach works even if the load conditions are typical of a real Fog deployment in a smart city. we showed that a fully distributed scheduling approach based on reinforcement learning, in which every node is an agent and it does not have any kind of load information about the others, is able to maximise the performances of every single node by not behaving in a selfish manner. However, other environmental characteristics can be studied in order to reveal the true efficiency of the approach, for example by introducing a variable communication delay between the nodes, considering that the nodes maintain a periodically updated value of the load of the others, or even increasing the complexity of the state in order to take into account other factors like CPU time and RAM consumption;
- Section 4.3 presented an approach for solving the online task scheduling in the Edge or Fog to cloud continuum computing model by using Reinforcement Learning. This approach is perfectly suiting the problems that regard this dynamic context, for example, heterogeneity of the nodes, difficulties in estimating the real execution speed of the nodes, the possible failure of the nodes, cooperation strategies and different QoS requirements (e.g. minimum frame rate). The results of my approach have been shown both in a single cluster and in a multi-cluster environment, and they demonstrated that in any of these cases, given a hypothetical traffic flow the agent, placed in the scheduler of each cluster, can derive the best scheduling policy without nothing anything about the characteristics of the worker nodes or of the neighbour clusters. As anticipated in the Section, unfortunately, some points have been left open and will be further investigated, for example, in the experiments we hypothesised that the nodes speed is fixed but, in general, it fluctuates over time since every worker node has an underlying operating system and CPU time may be reserved for other applications, a further study should investigate the frame skipping, that occurs when the processed frames return to the client in an order that is different from the generation one, and a further investigation should be focused on the consequences of using a higher number of task types.

[1] `https://www.coral.ai/products/accelerator`

Testbeds and implementation design

- Section 6.2 showed a long-term solution for building a cluster of Raspberry Pi that is self-enclosed and tries to minimise the intervention of a human operator. we designed a power supply and a control board system for using a desktop PC case (ATX2RPi8), we presented a set of software guidelines, a Testbed-as-a-service configuration file architecture and finally we showed a usage scenario of the cluster, namely the implementation and the benchmarking of a distributed scheduling algorithm in a very close to real Fog computing deployment. However, some aspects should be further investigated, for example, the nodes' configuration is static and cannot change dynamically. Indeed, it will be needed to consider the possibility to provide a centralised configuration solution for managing the single SBCs operating system parameters (like adding new users or SSH keys), for switching on/off desired nodes, or setting up a particular network topology for running specific benchmarks of other algorithms by using the JSON configuration files structure that we proposed;
- Section 6.3 presented the concept of "P2PFaaS" framework that we designed and implemented to test and experiment with load balancing and scheduling algorithms in real Fog and Edge environments. Due to the modularity of the framework, the extension is made as easy as possible, and further improvements have already been started regarding the energy aspect of the scheduling algorithms.

To summarize, future research directions from the topics covered in this thesis essentially regard two main areas. Firstly, Green Edge Computing, indeed the optimal usage of renewable energy in the case of Edge computing is a topic that is recently becoming particularly relevant, especially when applied to precision agriculture contexts. Indeed, the work presented in Section 5.3 is only a preliminary study which opens for me a new research branch on scheduling and load balancing algorithms focused on the energy consumption of the edge devices. Then, the second area regards the innovative core of my PhD which is the application of Reinforcement Learning strategies to distributed scheduling and load balancing algorithms. Indeed, in this thesis, we presented an innovative approach of using the models which regard the multi-agent approach to the problem. Indeed, they concretise in the mapping of each node to a learning agent and then make them learn even from the action of the others. A clear future research direction is the further development of RL models, introducing further QoS parameters and network characteristics like slow or saturated links. Moreover, these models can also be applied to optimize the usage of green energy in edge devices, designing algorithms which explicitly target energy consumption.

Bibliography

[1] T. G. Peter Mell, "The nist definition of cloud computing," NIST, Tech. Rep., 2011.

[2] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi, "Fog computing conceptual model," NIST, Tech. Rep., 2018.

[3] G. Proietti Mattia and R. Beraldi, "P2pfaas: A framework for faas peer-to-peer scheduling and load balancing in fog and edge computing," *SoftwareX*, vol. 21, p. 101 290, 2023, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2022.101290. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711022002084.

[4] G. Proietti Mattia, M. Magnani, and R. Beraldi, "A latency-levelling load balancing algorithm for fog and edge computing," in *25th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'22)*, Montreal, Canada, Oct. 2022. DOI: 10.1145/3551659.3559048. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3551659.3559048.

[5] R. Beraldi and G. Proietti Mattia, "On off-grid green solar panel supplied edge computing," in *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS) (IEEE MASS 2022)*, Denver, USA, Oct. 2022.

[6] G. Maiorano, G. Proietti Mattia, and R. Beraldi, "Local and remote fog based trade-offs for qoe in vr applications by using cloudxr and oculus air link," in *International Conference on Edge Computing and Applications (ICECAA 2022)*, Namakkal, India, Oct. 2022.

[7] G. Proietti Mattia and R. Beraldi, "On real-time scheduling in fog computing: A reinforcement learning algorithm with application to smart cities," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022, pp. 187–193. DOI: 10.1109/PerComWorkshops53856.2022.9767498.

[8] R. Beraldi, C. Canali, R. Lancellotti, and G. Proietti Mattia, "On the impact of stale information on distributed online load balancing protocols for edge computing," *Computer Networks*, p. 108 935, 2022, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2022.108935. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128622001207.

[9] G. Proietti Mattia and R. Beraldi, "Leveraging reinforcement learning for online scheduling of real-time tasks in the edge/fog-to-cloud computing continuum," in *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, 2021, pp. 1–9. DOI: 10.1109/NCA53618.2021.9685413.

[10] G. Proietti Mattia and R. Beraldi, "A study on real-time image processing applications with edge computing support for mobile devices," in *2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2021, pp. 1–7. DOI: 10.1109/DS-RT52167.2021.9576139.

[11] T. Alawsi, G. Proietti Mattia, Z. Al-Bawi, and R. Beraldi, "Smartphone-based colorimetric sensor application for measuring biochemical material concentration," *Sensing and Bio-Sensing Research*, p. 100 404, 2021, ISSN: 2214-1804. DOI: https://doi.org/10.1016/j.sbsr.2021.100404. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S221418042100009X.

[12] G. Proietti Mattia and R. Beraldi, "Towards testbed as-a-service: Design and implementation of an unattended soc cluster," in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021, pp. 1–8. DOI: 10.1109/ICCCN52240.2021.9522323. [Online]. Available: https://ieeexplore.ieee.org/document/9522323.

[13] R. Beraldi and G. Proietti Mattia, "Power of random choices made efficient for fog computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020, ISSN: 2372-0018. DOI: 10.1109/TCC.2020.2968443.

[14] R. Beraldi, C. Canali, R. Lancellotti, and G. Proietti Mattia, "A random walk based load balancing algorithm for fog computing," in *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, 2020, pp. 46–53. DOI: 10.1109/FMEC49853.2020.9144962.

[15] R. Beraldi, C. Canali, R. Lancellotti, and G. Proietti Mattia, "Distributed load balancing for heterogeneous fog computing infrastructures in smart cities," *Pervasive and Mobile Computing*, p. 101 221, 2020, ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2020.101221. [Online]. Available: http://www.

sciencedirect.com/science/article/pii/S1574119220300791.

[16] R. Beraldi, C. Canali, R. Lancellotti, and G. Proietti Mattia, "Randomized load balancing under loosely correlated state information in fog computing," in *23rd ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'20)*, Alicante, Spain, Nov. 2020. DOI: 10.1145/3416010.3423244.

[17] R. Beraldi and G. Proietti Mattia, "A randomized low latency resource sharing algorithm for fog computing," in *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT) (DS-RT'19)*, Cosenza, Italy, Oct. 2019. DOI: 10.1109/DS-RT47707.2019.8958709. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8958709.

[18] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[19] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001.

[20] M. Bramson, Y. Lu, and B. Prabhakar, "Asymptotic independence of queues under randomized load balancing," *Queueing Syst. Theory Appl.*, vol. 71, no. 3, pp. 247–292, Jul. 2012, ISSN: 0257-0130.

[21] F. Garcia-Carballeira and A. Calderón, "Reducing randomization in the power of two choices load balancing algorithm," in *2017 International Conference on High Performance Computing and Simulation (HPCS)*, Jul. 2017, pp. 365–372.

[22] A. Yousefpour, G. Ishigaki, and J. P. Jue, "Mean-field analysis of loss models with mixed-erlang distributions under power-of-d routing," in *29th International Teletraffic Congress (ITC 29)*, Sep. 2017. DOI: 10.23919/ITC.2017.8064362.

[23] T. Hellemans and B. Van Houdt, "On the power-of-d-choices with least loaded server selection," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, Jun. 2018. DOI: 10.1145/3224422. [Online]. Available: https://doi.org/10.1145/3224422.

[24] Y. L. Qiaomin Xie Xiaobo Dong and R. Srikant, "Power of d choices for large-scale bin packing: A loss model," in *Proceedings of ACM SIGMETRICS 2015*, IEEE, vol. 43, New York, NY, USA: ACM Press, 2015, pp. 321–334.

[25] S. R. E. Turner, "Resource pooling in stochastic networks," Ph.D. dissertation, University of Cambridge, 1997. [Online]. Available: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.627069.

[26] A. Mukhopadhyay, R. R. Mazumdar, and F. Guillemin, "The power of randomized routing in heterogeneous loss systems," in *2015 27th International Teletraffic Congress*, 2015, pp. 125–133.

[27] C. Fricker, F. Guillemin, P. Robert, and G. Thompson, "Analysis of an offloading scheme for data centers in the framework of fog computing," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 4, 16:1–16:18, Sep. 2016, ISSN: 2376-3639.

[28] R. Beraldi, H. Alnuweiri, and A. Mtibaa, "A power-of-two choices based algorithm for fog computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2018, ISSN: 2372-0018. DOI: 10.1109/TCC.2018.2828809.

[29] S. Fu, C.-Z. Xu, and H. Shen, "Random choices for churn resilient load balancing in peer-to-peer networks," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536282.

[30] E. C. P. N. G. C. F. Aires, "An algorithm to optimise the load distribution of fog environments," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, Piscataway, New Jersey, US: IEEE, Oct. 2017, pp. 1292–1297. DOI: 10.1109/SMC.2017.8122791.

[31] A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, and C. Z. Patrikakis, "A cooperative fog approach for effective workload balancing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 36–45, Mar. 2017, ISSN: 2325-6095. DOI: 10.1109/MCC.2017.25.

[32] Y. Song, S. S. Yau, R. Yu, X. Zhang, and G. Xue, "An approach to qos-based task distribution in edge computing networks for iot applications," in *2017 IEEE international conference on edge computing (EDGE)*, IEEE, Piscataway, New Jersey, US: IEEE, Jun. 2017, pp. 32–39. DOI: 10.1109/IEEE.EDGE.2017.50.

[33] S. K. Mishra, M. A. Khan, B. Sahoo, D. Puthal, M. S. Obaidat, and K.-F. Hsiao, "Time efficient dynamic threshold-based load balancing technique for cloud computing," in *2017 International Conference on Computer, Information and Telecommunication Systems (CITS)*, IEEE, 2017, pp. 161–165.

[34] D. Puthal, M. S. Obaidat, P. Nanda, M. Prasad, S. P. Mohanty, and A. Y. Zomaya, "Secure and sustainable load balancing of edge data centers in fog computing," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 60–65, 2018.

[35] D. Puthal, R. Ranjan, A. Nanda, P. Nanda, P. P. Jayaraman, and A. Y. Zomaya, "Secure authentication and load balancing of distributed edge datacenters," *Journal of Parallel and Distributed Computing*, vol. 124, pp. 60–69, 2019.

[36] S. K. Mishra, D. Puthal, B. Sahoo, S. Sharma, Z. Xue, and A. Y. Zomaya, "Energy-efficient deployment of edge dataenters for mobile clouds in sustainable iot," *IEEE Access*, vol. 6, pp. 56 587–56 597, 2018.

[37] A. Yousefpour, G. Ishigaki, and J. P. Jue, "Fog Computing: Towards Minimizing Delay in the Internet of Things," *Proceedings - 2017 IEEE 1st International Conference on Edge Computing, EDGE 2017*, pp. 17–24, 2017.

[38] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang, "Optimal Workload Allocation in Fog-Cloud Computing

Toward Balanced Delay and Power Consumption," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 1171–1181, Dec. 2016.

[39] C. Canali and R. Lancellotti, "GASP: Genetic Algorithms for Service Placement in fog computing systems," *Algorithms*, vol. 12, no. 10, pp. 1–19, 2019, ISSN: 1999-4893. DOI: 10.3390/a12100201.

[40] M. Dahlin, "Interpreting stale load information," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 10, pp. 1033–1047, Oct. 2000.

[41] M. Mitzenmacher, "How useful is old information?" *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 1, pp. 6–20, 2000.

[42] D. Huang, P. Wang, and D. Niyato, "A dynamic offloading algorithm for mobile computing," *IEEE Transactions on Wireless Communications*, vol. 11, no. 6, pp. 1991–1995, 2012.

[43] J. Zheng, Y. Cai, Y. Wu, and X. Shen, "Dynamic computation offloading for mobile cloud computing: A stochastic game-theoretic approach," *IEEE Transactions on Mobile Computing*, vol. 18, no. 4, pp. 771–786, 2019.

[44] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Transactions on Communications*, vol. 64, no. 10, pp. 4268–4282, 2016.

[45] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016.

[46] S. Wang, Y. Zhao, J. Xu, J. Yuan, and C.-H. Hsu, "Edge server placement in mobile edge computing," *Journal of Parallel and Distributed Computing*, vol. 127, pp. 160–168, 2019.

[47] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, 2018.

[48] M. Satyanarayanan, W. Gao, and B. Lucia, "The computing landscape of the 21st century," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '19, Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 45–50, ISBN: 9781450362733. DOI: 10.1145/3301293.3302357. [Online]. Available: https://doi.org/10.1145/3301293.3302357.

[49] E. Eriksson, G. Dán, and V. Fodor, "Predictive distributed visual analysis for video in wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 15, no. 7, pp. 1743–1756, 2016.

[50] C. Long, Y. Cao, T. Jiang, and Q. Zhang, "Edge computing framework for cooperative video processing in multimedia iot systems," *IEEE Transactions on Multimedia*, vol. 20, no. 5, pp. 1126–1139, 2018.

[51] Z. Zhou, H. Liao, B. Gu, K. M. S. Huq, S. Mumtaz, and J. Rodriguez, "Robust mobile crowd sensing: When deep learning meets edge computing," *IEEE Network*, vol. 32, no. 4, pp. 54–60, 2018.

[52] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas, and Q. Zhang, "Edge computing in iot-based manufacturing," *IEEE Communications Magazine*, vol. 56, no. 9, pp. 103–109, 2018.

[53] A. Aijaz, M. Dohler, A. H. Aghvami, V. Friderikos, and M. Frodigh, "Realizing the tactile internet: Haptic communications over next generation 5g cellular networks," *IEEE Wireless Communications*, vol. 24, no. 2, pp. 82–89, Apr. 2017, ISSN: 1536-1284. DOI: 10.1109/MWC.2016.1500157RP.

[54] 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*, ser. MCC '12, Helsinki, Finland, 2012, pp. 13–16, ISBN: 978-1-4503-1519-7.

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

[56] H. El-Sayed, S. Sankar, M. Prasad, *et al.*, "Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment," *IEEE Access*, vol. 6, pp. 1706–1717, 2017.

[57] R. K. Naha, S. Garg, D. Georgakopoulos, *et al.*, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE access*, vol. 6, pp. 47 980–48 009, 2018.

[58] R. Buyya, S. N. Srirama, G. Casale, *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Comput. Surv.*, vol. 51, no. 5, Nov. 2018, ISSN: 0360-0300. DOI: 10.1145/3241737. [Online]. Available: https://doi.org/10.1145/3241737.

[59] OpenFog Consortium Architecture Working Group, "Openfog reference architecture for fog computing," OpenFog Consortium, Tech. Rep. OPFRA001.020817, Feb. 2017.

[60] Z. Li, M. L. Sichitiu, and X. Qiu, "Fog radio access network: A new wireless backhaul architecture for small cell networks," *IEEE Access*, vol. 7, pp. 14 150–14 161, 2019, ISSN: 2169-3536.

[61] P. Marsch, I. Da Silva, O. Bulakci, *et al.*, "5g radio access network architecture: Design guidelines and key considerations," *IEEE Communications Magazine*, vol. 54, no. 11, pp. 24–32, Nov. 2016, ISSN: 0163-6804.

[62] ETSI Industry Specification Group (ISG), "Mobile edge computing (mec); framework and reference architecture," ETSI, Standard ETSI GS MEC 003 V1.1.1, Mar. 2016.

[63] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 1270–1278.

[64] Y. Xiao and M. Krunz, "Distributed optimization for energy-efficient fog computing in the tactile internet," *IEEE Journal on Selected Areas in Communica-*

tions, vol. 36, no. 11, pp. 2390–2400, Nov. 2018, ISSN: 0733-8716. DOI: 10.1109/JSAC.2018.2872287.

[65] S. M. A. Oteafy and H. S. Hassanein, "Leveraging tactile internet cognizance and operation via iot and edge technologies," *Proceedings of the IEEE*, pp. 1–12, 2018, ISSN: 0018-9219. DOI: 10.1109/JPROC.2018.2873577.

[66] K. Velasquez, D. P. Abreu, M. R. Assis, *et al.*, "Fog orchestration for the internet of everything: State-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 14, 2018.

[67] A. N. M. Roberto Beraldi Abderrahmen Mtibaa, "Cico: A credit-based incentive mechanism for cooperative fog computing paradigms," in *Globecom 2018*, IEEE, 2018.

[68] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, ACM, 2013, pp. 69–84.

[69] R. Beraldi and H. Alnuweiri, "Sequential randomization load balancing for fog computing," in *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Sep. 2018.

[70] T. Pering, Y. Agarwal, R. Gupta, and R. Want, "Coolspots: Reducing the power consumption of wireless mobile devices with multiple radio interfaces," in *Proceedings of the 4th international conference on Mobile systems, applications and services*, ACM, 2006, pp. 220–232.

[71] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2017, pp. 2658–2659.

[72] 5G PPP Architecture Working Group and others, "View on 5g architecture," *White Paper, December*, 2017.

[73] J. Hong, Y. Hong, and J. Youn, *Problem statement of iot integrated with edge computing*, 2018.

[74] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on fog computing: Stateof-the-art and research challenges," in *IEEE Communications Surveys*, IEEE, vol. 20, 2017, pp. 416–464. DOI: 10.1109/COMST.2017.2771153.

[75] C. Canali and R. Lancellotti, "Paffi: Performance analysis framework for fog infrastructures in realistic scenarios," in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, Rome, Italy, Oct. 2019, pp. 1–8.

[76] C. Graham, "Chaoticity on path space for a queueing network with selection of the shortest queue among several," *Journal of Applied Probability*, vol. 1, no. 37, pp. 198–211, 2000.

[77] E. Khorov, A. Lyakhov, A. Krotov, and A. Guschin, "A survey on IEEE 802.11 ah: An enabling networking technology for smart cities," *Computer Communications*, vol. 58, pp. 53–69, 2015.

[78] P. Varshney and Y. Simmhan, "Characterizing application scheduling on edge, fog, and cloud computing resources," *Software: Practice and Experience*, vol. 50, no. 5, pp. 558–595, 2020. DOI: https://doi.org/10.1002/spe.2699. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2699.

[79] L. Kleinrock, *Queueing Systems, vol 1: theory*. 1975, vol. John Wiley & Sons, Inc. DOI: https://doi.org/10.1016/j.pmcj.2020.101221.

[80] E. U. Michael Mitzenmacher, *Probability and computing: randomized algorithms and probabilistic analysis*. The Edinburgh Building, Cambridge CB2 2RU, UK: CAMBRIDGE UNIVERSITY PRESS, 2005, ISBN: 0521835402.

[81] M. Haghi Kashani and E. Mahdipour, "Load balancing algorithms in fog computing: A systematic review," *IEEE Transactions on Services Computing*, pp. 1–1, 2022. DOI: 10.1109/TSC.2022.3174475.

[82] M. Kaur and R. Aron, "A systematic study of load balancing approaches in the fog computing environment," *The Journal of Supercomputing*, vol. 77, no. 8, pp. 9202–9247, Aug. 2021, ISSN: 1573-0484. DOI: 10.1007/s11227-020-03600-8. [Online]. Available: https://doi.org/10.1007/s11227-020-03600-8.

[83] A. Chandak and N. K. Ray, "A review of load balancing in fog computing," in *2019 International Conference on Information Technology (ICIT)*, 2019, pp. 460–465. DOI: 10.1109/ICIT48102.2019.00087.

[84] S. Harnal, G. Sharma, N. Seth, and R. D. Mishra, "Load balancing in fog computing using qos," in *Energy Conservation Solutions for Fog-Edge Computing Paradigms*. Singapore: Springer Singapore, 2022, pp. 147–172, ISBN: 978-981-16-3448-2. DOI: 10.1007/978-981-16-3448-2_8. [Online]. Available: https://doi.org/10.1007/978-981-16-3448-2_8.

[85] D. Baburao, T. Pavankumar, and C. S. R. Prabhu, "Load balancing in the fog nodes using particle swarm optimization-based enhanced dynamic resource allocation method," *Applied Nanoscience*, Jul. 2021, ISSN: 2190-5517. DOI: 10.1007/s13204-021-01970-w. [Online]. Available: https://doi.org/10.1007/s13204-021-01970-w.

[86] S. S. Tripathy, R. K. Barik, and D. S. Roy, "Secure-m2fbalancer: A secure mist to fog computing-based distributed load balancing framework for smart city application," in *Advances in Communication, Devices and Networking*, S. Dhar, S. C. Mukhopadhyay, S. N. Sur, and C.-M. Liu, Eds., Singapore: Springer Singapore, 2022, pp. 277–285, ISBN: 978-981-16-2911-2.

[87] Q.-M. Nguyen, L.-A. Phan, and T. Kim, "Load-balancing of kubernetes-based edge computing infrastructure using resource adaptive proxy," *Sensors*, vol. 22, no. 8, 2022, ISSN: 1424-8220. DOI: 10.3390/

s22082869. [Online]. Available: https://www.mdpi.com/1424-8220/22/8/2869.

[88] A. Singh, G. S. Aujla, and R. S. Bali, "Container-based load balancing for energy efficiency in software-defined edge computing environment," *Sustainable Computing: Informatics and Systems*, vol. 30, p. 100 463, 2021, ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2020.100463. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537920301876.

[89] F. Zhang, R. Deng, X. Zhao, and M. M. Wang, "Load balancing for distributed intelligent edge computing: A state-based game approach," *IEEE Transactions on Cognitive Communications and Networking*, vol. 7, no. 4, pp. 1066–1077, 2021. DOI: 10.1109/TCCN.2021.3087178.

[90] S. Sthapit, J. Thompson, N. M. Robertson, and J. R. Hopgood, "Computational load balancing on the edge in absence of cloud and fog," *IEEE Transactions on Mobile Computing*, vol. 18, no. 7, pp. 1499–1512, 2019. DOI: 10.1109/TMC.2018.2863301.

[91] X. Xu, S. Fu, Q. Cai, *et al.*, "Dynamic resource allocation for load balancing in fog environment," *Wireless Communications and Mobile Computing*, vol. 2018, p. 6 421 607, Apr. 2018, ISSN: 1530-8669. DOI: 10.1155/2018/6421607. [Online]. Available: https://doi.org/10.1155/2018/6421607.

[92] H. M. Shakir and J. Karimpour, "Systematic study of load balancing in fog computing in iot healthcare system," in *2021 International Conference on Advanced Computer Applications (ACA)*, 2021, pp. 132–137. DOI: 10.1109/ACA52198.2021.9626813.

[93] M. Asif-Ur-Rahman, F. Afsana, M. Mahmud, *et al.*, "Toward a heterogeneous mist, fog, and cloud-based framework for the internet of healthcare things," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4049–4062, 2019. DOI: 10.1109/JIOT.2018.2876088.

[94] S. Malik, K. Gupta, D. Gupta, *et al.*, "Intelligent load-balancing framework for fog-enabled communication in healthcare," *Electronics*, vol. 11, no. 4, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11040566. [Online]. Available: https://www.mdpi.com/2079-9292/11/4/566.

[95] H. A. Khattak, H. Arshad, S. u. Islam, *et al.*, "Utilization and load balancing in fog servers for health applications," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, p. 91, Apr. 2019, ISSN: 1687-1499. DOI: 10.1186/s13638-019-1395-3. [Online]. Available: https://doi.org/10.1186/s13638-019-1395-3.

[96] A. Mijuskovic, A. Chiumento, R. Bemthuis, A. Aldea, and P. Havinga, "Resource management techniques for cloud/fog and edge computing: An evaluation framework and classification," *Sensors*, vol. 21, no. 5, 2021, ISSN: 1424-8220. DOI: 10.3390/s21051832. [Online]. Available: https://www.mdpi.com/1424-8220/21/5/1832.

[97] N. Agrawal, "Dynamic load balancing assisted optimized access control mechanism for edge-fog-cloud network in internet of things environment," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 21, e6440, 2021. DOI: https://doi.org/10.1002/cpe.6440. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6440. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6440.

[98] R. Fantacci and B. Picano, "Performance analysis of a delay constrained data offloading scheme in an integrated cloud-fog-edge computing system," *IEEE Transactions on Vehicular Technology*, vol. 69, pp. 12 004–12 014, 2020.

[99] F. Alqahtani, M. Amoon, and A. A. Nasr, "Reliable scheduling and load balancing for requests in cloud-fog computing," *Peer-to-Peer Networking and Applications*, vol. 14, no. 4, pp. 1905–1916, Jul. 2021, ISSN: 1936-6450. DOI: 10.1007/s12083-021-01125-2. [Online]. Available: https://doi.org/10.1007/s12083-021-01125-2.

[100] W. Li, S. Cao, K. Hu, J. Cao, and R. Buyya, "Blockchain-enhanced fair task scheduling for cloud-fog-edge coordination environments: Model and algorithm," *Security and Communication Networks*, vol. 2021, p. 5 563 312, Apr. 2021, ISSN: 1939-0114. DOI: 10.1155/2021/5563312. [Online]. Available: https://doi.org/10.1155/2021/5563312.

[101] E. Batista, G. Figueiredo, and C. Prazeres, "Load balancing between fog and cloud in fog of things based platforms through software-defined networking," *Journal of King Saud University - Computer and Information Sciences*, 2021, ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2021.10.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157821002901.

[102] F. M. Talaat, M. S. Saraya, A. I. Saleh, H. A. Ali, and S. H. Ali, "A load balancing and optimization strategy (lbos) using reinforcement learning in fog computing environment," *Journal of Ambient Intelligence and Humanized Computing*, vol. 11, no. 11, pp. 1–16, Nov. 2020, ISSN: 1868-5145. DOI: 10.1007/s12652-020-01768-8. [Online]. Available: https://doi.org/10.1007/s12652-020-01768-8.

[103] A. AlOrbani and M. Bauer, "Load balancing and resource allocation in smart cities using reinforcement learning," in *2021 IEEE International Smart Cities Conference (ISC2)*, 445 Hoes Lane, Piscataway, NJ 08854-4141 USA: IEEE, 2021, pp. 1–7. DOI: 10.1109/ISC253183.2021.9562941.

[104] G. Proietti Mattia and R. Beraldi, "Online distributed scheduling in fog computing based on reinforcement learning for smart cities," *IEEE Transactions on Mobile Computing*, 2022.

[105] M. E. Aydin and E. Öztemel, "Dynamic job-shop scheduling using reinforcement learning agents,"

Robotics and Autonomous Systems, vol. 33, no. 2-3, pp. 169–178, 2000.

[106] L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu, and L. Li, "Delay-aware and energy-efficient computation offloading in mobile edge computing using deep reinforcement learning," *IEEE Transactions on Cognitive Communications and Networking*, pp. 1–1, 2021. DOI: 10.1109/TCCN.2021.3066619.

[107] M. K. Pandit, R. N. Mir, and M. A. Chishti, "Adaptive task scheduling in iot using reinforcement learning," *International Journal of Intelligent Computing and Cybernetics*, 2020.

[108] S. Nath and J. Wu, "Deep reinforcement learning for dynamic computation offloading and resource allocation in cache-assisted mobile edge computing systems," *Intelligent and Converged Networks*, vol. 1, no. 2, pp. 181–198, 2020. DOI: 10.23919/ICN.2020.0014.

[109] L. Mai, N.-N. Dao, and M. Park, "Real-time task assignment approach leveraging reinforcement learning with evolution strategies for long-term latency minimization in fog computing," *Sensors*, vol. 18, no. 9, 2018, ISSN: 1424-8220. DOI: 10.3390/s18092830. [Online]. Available: https://www.mdpi.com/1424-8220/18/9/2830.

[110] S. Bian, X. Huang, Z. Shao, and Y. Yang, "Neural task scheduling with reinforcement learning for fog computing systems," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6. DOI: 10.1109/GLOBECOM38437.2019.9014045.

[111] J. Zhang, H. Guo, and J. Liu, "A reinforcement learning based task offloading scheme for vehicular edge computing network," in *Artificial Intelligence for Communications and Networks*, S. Han, L. Ye, and W. Meng, Eds., Cham: Springer International Publishing, 2019, pp. 438–449, ISBN: 978-3-030-22971-9.

[112] H. Li, K. Ota, and M. Dong, "Deep reinforcement scheduling for mobile crowdsensing in fog computing," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, pp. 1–18, 2019.

[113] Q. Yang and P. Li, "Deep reinforcement learning based energy scheduling for edge computing," in *2020 IEEE International Conference on Smart Cloud (SmartCloud)*, 2020, pp. 175–180. DOI: 10.1109/SmartCloud49737.2020.00041.

[114] S. Park and Y. Yoo, "Real-time scheduling using reinforcement learning technique for the connected vehicles," in *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, 2018, pp. 1–5. DOI: 10.1109/VTCSpring.2018.8417770.

[115] T. Sen and H. Shen, "Machine learning based timeliness-guaranteed and energy-efficient task assignment in edge computing systems," in *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, 2019, pp. 1–10. DOI: 10.1109/CFEC.2019.8733153.

[116] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2020. DOI: 10.1109/TMC.2020.3017079.

[117] A. I. Orhean, F. Pop, and I. Raicu, "New scheduling approach using reinforcement learning for heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 292–302, 2018, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2017.05.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301521.

[118] J. Wang, L. Zhao, J. Liu, and N. Kato, "Smart resource allocation for mobile edge computing: A deep reinforcement learning approach," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, pp. 1529–1541, 2021.

[119] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. (Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, pp. 939–951, 2021.

[120] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, pp. 242–253, 2021.

[121] S. Yu, X. Chen, Z. Zhou, X. Gong, and D. Wu, "When deep reinforcement learning meets federated learning: Intelligent multitimescale resource management for multiaccess edge computing in 5g ultradense network," *IEEE Internet of Things Journal*, vol. 8, pp. 2238–2251, 2021.

[122] Y. He, Y. Wang, C. Qiu, Q. Lin, J. Li, and Z. Ming, "Blockchain-based edge computing resource allocation in iot: A deep reinforcement learning approach," *IEEE Internet of Things Journal*, vol. 8, pp. 2226–2237, 2021.

[123] Z. Safavifar, S. Ghanadbashi, and F. Golpayegani, "Adaptive workload orchestration in pure edge computing: A reinforcement-learning model," in *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2021, pp. 856–860. DOI: 10.1109/ICTAI52525.2021.00137.

[124] J. Santos, T. Wauters, B. Volckaert, and F. D. Turck, "Resource provisioning in fog computing through deep reinforcement learning," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 431–437.

[125] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Demand-driven deep reinforcement learning for scalable fog and service placement," *IEEE Transactions on Services Computing*, pp. 1–1, 2021. DOI: 10.1109/TSC.2021.3075988.

[126] X. Zhou, Z. Liu, M. Guo, J. Zhao, and J. Wang, "Sacc: A size adaptive content caching algorithm in fog/edge computing using deep reinforcement learning," *IEEE Transactions on Emerging Topics in*

Computing, pp. 1–1, 2021. DOI: 10.1109/TETC.2021.3115793.

[127] D. Lan, A. Taherkordi, F. Eliassen, Z. Chen, and L. Liu, "Deep reinforcement learning for intelligent migration of fog services in smart cities," in *Algorithms and Architectures for Parallel Processing*, M. Qiu, Ed., Cham: Springer International Publishing, 2020, pp. 230–244, ISBN: 978-3-030-60239-0.

[128] O. Houidi, D. Zeghlache, V. Perrier, *et al.*, "Constrained deep reinforcement learning for smart load balancing," in *2022 IEEE 19th Annual Consumer Communications Networking Conference (CCNC)*, 2022, pp. 207–215. DOI: 10.1109/CCNC49033.2022.9700657.

[129] Z. Wan, Z. Zhang, R. Yin, and G. Yu, "Kfiml: Kubernetes-based fog computing iot platform for online machine learning," *IEEE Internet of Things Journal*, pp. 1–1, 2022. DOI: 10.1109/JIOT.2022.3168085.

[130] A. Mseddi, W. Jaafar, H. Elbiaze, and W. Ajib, "Intelligent resource allocation in dynamic fog computing environments," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, 2019, pp. 1–7. DOI: 10.1109/CloudNet47604.2019.9064110.

[131] X. Chen, S. Leng, K. Zhang, and K. Xiong, "A machine-learning based time constrained resource allocation scheme for vehicular fog computing," *China Communications*, vol. 16, no. 11, pp. 29–41, 2019. DOI: 10.23919/JCC.2019.11.003.

[132] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[133] X. Xiong, K. Zheng, L. Lei, and L. Hou, "Resource allocation based on deep reinforcement learning in iot edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1133–1146, 2020. DOI: 10.1109/JSAC.2020.2986615.

[134] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Performance optimization in mobile-edge computing via deep reinforcement learning," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, 2018, pp. 1–6. DOI: 10.1109/VTCFall.2018.8690980.

[135] S. Sheng, P. Chen, Z. Chen, L. Wu, and Y. Yao, "Deep reinforcement learning-based task scheduling in iot edge computing," *Sensors*, vol. 21, no. 5, 2021, ISSN: 1424-8220. DOI: 10.3390/s21051666. [Online]. Available: https://www.mdpi.com/1424-8220/21/5/1666.

[136] Y. Wei, Z. Zhang, F. R. Yu, and Z. Han, "Joint user scheduling and content caching strategy for mobile edge networks using deep reinforcement learning," in *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2018, pp. 1–6. DOI: 10.1109/ICCW.2018.8403711.

[137] S. Huang, B. Lv, R. Wang, and K. Huang, "Scheduling for mobile edge computing with random user arrivals—an approximate mdp and reinforcement learning approach," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7735–7750, 2020. DOI: 10.1109/TVT.2020.2990482.

[138] Y. Zhang, Z. Zhou, Z. Shi, L. Meng, and Z. Zhang, "Online scheduling optimization for dag-based requests through reinforcement learning in collaboration edge networks," *IEEE Access*, vol. 8, pp. 72 985–72 996, 2020. DOI: 10.1109/ACCESS.2020.2987574.

[139] A. Luckow, K. Rattan, and S. Jha, "Exploring task placement for edge-to-cloud applications using emulation," in *2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC)*, 2021, pp. 79–83. DOI: 10.1109/ICFEC51620.2021.00019.

[140] Z. Tang, W. Jia, X. Zhou, W. Yang, and Y. You, "Representation and reinforcement learning for task scheduling in edge computing," *IEEE Transactions on Big Data*, pp. 1–1, 2020. DOI: 10.1109/TBDATA.2020.2990558.

[141] K. Wang, X. Wang, X. Liu, and A. Jolfaei, "Task offloading strategy based on reinforcement learning computing in edge computing architecture of internet of vehicles," *IEEE Access*, vol. 8, pp. 173 779–173 789, 2020. DOI: 10.1109/ACCESS.2020.3023939.

[142] M. Li, J. Gao, L. Zhao, and X. Shen, "Deep reinforcement learning for collaborative edge computing in vehicular networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 6, no. 4, pp. 1122–1135, 2020. DOI: 10.1109/TCCN.2020.3003036.

[143] D. Zeng, L. Gu, S. Pan, J. Cai, and S. Guo, "Resource management at the network edge: A deep reinforcement learning approach," *IEEE Network*, vol. 33, no. 3, pp. 26–33, 2019. DOI: 10.1109/MNET.2019.1800386.

[144] B. Huang, Y. Xiang, D. Yu, J. Wang, Z. Li, and S. Wang, "Reinforcement learning for security-aware workflow application scheduling in mobile edge computing," *Security and Communication Networks*, vol. 2021, p. 5 532 410, May 2021, ISSN: 1939-0114. DOI: 10.1155/2021/5532410. [Online]. Available: https://doi.org/10.1155/2021/5532410.

[145] F. Hu, Y. Deng, W. Saad, M. Bennis, and A. H. Aghvami, "Cellular-connected wireless virtual reality: Requirements, challenges, and solutions," *IEEE Communications Magazine*, vol. 58, no. 5, pp. 105–111, 2020. DOI: 10.1109/MCOM.001.1900511.

[146] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[147] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[148] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *4th International Conference on Learning Representations,*

ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.06530.

[149] A. G. Howard, M. Zhu, B. Chen, *et al.*, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, 2017. arXiv: 1704.04861 [cs.CV].

[150] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018.

[151] L. N. Huynh, R. K. Balan, and Y. Lee, "Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices," in *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, ACM, 2016, pp. 25–30.

[152] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ACM, 2017, pp. 82–95.

[153] N. D. Lane, S. Bhattacharya, P. Georgiev, *et al.*, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*, IEEE Press, 2016, p. 23.

[154] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, IEEE, 2018, pp. 1421–1429.

[155] S. Tuli, N. Basumatary, and R. Buyya, "Edgelens: Deep learning based object detection in integrated iot, fog and cloud computing environments," *arXiv preprint arXiv:1906.11056*, 2019.

[156] J. Wan, B. Chen, S. Wang, M. Xia, D. Li, and C. Liu, "Fog computing for energy-aware load balancing and scheduling in smart factory," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4548–4556, 2018. DOI: 10.1109/TII.2018.2818932.

[157] A. Hazra, M. Adhikari, T. Amgoth, and S. N. Srirama, "Fog computing for energy-efficient data offloading of iot applications in industrial sensor networks," *IEEE Sensors Journal*, vol. 22, no. 9, pp. 8663–8671, 2022. DOI: 10.1109/JSEN.2022.3157863.

[158] M. Adhikari and H. Gianey, "Energy efficient offloading strategy in fog-cloud environment for iot applications," *Internet of Things*, vol. 6, p. 100 053, 2019, ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2019.100053. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660519300265.

[159] A. Gougeon, B. Camus, and A.-C. Orgerie, "Optimizing green energy consumption of fog computing architectures," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 75–82. DOI: 10.1109/SBAC-PAD49847.2020.00021.

[160] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.

[161] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar, "Squeezing deep learning into mobile and embedded devices," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 82–88, 2017, ISSN: 1558-2590. DOI: 10.1109/MPRV.2017.2940968.

[162] A. Ignatov, N. Kobyshev, R. Timofte, K. Vanhoey, and L. Van Gool, "Dslr-quality photos on mobile devices with deep convolutional networks," in *The IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[163] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu, "Deep learning for sensor-based activity recognition: A survey," *Pattern Recognition Letters*, vol. 119, pp. 3–11, 2019.

[164] V. Radu, N. D. Lane, S. Bhattacharya, C. Mascolo, M. K. Marina, and F. Kawsar, "Towards multimodal deep learning for activity recognition on mobile devices," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, ACM, 2016, pp. 185–188.

[165] Y. Rivenson, H. Ceylan Koydemir, H. Wang, *et al.*, "Deep learning enhanced mobile-phone microscopy," *Acs Photonics*, vol. 5, no. 6, pp. 2354–2364, 2018.

[166] B. Amos, B. Ludwiczuk, M. Satyanarayanan, *et al.*, "Openface: A general-purpose face recognition library with mobile applications," *CMU School of Computer Science*, vol. 6, 2016.

[167] X. Ran, H. Chen, Z. Liu, and J. Chen, "Delivering deep learning to mobile devices via offloading," in *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, ACM, 2017, pp. 42–47.

[168] R. Xie, X. Jia, L. Wang, and K. Wu, "Energy efficiency enhancement for cnn-based deep mobile sensing," *IEEE Wireless Communications*, vol. 26, no. 3, pp. 161–167, Jun. 2019, ISSN: 1558-0687. DOI: 10.1109/MWC.2019.1800321.

[169] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?" In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, ACM, 2015, pp. 117–122.

[170] R. Girshick, "Fast r-cnn," in *The IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015.

[171] W. Liu, D. Anguelov, D. Erhan, *et al.*, "Ssd: Single shot multibox detector," in *European conference on computer vision*, Springer, 2016, pp. 21–37.

[172] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[173] S. Yi, C. Li, and Q. Li, "A survey of fog computing: Concepts, applications and issues," in *Proceedings of the 2015 workshop on mobile big data*, ACM, 2015, pp. 37–42.

[174] J. Pedoeem and R. Huang, "Yolo-lite: A real-time object detection algorithm optimized for non-gpu computers," *arXiv preprint arXiv:1811.05588*, 2018.

[175] J. Hosang, R. Benenson, and B. Schiele, "Learning non-maximum suppression," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017.

[176] J. Redmon, *Darknet: Open source neural networks in c*, http://pjreddie.com/darknet/, 2013–2016.

[177] W.-F. Alliance, "Wi-fi 6: High performance, next generation wi-fi," 2018, Tech. Rep., 2018. [Online]. Available: https://www.wi-fi.org/download.php?file=/sites/default/files/private/Wi-Fi_6_White_Paper_20181003.pdf.

[178] H. Feng, G. Mu, S. Zhong, P. Zhang, and T. Yuan, "Benchmark analysis of yolo performance on edge intelligence devices," *Cryptography*, vol. 6, no. 2, 2022, ISSN: 2410-387X. DOI: 10.3390/cryptography6020016. [Online]. Available: https://www.mdpi.com/2410-387X/6/2/16.

[179] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: Quantifying the carbon footprint of computation," *Advanced Science*, vol. 8, no. 12, p. 2 100 707, 2021. DOI: https://doi.org/10.1002/advs.202100707. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202100707. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/advs.202100707.

[180] P. Cooper, "The absorption of radiation in solar stills," *Solar Energy*, vol. 12, no. 3, pp. 333–346, 1969, ISSN: 0038-092X. DOI: https://doi.org/10.1016/0038-092X(69)90047-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0038092X69900474.

[181] S. A. Kalogirou, *Solar energy engineering: processes and systems*. Academic press, 2013.

[182] K. Doucet and J. Zhang, "Learning cluster computing by creating a raspberry pi cluster," in *Proceedings of the SouthEast Conference*, ser. ACM SE '17, Kennesaw, GA, USA: Association for Computing Machinery, 2017, pp. 191–194, ISBN: 9781450350242. DOI: 10.1145/3077286.3077324. [Online]. Available: https://doi.org/10.1145/3077286.3077324.

[183] K. Doucet and J. Zhang, "The creation of a low-cost raspberry pi cluster for teaching," in *Proceedings of the Western Canadian Conference on Computing Education*, ser. WCCCE '19, Calgary, AB, Canada: Association for Computing Machinery, 2019, ISBN: 9781450367158. DOI: 10.1145/3314994.3325088. [Online]. Available: https://doi.org/10.1145/3314994.3325088.

[184] M. d'Amore, R. Baggio, and E. Valdani, "A practical approach to big data in tourism: A low cost raspberry pi cluster," in *Information and Communication Technologies in Tourism 2015*, I. Tussyadiah and A. Inversini, Eds., Cham: Springer International Publishing, 2015, pp. 169–181, ISBN: 978-3-319-14343-9.

[185] J. Saffran, G. Garcia, M. A. Souza, *et al.*, "A low-cost energy-efficient raspberry pi cluster for data mining algorithms," in *Euro-Par 2016: Parallel Processing Workshops*, F. Desprez, P.-F. Dutot, C. Kaklamanis, *et al.*, Eds., Cham: Springer International Publishing, 2017, pp. 788–799, ISBN: 978-3-319-58943-5.

[186] A. Mappuji, N. Effendy, M. Mustaghfirin, F. Sondok, R. P. Yuniar, and S. P. Pangesti, "Study of raspberry pi 2 quad-core cortex-a7 cpu cluster as a mini supercomputer," in *2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE)*, 2016, pp. 1–4. DOI: 10.1109/ICITEED.2016.7863250.

[187] P. Abrahamsson, S. Helmer, N. Phaphoom, *et al.*, "Affordable and energy-efficient cloud computing clusters: The bolzano raspberry pi cloud cluster experiment," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2, 2013, pp. 170–175. DOI: 10.1109/CloudCom.2013.121.

[188] M. F. Cloutier, C. Paradis, and V. M. Weaver, "A raspberry pi cluster instrumented for fine-grained power measurement," *Electronics*, vol. 5, no. 4, p. 61, 2016. DOI: 10.3390/electronics5040061.

[189] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, "The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures," in *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, 2013, pp. 108–112. DOI: 10.1109/ICDCSW.2013.25.

[190] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. O'Brien, "Iridis-pi: A low-cost, compact demonstration cluster," *Cluster Computing*, vol. 17, no. 2, pp. 349–358, 2014. DOI: 10.1007/s10586-013-0282-7. [Online]. Available: https://doi.org/10.1007/s10586-013-0282-7.

[191] E. Wilcox, P. Jhunjhunwala, K. Gopavaram, and J. Herrera, "Pi-crust: A raspberry pi cluster implementation," Technical report, Texas A&M University, Tech. Rep., 2015.

[192] D.-N. Le, S. Pal, and P. K. Pattnaik, "Openfaas," *Cloud Computing Solutions: Architecture, Data Storage, Implementation and Security*, pp. 287–303, 2022.

[193] F. Rossi, S. Falvo, and V. Cardellini, "Gofs: Geo-distributed scheduling in openfaas," in *2021 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2021, pp. 1–6.

[194] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "Ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[195] X. Liu, L. Fan, J. Xu, *et al.*, "Fogworkflowsim: An automated simulation toolkit for workflow performance evaluation in fog computing," in *2019 34th IEEE/ACM*

International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 1114–1117.

[196] I. Lera, C. Guerrero, and C. Juiz, "Yafs: A simulator for iot scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91 745–91 758, 2019. DOI: 10.1109/ACCESS.2019.2927895.

[197] A. W. Malik, T. Qayyum, A. U. Rahman, M. A. Khan, O. Khalid, and S. U. Khan, "Xfogsim: A distributed fog resource management framework for sustainable iot services," *IEEE Transactions on Sustainable Computing*, vol. 6, no. 4, pp. 691–702, 2021. DOI: 10.1109/TSUSC.2020.3025021.

[198] T. Qayyum, A. W. Malik, M. A. Khan Khattak, O. Khalid, and S. U. Khan, "Fognetsim++: A toolkit for modeling and simulation of distributed fog environment," *IEEE Access*, vol. 6, pp. 63 570–63 583, 2018. DOI: 10.1109/ACCESS.2018.2877696.

[199] Intel, "Atx motherboard specification, v2.2," Tech. Rep. [Online]. Available: https://web.archive.org/web/20120725150314/http://www.formfactors.org/developer/specs/atx2_2.pdf.

[200] D. Pliatsios, P. Sarigiannidis, S. Goudos, and G. K. Karagiannidis, "Realizing 5g vision through cloud ran: Technologies, challenges, and trends," *EURASIP Journal on Wireless Communications and Networking*, vol. 2018, no. 1, p. 136, May 2018, ISSN: 1687-1499. DOI: 10.1186/s13638-018-1142-1. [Online]. Available: https://doi.org/10.1186/s13638-018-1142-1.

[201] R. Beraldi and H. Alnuweiri, "Exploiting power-of-choices for load balancing in fog computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*, Piscataway, New Jersey, US: IEEE, Jun. 2019, pp. 80–86. DOI: 10.1109/ICFC.2019.00019.