Online Decentralized Scheduling in Fog Computing for Smart Cities based on Reinforcement Learning

Gabriele Proietti Mattia, Roberto Beraldi

***Abstract*—Fog Computing is a widely adopted paradigm that allows distributing the computation in a geographic area. This makes it possible to implement time-critical applications and opens the study to a series of solutions that permit smartly organizing the traffic among a set of fog nodes, which constitute the core of the Fog Computing paradigm. As a typical smart city setting is subject to a continuous change in traffic conditions, it is necessary to design algorithms that can manage all the computing resources by properly distributing the traffic among the nodes in an adaptive way. In this paper, we propose a cooperative and decentralized algorithm based on Reinforcement Learning that is able to perform online scheduling decisions among fog nodes. This can be seen as an improvement over the power-of-two random choices paradigm used as a baseline. By showing results from our delay-based simulator and then from our framework "P2PFaaS" installed on 12 Raspberry Pis, we show how our approach maximizes the rate of the tasks executed within the deadline, outperforming the power-of-two random choices both in a fixed load condition and with traffic extracted from a real smart city scenario.**

***Index Terms*—Fog computing, Scheduling, Real-time, Reinforcement Learning, Smart Cities**

I. Introduction

FOG Computing [1] is a well-known computing paradigm that is generally used for distributing the computation in a geographic domain in order to deploy the applications as near as possible to end users. Indeed, when the tasks that the application should carry out are strict in their deadlines, a cloud approach could not be feasible. For instance, when we refer to shared Virtual or Augmented Reality (VR/AR) [2] experiences or real-time monitoring [3], [4]. Distributing the load involves the setup of different computing nodes, called "fog nodes", which, for example, can be spread across a smart city. To make load distribution effective, it is conceptually convenient to take the point of view where an abstract service exists, and it is made available to the entire system of fog nodes, although each node hosts a replica of the same service. Data generated by fog nodes can exploit the actual service of any other node. Aside from the delay, these nodes are functionally equivalent, but they can be heterogeneous in their computational power.

In our setting, the users request the service to the nearest fog node in the form of a sequence of task execution requests, which may have a payload attached. This model particularly fits the FaaS (Function-as-a-Service) paradigm of serverless computing. As anticipated, since we target the aforementioned applications, we remark that tasks must be completed within a precise deadline, which can vary according to the specific application. Since the traffic to the fog nodes, which are generally heterogeneous [1], also varies over time, there may be situations in which a node is no longer able to offer the service in such a way the requests meet the given deadline. With a network of fog nodes that can communicate with each other, having a central entity that manages how the load must be partitioned is not a scalable solution. Moreover, this entity must be chosen among the nodes, and this introduces a series of problems that are well-addressed in literature. For these reasons, we envision the design of a cooperative, decentralized, and distributed scheduling algorithm that decides for every single task that arrives at a given node if the task must be executed locally or forwarded to some neighbor, whose load may be lower and increasing the probability for the task to meet the deadline. As may be known to the reader, task execution latency and the nodes' load are bonded by a non-decrescent function.

In the context of online, decentralized and distributed scheduling, a well-known approach that is proven to perform efficiently is the power-of-two random choices paradigm [5], where every scheduling decision (that is always made 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 a random-probed node. Since executing a probing for each request is not always the best choice, adding a control threshold to decide when to trigger a new probing request [6] is shown to be an effective way to increase 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, namely, 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. Finally, it doesn't take task heterogeneity into account (iii). This work aims 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-two random choices paradigm, we also study a scheduling policy that directly forwards

G. Proietti Mattia and R. Beraldi are with the Department of Computer, Control and Management Engineering "Antonio Ruberti", Sapienza University of Rome, Italy (e-mail: proiettimattia@diag.uniroma1.it, beraldi@diag.uniroma1.it).

This work was supported by project "FogAware" (CUP: B89C20002240001) granted by Sapienza University of Rome.

Manuscript received August 29, 2023; revised March 18, 2024.

the tasks to a specific node without probing. Beyond the power-of-two random choices, other approaches, which, for example, involve heuristics, may still not fit distributed online scheduling based on deadlines since they may be able to adapt the scheduling policy, but they are not good when the state of the other nodes must be predicted. This is because the best scheduling decision for meeting the deadline, which is intuitively to assign the task to the faster and least loaded node, requires knowing not only the exact state of all the neighbors but also their performance factors due to the fact that nodes are heterogeneous. Reinforcement Learning, unlike heuristics and randomized approaches, allows inferring the state of the other neighbors given the current state of the node and the experience without explicitly asking for it and dynamically changing the scheduling policy.

In our 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 execution is completed, 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 condition: the task has been completed by the defined deadline. We are not interested in creating more complex assignments of the reward since once the task is executed within the deadline, nodes have no advantages over tasks that are completed earlier than the deadline. This is a typical case of video frames processing tasks.

We can summarize the main contributions of this work as follows.

- Design of a decentralized RL-based algorithm that selects the best scheduling decision according to the current load situation; this is a step ahead of the power-of-two random choices approach since it allows for more complex policies, it is dynamic over time, and it can deal with different kinds of time-constrained (i.e. with deadline requirements) tasks, moreover, our approach (i) models the deadline as rewards, (ii) combines SARSA average reward and Q-Table for reducing the inference and training time of the model.
- Study of a geographic setting that 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 that shows the efficiency of the algorithm in a previously defined geographic environment compared to the classic power-of-two random choices strategy.
- Results from a pseudo-real deployment with our prototype framework “P2PFaaS” in a rack of 12 Raspberry Pis which shows that the algorithm achieves the same performance indicators even outside the simulation.

The rest of this paper is organized as follows. In Section II, we present some related works. In Section III, we define the system model, while in Section IV, we describe the Reinforcement Learning approach we propose. Then, in Section V we show the results of the proposed algorithm in a simulated environment, while in Section VI, we present a real implementation infrastructure of the algorithm and the results of the tests in that environment. Finally, we draw conclusions in Section VII.

II. Related Work

The main research area in which this work lies is the approach to the problem of scheduling and load balancing in the Fog or Edge computing environments by means of 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 machine learning approach was introduced in 1999 by [7]. From there, the scenario changed radically with the development of new machine learning techniques and with the spread of distributed systems, which concretized in the Fog and in the Edge computing layers [1].

In this section, we first describe the works that target our problem but are based on heuristics and mathematical modeling. Then, we specifically focus on works that use Reinforcement Learning.

A. Heuristics and model-based approaches

The scheduling task in Fog Computing is widely studied in literature [8]. In particular, these studies focus on the assignment of task execution to a particular node in the system but follow specific constraints, which essentially regard delay and energy. Azizi et al. in [9] propose a deadline-aware scheduling solution based on semi-greedy approaches that aim to minimize deadline violations. However, in our work, we suppose that users are able to specify a tolerance limit on the deadline that matches the minimum frame rate acceptable. Hassan et al. [10] instead propose a heuristic called “MinRes” that has the target of minimizing the response time. However, the tests are conducted only in simulations. Focusing on smart factories, Zhou et al.in [11] propose a solution for task allocation based on a genetic scheduling algorithm. Here, the target is to optimize task execution time and, at the same time, balance the resources of the clusters. The solution is tested only in simulations and the model does not follow an online scheduling behavior, which is specifically our case. Finally, Abdel-Basset et al. in [12] propose a task scheduling meta-heuristic that instead targets energy but uses the Harris Hawks optimization algorithm for improving the QoS. However, the proposed solution does not address streaming processing but supposes that tasks arrive in batches and then are scheduled. In our work, instead, we specifically rely on online scheduling for targeting users, which continuously generates tasks to be executed, like in the case of video frames to be analyzed. Finally, Li et al. in [13] proposes a smart resource partitioning based on the popularity rank of a service deployed in the Fog Computing layer. However, the approach follows a different task model, and the scheduling is not done online.

B. Reinforcement Learning based studies

In Table I, 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
Ale et al. [14]	✓	✓	-	-	Q-Learning	Two DNNs
Pandit et al. [15]	✓	-	-	-	Q-Learning	Two-level DNNs
Nath et al. [16]	-	-	-	-	DDPG	Two DNNs
Mai et al. [17]	✓	✓	-	-	Custom	Single DNN
Bian et al. [18]	✓	-	-	-	Custom	RNN
Zhang et al. [19]	✓	-	-	-	Q-Learning	Two DNNs
Li et al. [20]	-	-	✓	-	Q-Learning	Single DNN
Yang et al. [21]	-	-	-	-	Q-Learning	Single DNN
Park et al. [22]	✓	✓	-	-	Q-Learning	Q-Table
Sen et al. [23]	✓	✓	-	-	Q-Learning	Q-Table
Tuli et al. [24]	-	✓	-	-	A3C	Residual RNN
Orhean et al. [25]	-	-	-	-	Q-Learning, Sarsa	Q-Table
Aydin et al. [7]	-	-	-	-	Actor-Critic	Residual RNN
Wang et al. [26]	-	-	-	-	Q-Learning	Two DNNs
Yu et al. [27]	-	-	-	-	Custom	Two DNNs
He et al. [28]	-	-	-	-	Actor-Critic	Custom
Safavifar et al. [29]	-	-	-	-	Q-Learning	Q-Table
Santos et al. [30]	-	-	-	-	Q-Learning	DNNs
Sami et al. [31]	-	-	-	-	Q-Learning	DNNs
Zhou et al. [32]	-	-	-	-	Actor-Critic	DNNs
Lan et al. [33]	-	-	-	-	DDPG	Two DNNs
Wang et al. [34]	-	-	-	-	MRL	Custom
Alorbani et al. [35]	✓	-	-	-	Q-Learning	Q-Table
Houidi et al. [36]	-	-	✓	-	DDPG	Custom
Talaat et al. [37]	-	-	-	-	Q-Learning	Q-Table
Wang et al. [38]	✓	-	✓	-	Q-Learning	Q-Table
our work	✓	✓	✓	✓	Sarsa	Lin. Approx. + Q-Table

TABLE I
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.

As in our work, [25] 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 differently from this work, they do not consider the average reward approach that best fits a continuous learning task. A similar geographic approach, by using a similar traffic dataset is followed by [38], instead [35] focuses on smart cities but does not use a real traffic dataset, and [36] which uses a real node topology.

In [14], the authors present a Deep Reinforcement Learning approach in a MEC environment that is based on Q-Learning for selecting the best edge server for offloading in order to minimize the energy consumption (also studied in [21], [23], [30]) while at the same time maximizing 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 our work, that approach is not fully decentralized and requires a central entity that collects information about the state of each node and takes the decision (as also studied in [29]). However, the followed approach is very common and also used in [16], [19], [26] and [33]. Other works instead rely, for example, on Meta Reinforcement Learning (MRL) [34], blockchain [28], or even a genetic algorithm [37].

Pandit *et al.*, in [15], 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 most suitable fog node to which to schedule the task. In our work, instead, we do not rely on a neural network, since we keep the state as small as possible since when dealing with deadlines, the inference time is critical.

The approach followed by [16], but in the context of MEC cells (as in [27]), 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 discretization in the standard Q-Learning approach. In our work, instead, the state is discrete, and we rely on a basic RL approach for reducing the inference time.

In [18], 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.* [24] uses the Asynchronous Advantage Actor Critic (A3C) algorithm in an 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 since they used simulators. Instead, [32] uses the Actor-Critic paradigm for a size adaptive caching scheme.

In a broader sense of schedule, other works are instead focused on resource allocation [39], [40], [26], [30], [31], but the task model does not fit the one that is studied in this paper.

III. SYSTEM MODEL AND PROBLEM DEFINITION

In order to reach our goal of adaptability and optimality, we follow the approach presented in [41], and we frame our problem as a Markov Decision Process (MDP), which is solved using Reinforcement Learning (RL). We use a model-free approach with the advantage that it doesn't require 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 has been ported on a real deployment. To proceed with our discussion, we need preliminarily to identify the two main entities of RL: the environment and the agent.

A. 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

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 neighbor 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 trigger the training process
Fog Nodes	
K	Maximum number of parallel executing tasks
K_e	Maximum length of the execution queue Q_e
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 II
LIST OF SYMBOLS USED

total rate of the arriving requests to a node i to be λ_i req/s. One example of an application scenario for such tasks is Virtual Reality (VR), where a user needs to execute compute-intensive tasks like recognizing 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 [2]. They also have a physical size b, which 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 saying that it is capable of executing K_i tasks at a time in time-sharing with no queue, that is a mechanism closer to reality.

The total load to a specific node i is:

$$\rho_i = \frac{\lambda_i}{\mu} \tag{1}$$

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:

$$d_t = \frac{b}{r} \tag{2}$$

Moreover, node A can probe another node B, meaning that A can ask B its current queue length in order to make a scheduling decision. The queue length of a node is usually referring to its current "state". We remark that there's no broadcast of states between the nodes, the only way for a node to know the state of another node is to explicitly ask for it with the probing.

B. Agent

Upon each task request's 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 be performed. The action $a \in \mathcal{A}$ to take is determined by a function $\pi(s)$, called *policy*, of the current observed state $s \in \mathcal{S}$.

$$\pi : S \rightarrow \mathcal{A} \tag{3}$$

1) Observed states: The agent running on node F_i is able to observe its current state, which refers 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 made by observing the queue length of the current node. As stated in the introduction (Section I), since the environment is fully distributed we cannot have updated information about the state of the other nodes, each node only knows who are their neighbors, and in our specific case, the topology is a fully connected graph. In any case, the agent should be able to perform the correct decision, this is because the load condition 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 [42] is used to represent it as a vector $v \in \mathbb{N}^8$.

2) 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 directly forwarding a task to the neighbor j without probing, obviously with $j \neq i$.

It is worth reminding 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, $k_i^t = K_i$). Moreover, when a task is forwarded, the remote node does not apply the learning process, indeed the task is added the to internal execution queue Q_e if there is room otherwise it is rejected.

C. 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:

$$R_j(s,a) = \begin{cases} 1 & \text{if } W \leq T \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The agent cannot know the reward until a task has completed its execution path and in the meantime, 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 has been completed. This is discussed in Section IV.

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

D. Delay model

Dealing with deadlines requires a fine-grained model of the delays. In our study, the environment is both simulated and represented by a real-world deployment. While we leave the discussion about how the algorithm has been implemented in the real deployment in Section VI, we now deepen how the simulator has been conceived to represent a great part of the delays that can exist in the described environment.

In the simulator, the environment is seen 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 a 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 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 1. Suppose that a client sends a task request to node i that can decide to forward the task to node j. The task enters the transmission queue Q_t:

1) when the client transmits it to a node i, in this case, after the scheduling decision is taken, if it includes the probing, then the task will be added to the probing queue Q_p;
2) when it is transmitted from a node i to a node j;
3) when it returns from a node j to a node j;
4) when it returns to the client from node i.

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

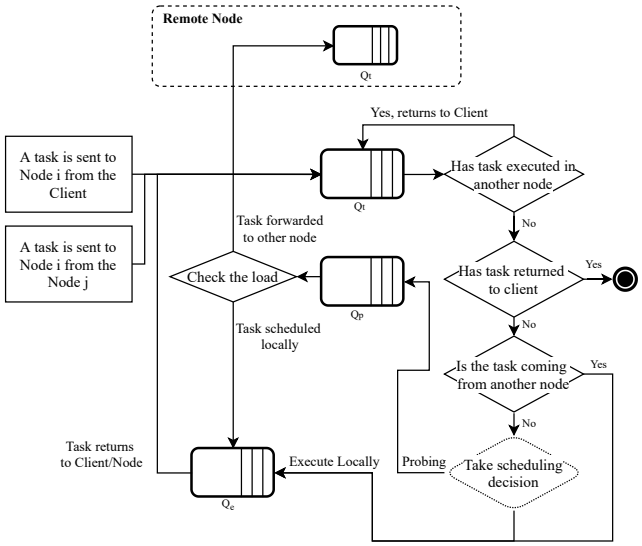


Fig. 1. The logic of the delay model used in the simulator.

The total time of a task to be executed, from the client's perspective, is the summation of all the time spent in all the queues during its entire execution path, it is referred to as W and it is measured in seconds.

E. Geographic Traffic

A peculiar characteristic of Fog Computing is that nodes can be positioned in a geographic scenario [1]. In order to evaluate the adaptivity of our solution in real traffic conditions, 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 in the 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, considering 1 km of radius for the service to be available, this is in line with the capability of an RRU to which has been attached a computing node. We estimated the taxi traffic by dividing the day into 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 trip. We used the first three months of data by averaging the daily traffic within each time slot for every fog node.

By normalizing in the range between 0 and 0.9, the final traffic distribution is represented in Figure 2. This range is given for simplicity and derives from a reasonable assumption

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

that fog nodes never saturate, leaving the opposite case as future work. Moreover, the final curves have been smoothed with the Savitzky-Golay filter, using an order 4 polynomial and a window of size 17.

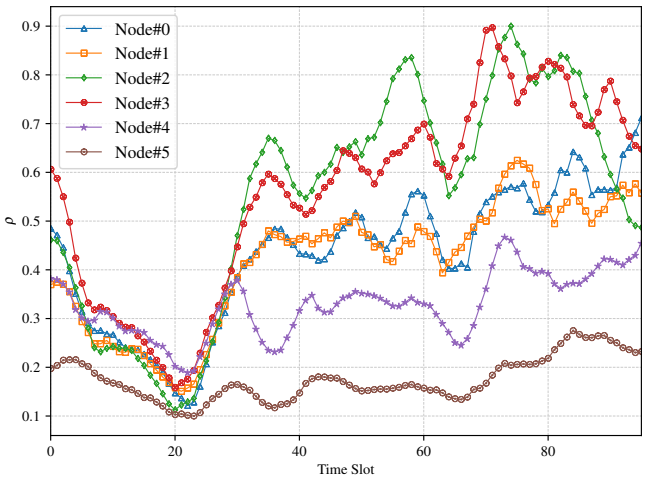


Fig. 2. The average distribution of the traffic during the day for the picked fog nodes.

The number of the nodes and their position has been chosen arbitrarily without loss of generality. Indeed, the load traces are fundamentally heterogeneous and variable in time, and this allows us to test the adaptiveness of the algorithms in dynamic traffic situations. Then, the number of nodes is kept relatively low because we envision the algorithm to be applied to a specific and limited district of the city. For addressing scalability, multiple sets of nodes can be installed in multiple districts and use the same proposed solution. This is because, due to the deadline constraints, we do not expect user requests to be migrated too far distant from the user position.

The final result of this study has been used both in the simulation environment (Section V) and in the (pseudo) real deployment (Section VI) by setting the load to a specific fog node i (ρ_i) to be equal to the value of the respective curve in that precise moment of the simulation.

IV. Online scheduling decisions with RL

The final objective of the agent is 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 continuous learning task.

In a continuing learning task, it is not useful to discount future rewards, but it is better to consider 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 [42]:

$$q_*(s,a) = \sum_{r,s'} p(s', r|s, a) \left[r - \max_{\pi} r(\pi) + \max_{a'} q_*(s', a') \right] \tag{5}$$

Where $r(\pi)$ is a function that returns the average reward of the policy π. Since we assume that the underlying model of the system is not known, we recur to the temporal difference approach for finding the policy. The classic strategy based on the temporal difference approach is Q-Learning, however, Q-Learning is an off-policy method. This means that the next action is chosen by not following the current learned policy. This, in our specific context, increases the convergence times and makes the solution unstable. For all of these reasons, we choose the Sarsa approach which is an on-policy method [42]. At a 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 [42]:

$$\delta_t = R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \vec{w}_t) - \hat{q}(S_t, A_t, \vec{w}_t) \tag{6}$$

Therefore the δ_t describes that from the immediate reward R_{t+1} we subtract the current average reward $\bar{R}_{t+1}$ and we sum the value of the next action, chosen by using the current policy, in the next state. 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 III-B. The final strategy for learning the policy in the simulator is called Differential Semi-Gradient Sarsa (Algorithm 2).

In the experimental setting (section VI), 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 Q-Table as the final function approximation mechanism, and in practice, the cells of the table have been updated according to the following Equation 7.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Delta_t \tag{7}$$

The Δ_t can be written by following the Sarsa approach and still considering the average reward $\bar{R}_{t+1}$, as in the following Equation 7.

$$\Delta_t = [R_{t+1} - \bar{R}_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{8}$$

As in the linear approximation solution, we also apply a discount factor to the average reward, which becomes as in Equation 9.

$$\bar{R}_{t+1} = \bar{R}_t + \beta \Delta_t \tag{9}$$

However, in our specific setting, we do not have a real notion of immediate reward because 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, the training is then started by following a FIFO order. This is explained in the following algorithm's description.

The Algorithm 1 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 III) 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 than 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 1 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.getNeighbors())
    if RandomNode.getLoad() $<$ Node.getLoad() **then**
      forwardTo(RandomNeighbor, Task)
    else
      Node.execute(Tasks)
    end if
  else
    Node $\leftarrow$ pickNode(a)
    forwardTo(Node, Tasks)
  end if

Every time that a task completes its execution (that means that the result payload of the task is returned to the client), whether it is local or remote, Algorithm 2 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 and 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 in the case of simulation, while in the real deployment, we update the Q-Table according to Equation 7.

Algorithm 2 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

A. Complexity and Convergence Analysis

In this section, we give an analysis of the convergence and complexity of the proposed solution. In our approach, as we described in Section IV, we start training the model after we have exactly Z tasks that have been completed, and therefore, for each task j, we have a triple which contains the state, the action and the reward, namely (s_j, a_j, r_j).

First of all, regarding the complexity of the training algorithm, we anticipate that we used the Tile Code technique in simulations and the Q-Table approach in real experiments with Raspberry Pi boards. The usage of different methods is due to software libraries' dependencies. In simulations, we used a pre-written solution to the problem, while in the real environment, we rewrote the training code from scratch, and we chose the Q-Table approach for function approximation. In both cases, the weights update algorithm requires a constant number of operations, independently from the input, and therefore its cost is $O(1)$. This is because, in the Tile Code technique weights are arranged through a hash table, and also in the Q-Table the access to the weights by index. Moreover, we highlight that by not using a Deep Neural Network for training the model we do not have the cost of the backpropagation for updating the weights.

Inference and training time are critical in this particular situation. Indeed, when a task arrives to a node, the time needed for deciding the action will necessarily consume part of the available time for executing the tasks, which always have a deadline. This means that the inference must be as fast as possible since tasks may have deadlines in the order of tens of milliseconds. Moreover, the training time can also have an impact on the convergence of the model, and this is explained in Figure 3. Suppose that at a certain time t_Z, all the tasks have been completed (even in a different order with respect to the one that they arrived). This means that at time t_Z, the training can start, and triple by triple, the weights are accordingly updated. Unfortunately, during this period, other new tasks arrive at the node. For these tasks, the node necessarily has to decide by using an RL model that is not fully updated. We call this period Δt_{stale}, and we remark that it is critical only to the convergence time of the training algorithm and not to the convergence itself. Indeed, the assumptions that guarantee the convergence as in Chapter 10.3 of [42] are not violated. Moreover, in order to avoid saturating the queue that keeps the training triples, $\Delta t_{\text{stale}} < t_Z$, in other words, the time needed for training the model with the Z tasks triples must be lower than the time for completing all the Z tasks. For this reason, Z cannot be too small, for example, to be equal to one, as we tested empirically, and we treat the Z parameter as a learning hyperparameter of the system.

V. Simulation Results

The results that we are going to present in this section follow 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 μ_i. We suppose that nodes are connected with a link of 1Gbps, the payload of each task is 100kb, and the probing

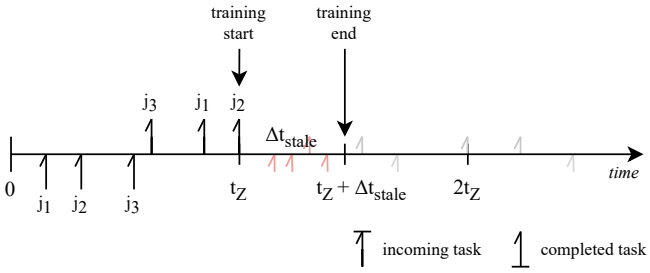


Fig. 3. Time diagram, which involves the job arrival, execution, and training of the RL model. The Δt_{stale} is the time between Z tasks completed their execution and the time in which the training of the model ends. In the image, Z is supposed to be equal to three.

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 to each other, hence they amount to a few units. In other words, cooperation occurs only among nearby nodes. The simulator that we wrote for performing these tests has been published as open source[2], and it relies on discrete event simulator library "Simpy".

In the following subsections, we evaluate the proposed algorithm in the scope of two settings.In the first setting (Section V-A), we set different traffic loads λ to the nodes, but we make them fixed over time. Here we show how the RL approach can outperform the classic power-of-two random choices strategy finding a solution that maximizes ι in every node when using the set of actions $\mathcal{A}'$. In the second setting (Section V-B), we apply the traffic study results (Section III-E), and we make nodes to follow the load traces in Figure 2. In this setting, we again show how the RL approach can outperform the standard one when the traffic is variable over time.

In all of these experiments, the proposed RL algorithm is labeled as "Sarsa" while the power-of-two random choices one "Pwr2", that specifically refers to the *two* random 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.

A. Heterogeneous Loads

In this experiment, we used two kinds of tasks, one (type 0) that is expected to run at 60fps and therefore, we supposed that it has a deadline of 16ms and mean duration of 8ms ($\sigma = 0.4$ms) and one (type 1) that is expected to run at 30fps and therefore it has a deadline of 40ms and a mean duration of 20ms ($\sigma = 0.4$ms).

Figure 4 shows the behavior 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 λ_i: 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-two random choices with the threshold set to 2 (i.e., policy 00111). What emerges again is that the learning approach exactly mimics the power-of-two random choices from 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 as the power-of-two random choices with the threshold set to 2.

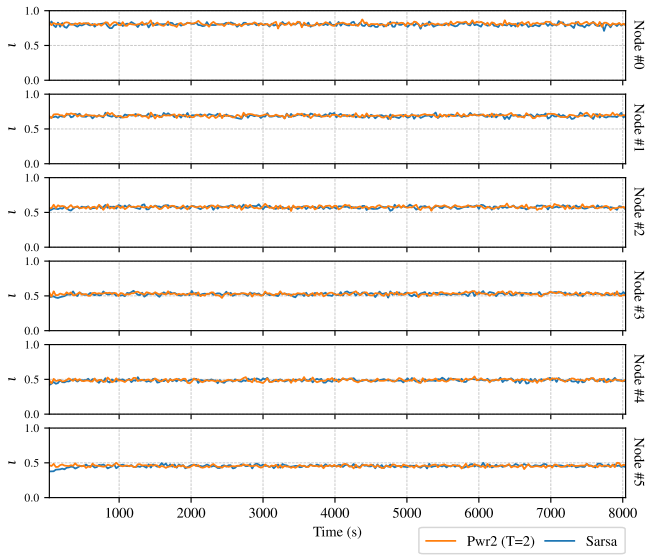


Fig. 4. Comparison between Sarsa and Pwr2: behavior of the in-deadline rate ι for every node when the load is fixed and the same to every node

At this point, we wonder how we can increase the performance by using the same reduced set of policies. One of the ways to do is to increase the action space of the agent. For this reason, we introduced the set of actions $\mathcal{A}'_i$. Figure 5 shows the behavior 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 forward tasks directly 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 is equal to every node. This means that even the nodes that, with the policy 001111 could have a better reward but they are not selfish and voluntarily decrease their reward for making the others achieve the best reward. We do believe that this behavior 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 by 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.

B. Geographic Scenario

As described in Section III the open data for New York City has been used for generating the traffic to six fog nodes. 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 2). In Figure 6, we can observe how the behavior that

[2]https://gitlab.com/gabrielepmattia/simulator-2023-tccn

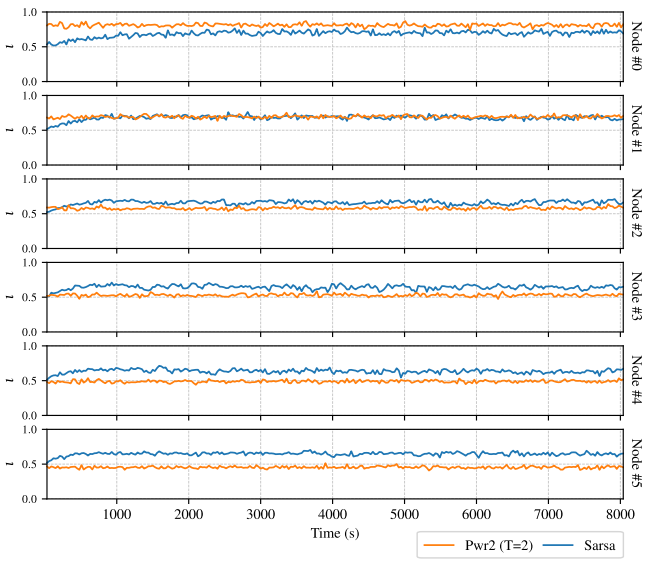


Fig. 5. Comparison between Sarsa and Pwr2: behavior of the in-deadline rate ι for every node when load is fixed but heterogeneous

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 following the policy 001111 (power-of-two random choices with threshold 2) could make a node able to reach a better reward, and this behavior is invariant with respect to the traffic variability.

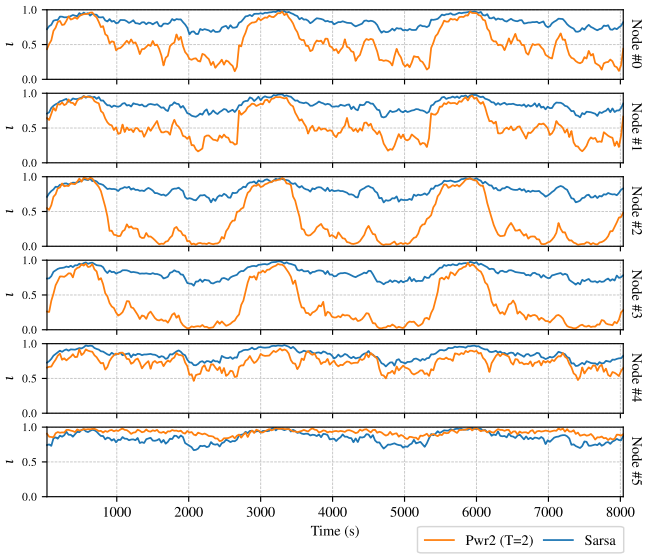


Fig. 6. Comparison between Sarsa and Pwr2: behavior of the in-deadline rate ι for every node when the load is variable according to the geographic scenario, Figure 2

As a final note, we remark that the policy that is chosen by this Reinforcement Learning approach, both in the fixed and in the geographic scenario, is not trivial and easy to be manually determined but it dynamically changes over time, and unfortunately, it is not feasible to be represented in a figure, for this reason, its chart has been skipped.

VI. Experimental Setting

In this section, we introduce and describe the implementation of the proposed RL scheduling policy in a pseudo-real setting by using our open-source framework "P2PFaaS" [43] which has been installed on 12 Raspberry Pi 4 (six of which with 4GB of RAM and size with 8GB of RAM) by using the OpenBalena[3] framework, a partial open-source framework which allows deploying sets of Docker containers on a fleet of SBCs (Single-Board Computers). Indeed, the framework is composed of a set of Docker containers that implement scheduling, discovery, and scheduling policy learning capabilities. We then used a Dell XPS 8900 desktop PC with Fedora to generate the traffic toward all the nodes. The PC has an Intel i7-6700 processor and 16GB of RAM. All the entities have been attached to a private subnetwork by using a 1Gbps Ethernet wired connection.

This environment is called "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 does not have a great impact on the trustworthiness of the final results achieved.

A. 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[4] IaaS framework which allows the simultaneous deployment and management of Docker containers in clusters of devices of the same type. In the final deployment, the Raspberry Pi nodes are attached to a 1 Gigabit network switch, then there is a router and a server which has two purposes: the first is to host the OpenBalena framework, and the second is to generate the traffic to the nodes and collect the data. The traffic generator script 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).

B. Single Node behavior

The FaaS function used in all the following tests is a face detection function[5] 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 III, we used two different images. 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

[3]https://www.balena.io/open
[4]https://github.com/balena-io/open-balena
[5]https://github.com/esimov/pigo-openfaas

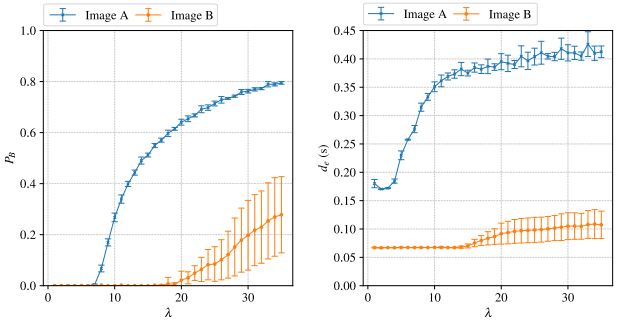
when receiving the output) and the TCP connection time that is irreducible unless we do not relay anymore on HTTP calls for triggering the function, but we use raw sockets. Again, the keep-alive feature cannot be used for the same aforementioned reasons.

	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 III
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)

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 that the node is able to execute four tasks at a time with a single core that 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 still expect to have 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* queue model, which is normally used ([6], [5]) for modeling a fog node. In real world, due to the underlying operating system (in our case balenaOS and in the end, the Linux Kernel) that continuously balances the load among the different cores and therefore it is quite difficult to have a task that strictly holds a core for its entire processing time, moreover, there is an underlying set of programs 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 the desired load level 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 that triggered the execution of a FaaS at gradually increasing (λ) rates, the node is configured without a queue, with three executable tasks in parallel[6], and the arrival distribution is deterministic (i.e. inter-arrival time is fixed). Figure 7 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 Image B $\lambda \approx 18$ reqs/s. This result could not be derived in theory, since logically, suppose 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 three cores this should be multiplied by three 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 7b 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 to the duration when $\lambda = 1$ req/s.

[6]In this and in the following experiments we leave one core free to execute the OS and the background services.



(a) Blocking probability (P_B) (b) Average task execution time (d_e)

Fig. 7. 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.

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 paper 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.

C. Performance metric

Let us introduce performance metrics that will allow us to compare the different scheduling policies. In our experiments, our purpose is obviously to maximize the reward, but an approach that also distributes it with respect to one that makes every node behave selfishly is 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 III-C, it is equal to the percentage of tasks that are completed within the deadline, therefore, the higher the value the higher is the goodness of the scheduling policy since it is able to execute more tasks 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, therefore, the higher the cooperation, the higher

the amount of data that is exchanged and this can have an impact on the transmission time of the tasks.

Similarly to the metrics of the simulations, we represent their behavior 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 best in the simulations, namely $\mathcal{A}'$ (Section III-B2) 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 random choices where nodes are assumed to be M/M/k/k queues (the most similar to a real environment), or even better, a model that mimics real nodes has not been studied yet and it is left as future work, therefore the $T = 1$ decision is not fully justified, however, given the previous studies [6] we expect to be the best threshold to use for comparison.

D. Results

In this section, we will present the test results of the proposed policy called, as in the simulations, "Sarsa", which run in the presented pseudo-real environment. Table IV 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 maximum length of the execution queue K_e 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 reward. We 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, only uses one deadline but the traffic, differently from the other nodes, is geographic and derived from the study in Section V-B. 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 #0 in the study has been assigned to the Raspberry Pi node #0 and #6, the curve of node #1 to nodes #1 and #7 and so on.

In every experiment and for every node, the traffic is distributed according to a Poisson distribution (except for the geographic traffic in which the inter-arrival time is deterministic but changes according to the load curve), moreover, the ϵ-greedy policy for action selection starts with $\epsilon = 0.9$ and has a decay of 0.9995 that is applied whenever a new task arrives.

				Deadlines (ms)				
#	Traffic	Payload	γ	T_A	T_B	K	K_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 IV
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 MAXIMUM LENGTH OF THE EXECUTION QUEUE K_e AND THE DURATION.

1) Experiment 1 - No deadline: In this experiment, we used fixed and heterogeneous traffic, in particular, the traffic λ_i from $i = 0$ to $i = 11$ is 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 11 and 12. The image payload used is only Image A, and the learning parameters are $\alpha = \beta = 0.01$ with a window size $Z = 20$. In Figure 8 we can observe the behavior 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 do not perform any comparison, we only evaluate the behavior 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 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 stabilized situation. Moreover, we can notice how the P_f is not depending on the load, but we remind that in this case, deadlines are infinite, and therefore, the learner only tries to execute more tasks as possible without rejecting them.

2) Experiment 2 - Fixed load: In this experiment, we introduce, as in the simulations, the concept of deadlines. We use the two presented payloads (Table III) Image A and Image B, assigning two different deadlines. We calculate the deadline on the average d_t as shown in Figure III. The deadline T_X where X is the (Image) A or B is given by equation

$$T_X = d_{t_X} + \gamma d_{t_X} \tag{10}$$

For example, for Image A and choosing $\gamma = 0.05$, $T_A = 188.25 + 0.05 \cdot 188.25 = 197.66$ms. We suppose that the task arrives in percentage 50/50, and the fixed traffic λ_i from $i = 0$ to $i = 11$ 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 VI-B, 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 $\gamma = 0$, which maps to a deadline equal to the average duration of a task, to $\gamma =$

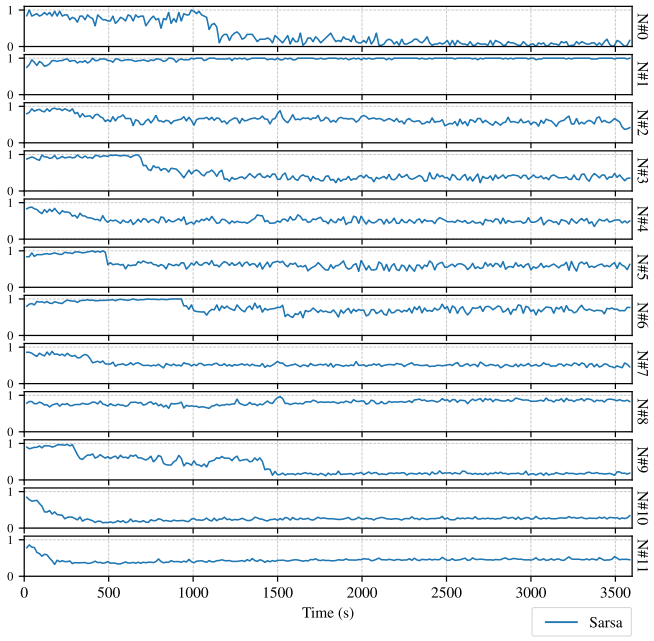


Fig. 8. Experiment 1: P_f, behavior of the percentage of forwarded tasks during the experiment, the average is performed every 15 seconds. In this experiment, deadlines are not considered and only a single payload is used, Image A.

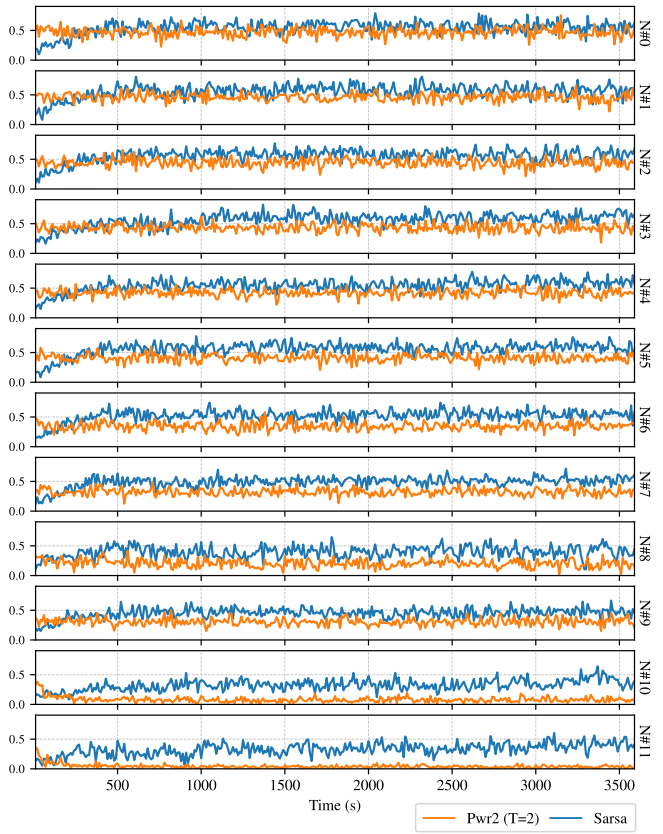


Fig. 9. Experiment 2.2: behavior of ι during the entire test duration with deadline set at 5% of the total task duration d_t. Values are averaged every 15 seconds.

0.3. Our "Sarsa" approach is compared with the Power-of-two random choices with $T = 1$, as widely presented in Section V.

In every experiment performed, our Sarsa approach performs better of power-of-two random choices "Pwr2", but for space issues, we only report the reward (ι) behavior of experiment 2.5 (Figure 9), in which the effect is more evident. What we can observe is a very similar behavior to the simulations, in particular to the Figure 5, 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.

To have a clearer picture of the results of the proposed algorithm across all the nodes, we analyzed the mean and the variance of the reward (ι) and of the forwarding probability P_f when the deadlines vary. Figure 10 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, our 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 $\gamma = 0.3$.

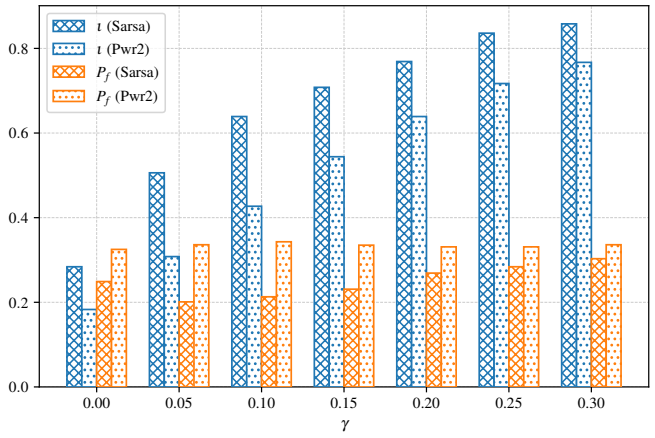


Fig. 10. Experiments 2.x: Comparison of average reward ι and forwarding probability P_f across all the 12 Raspberry Pi nodes in the last 200s of a 3600s test.

Figure 11 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 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.

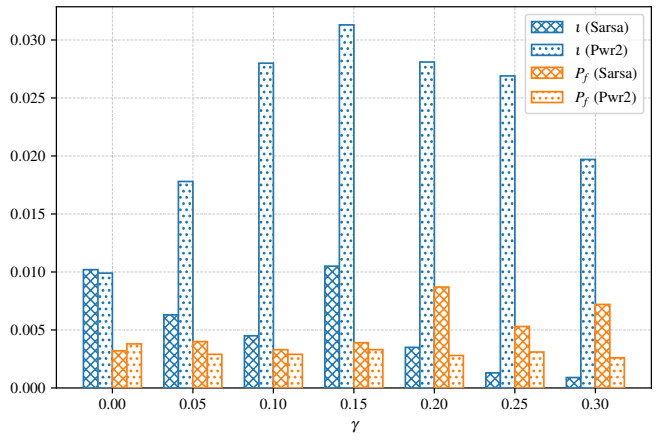


Fig. 11. Experiments 2.x: Comparison of variance of average reward ι and forwarding probability P_f across all the 12 Raspberry Pi nodes in the last 200s of a 3600s test.

3) Experiment 3 - Geographical: The final experiment uses a traffic distribution that is derived from the study in Section V-B. 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 map to nodes 0-5 of Figure 2. 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 VI-D2) in which the maximum load was set to 14, 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 the entire duration of the test, except for the initial training phase of the algorithm. However, the most evident improvement can be recognized 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 behavior is determined by the hardware differences between the nodes. In the real world is quite impossible to have two nodes that have exactly the same performance characteristics, in our 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. This final result inadvertently shows that our learning approach is able to derive the best policy even if the nodes are heterogeneous, which is an unavoidable characteristic of real devices.

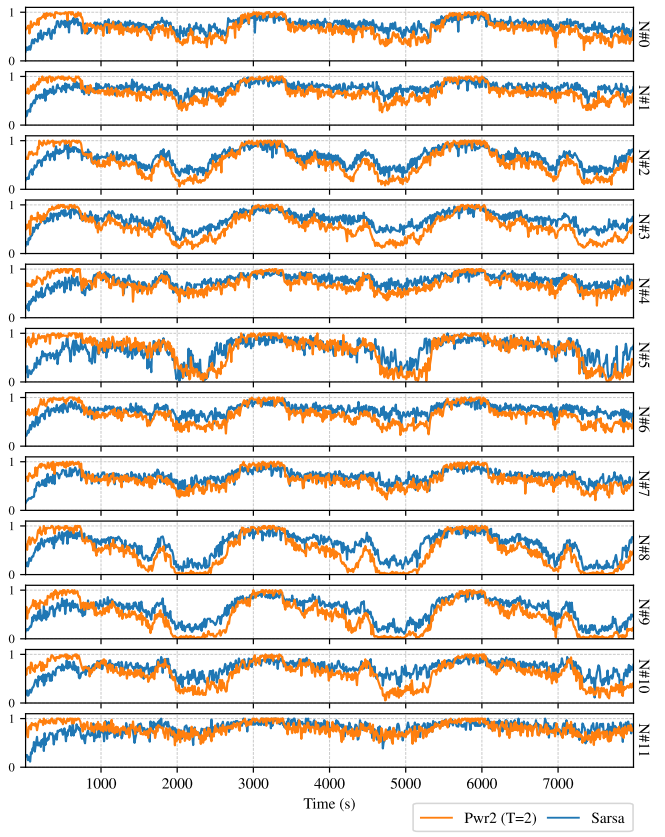


Fig. 12. Experiment 3: behavior of ι during the entire test duration with deadline set at 5% of the total task duration d_t. Traffic to every node is the one described in 2. Values are averaged every 10 seconds.

VII. Conclusions and Future Work

In this paper, we addressed the problem of extending the power-of-two random choices distributed scheduling scheme with Reinforcement Learning in order to be able to efficiently schedule real-time and deadline-constrained tasks. Starting with simulation, we demonstrated that our approach is able to outperform the power-of-two random choices when it is made able to directly forward tasks to specific nodes, and this holds even if the traffic traces are extracted from a real-world dataset. Moreover, the approach that makes each node act as an independent learner and does not have any kind of load information about the others is able to maximize the overall performance of the system by making each node not

behave in a selfish manner. Then, in the second part of the work, we introduced an improved version of the P2PFaaS framework, which implements the proposed RL algorithm, and we showed that the same results characteristics obtained in the simulations are maintained in a (pseudo) real deployment with 12 Raspberry Pis. The real implementation of the approach made us realize which are the critical points that must be addressed in order to make the algorithm work even in a real deployment. However, further improvements to the approach can be studied by, for example, taking into account the energy aspect of the nodes, the complexity of the state can be increased by introducing more state parameters like the CPU time, which has a great impact on the duration of the tasks, and finally, what could be further investigated is a second learning process regarding the tasks that are forwarded. Indeed, these tasks are, in this work, always accepted by the remote nodes, but these nodes may want to reject them to improve their reward.

Gabriele Proietti Mattia is currently a PostDoc researcher (RTD-A) at the Department of Computer, Control and Management Engineering "Antonio Ruberti" of the Sapienza University of Rome where he received the BSc, MSc and PhD degrees in Engineering in Computer Science in 2017, 2019 and 2023, respectively. His research interests include Fog and Edge computing, distributed scheduling, load balancing, and Reinforcement Learning.

Roberto Beraldi received the laurea Degree from the "University of Calabria" in 1991, a master degree from CEFRIEL (Politecnico di Milano) in 1992 and a PhD in computer science in 1996. From 1996 he worked at Italian's National Institute of Statistics (ISTAT), and since 2002, works at the Department of Computer, Control and Management Engineering "Antonio Ruberti" of Sapienza University of Rome, Italy, where he is currently an Associate Professor. His research interests include mobile networking, Fog/Edge computing, and distributed systems. He regularly serves as TPC member of international conferences and journals in these fields.

REFERENCES

[1] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi, "fog computing conceptual model," NIST, Tech. Rep., 2018.

[2] 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.

[3] J. Barthélemy, N. Verstaevel, H. I. Forehead, and P. Perez, "Edge-computing video analytics for real-time traffic monitoring in a smart city," *Sensors (Basel, Switzerland)*, vol. 19, 2019.

[4] S. V. Broucke and N. Deligiannis, "Visualization of real-time heterogeneous smart city data using virtual reality," *2019 IEEE International Smart Cities Conference (ISC2)*, pp. 685–690, 2019.

[5] 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.

[6] R. Beraldi and G. Proietti Mattia, "Power of random choices made efficient for fog computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.

[7] 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.

[8] B. Jamil, H. Ijaz, M. Shojafar, K. Munir, and R. Buyya, "Resource allocation and task scheduling in fog computing and internet of everything environments: A taxonomy, review, and future directions," *ACM Comput. Surv.*, vol. 54, no. 11s, sep 2022. [Online]. Available: https://doi.org/10.1145/3513002

[9] S. Azizi, M. Shojafar, J. Abawajy, and R. Buyya, "Deadline-aware and energy-efficient iot task scheduling in fog computing systems: A semi-greedy approach," *Journal of Network and Computer Applications*, vol. 201, p. 103333, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804522000029

[10] H. O. Hassan, S. Azizi, and M. Shojafar, "Priority, network and energy-aware placement of iot-based application services in fog-cloud environments," *IET Communications*, vol. 14, no. 13, pp. 2117–2129, 2020. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-com.2020.0007

[11] M.-T. Zhou, T.-F. Ren, Z.-M. Dai, and X.-Y. Feng, "Task scheduling and resource balancing of fog computing in smart factory," *Mobile Networks and Applications*, Jun 2022. [Online]. Available: https://doi.org/10.1007/s11036-022-01992-w

[12] M. Abdel-Basset, D. El-Shahat, M. Elhoseny, and H. H. Song, "Energy-aware metaheuristic algorithm for industrial-internet-of-things task scheduling problems in fog computing applications," *IEEE Internet of Things Journal*, vol. 8, pp. 12 638–12 649, 2021.

[13] G. Li, J. Wu, J. Li, K. Wang, and T. Ye, "Service popularity-based smart resources partitioning for fog computing-enabled industrial internet of things," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4702–4711, 2018.

[14] 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.

[15] 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.

[16] 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.

[17] 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 computing," *Sensors*, vol. 18, no. 9, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/9/2830

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301521

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] A. AlOrbani and M. Bauer, "Load balancing and resource allocation in smart cities using reinforcement learning," in *2021 IEEE International Smart Cities Conference (ISC2)*, 2021, pp. 1–7.

[36] O. Houidi, D. Zeghlache, V. Perrier, P. T. Anh Quang, N. Huin, J. Leguay, and P. Medagliani, "Constrained deep reinforcement learning for smart load balancing," in *2022 IEEE 19th Annual Consumer Communications Networking Conference (CCNC)*, 2022, pp. 207–215.

[37] 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. 4951–4966, Nov 2020. [Online]. Available: https://doi.org/10.1007/s12652-020-01768-8

[38] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, pp. 939–951, 2021.

[39] 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.

[40] 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.

[41] 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.

[42] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[43] 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. 101290, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711022002084