Leveraging Reinforcement Learning for online scheduling of real-time tasks in the Edge/Fog-to-Cloud computing continuum

Gabriele Proietti Mattia, Roberto Beraldi
Department of Computer, Control and Management Engineering "Antonio Ruberti",
Sapienza University of Rome,
Email: proiettimattia@diag.uniroma1.it, beraldi@diag.uniroma1.it

***Abstract*—The computing continuum model is a widely accepted and used approach that make possible the existence of applications that are very demanding in terms of low latency and high computing power. In this three-layered model, the Fog or Edge layer can be considered as the weak link in the chain, indeed the computing nodes whose compose it are generally heterogeneous and their uptime cannot be compared with the one offered by the Cloud. Taking into account these inexorable characteristics of the continuum, in this paper, we propose a Reinforcement Learning based scheduling algorithm that makes per-job request decisions (online scheduling) and that is able to maintain an acceptable performance specifically targeting real-time applications. Through a series of simulations and comparisons with other fixed scheduling strategies, we demonstrate how the algorithm is capable of deriving the best possible scheduling policy when Fog or Edge nodes have different speeds and can unpredictably fail.**

I. Introduction

Delay-sensitive applications are characterised by tasks that must be completed in a given deadline and, as such, they cannot be processed in the Cloud. The network latency that is experienced for reaching the cloud server is inadmissible for these scenarios, let us think about real-time applications like Augmented Reality (AR) or Virtual Reality (VR). As seen in the last decade, the only way for coping this physical limitation is to add another layer in between the users and the cloud. This layer that is called Fog or Edge computing layer [1], according to the nuances of where the computation is located, has the principal characteristic that is placed near to the entities which require the service, this allows for drastically reducing the network latency. It is obvious that big data centres cannot be placed everywhere, as a consequence, the main distinct feature of this intermediate layer is that it can be composed by nodes that are smaller, have less computational power, and mostly they are heterogeneous [2]. This means that the same task of an application can have very different execution times when it is executed in a node or in another one and most of the time it is very challenging to know in advance how much time the execution will last, even if we know which are the technical specifications of the devices, like the CPU cores, the CPU clock rate, its architecture, the quantitative of RAM or we are perfectly aware of the code and of the libraries that the task may use. Moreover, these nodes have an underlying operating system and background software that can momentarily and unpredictably reduce the CPU time available for the tasks, protracting their execution time.

In this work 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 paper addresses the scheduling problem of the tasks, that is where to schedule them, by using an online Reinforcement Learning approach.

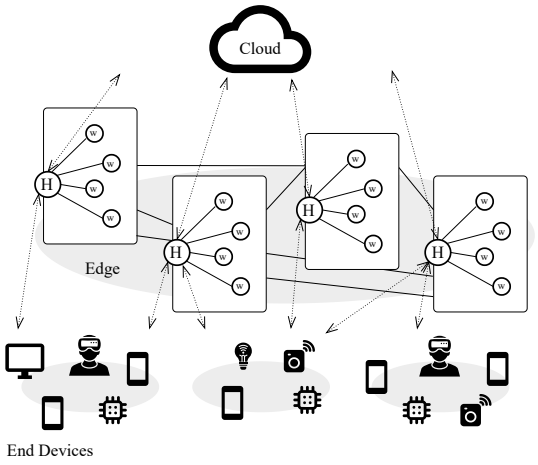

Fig. 1. The Edge-to-Cloud continuum environment considered as computing scenario.

Figure 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 work, 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 work 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 work 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;

The rest of this paper is organised as follows. In Section II we present some related works, in Section III we define the system model, instead in Section IV we describe the reinforcement learning approach the we propose. In Section V results of the simulation of the proposed protocol are illustrated and, finally, we will draw conclusions in Section VI.

II. Related Work

The Reinforcement Learning algorithm that we use in this work 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 [3]. However, there are many works in literature that rely on Deep Q Learning for solving the task scheduling problem in the edge or fog computing, especially because of the high dimensionality of the state space. For example, in [4], the authors propose a deep reinforcement approach for resource allocation in a MEC system, differently from our work, the allocation scheme is based on time slices and the objective is to minimise the execution time. In [5], the focus is instead on base stations that must be selected by the client in a ultra high-dense network, the authors show through numerical experiments the beneficial effect of their solution. The authors of [6] specifically studies the task scheduling in the edge computing and use the reinforcement learning for deciding the order of the execution of the tasks and in which machine they have to be executed, the approach uses the deep reinforcement learning but the scheduling is not done online. In [7] a solution for caching at the edge is proposed, the authors use reinforcement learning for finding an optimal stochastic allocation policy, the approach is tested with simulations. In [8], 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 our work is [9] which focuses on online scheduling and using time differential learning, but the tasks do not have to meet a deadline. Then [10] introduces a specific study on the task placement in the edge-to-cloud computing continuum.

Other works are still focused on scheduling but targeting the energy consumption [11], [12], vehicular networks [13], [14], network resources allocation [15] or security [16].

	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 I
List of symbols used

III. System Model and Problem Definition

The online scheduling problem that is configured in this work 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[1] 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.

[1]https://pypi.org/project/simpy/

A. Environment

As depicted in Figure 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 work we consider it as fixed and the dynamic case is left as future work.

B. Task and Delay models

In our 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 IV 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 workers, each transmission is elaborated one at a time but the queue has no fixed size. Figure 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.

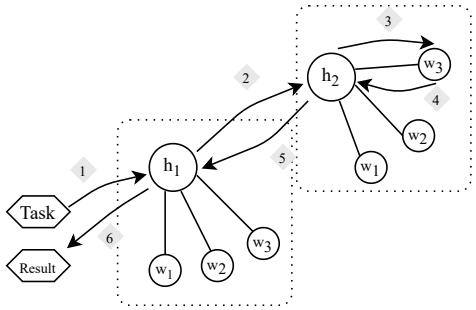

Fig. 2. The task request path from the client to the final worker node when s_1 decides to forward the task to another cluster. The transmission latency is simulated with a transmission queue per scheduler node.

C. The Agent

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

$$\pi : \mathcal{S} \rightarrow \mathcal{A} \tag{1}$$

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 V-A), the task can be assigned only to the worker nodes. The set of actions for the node i can be described as:

$$\mathcal{A}_i = \{\text{reject}, \text{cloud}\} \ \cup \ \mathcal{W}_i \tag{2}$$

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

$$\mathcal{A'}_i = \{\text{reject}, \text{cloud}\} \ \cup \ \mathcal{W}_i \ \cup \ \mathcal{C}_i \tag{3}$$

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

D. 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 our 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 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.

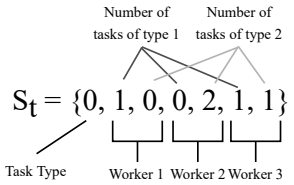

Fig. 3. The state representation of a scheduler node at time t with three worker nodes.

However, the state is not used as is for the learning process, indeed, the tiling [3] technique is used for mapping the vector to another vector but in a 24-dimensional vector space.

As presented in section V, 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.

E. Reward

The definition of the reward is crucial for obtaining the desired results of meeting the user QoS constraints. In our case, we focused the attention 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 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 work, 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 work 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 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:

$$R(s,a) = \begin{cases} 2 & \text{if } d_t \leq 1/\omega_n \\ 1 & \text{if } 1/\omega_n < d_t \leq 1/\omega_m \\ -1 & \text{if } d_t > 1/\omega_m \end{cases} \tag{4}$$

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.

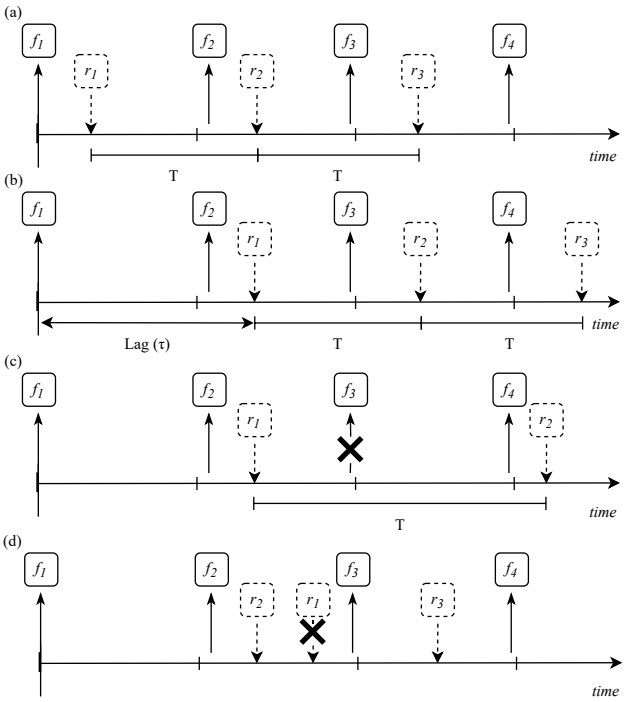

Fig. 4. The frame generation and process in real-time applications, four cases of identifying performances. In case (a) rate of frame generation is the same of the frame result receiving and there is no lag, in case (b) a lag is introduced, in case (c) the rate of the received frame is different by the one of the frame sent, finally in case (d) the order of received frames is altered.

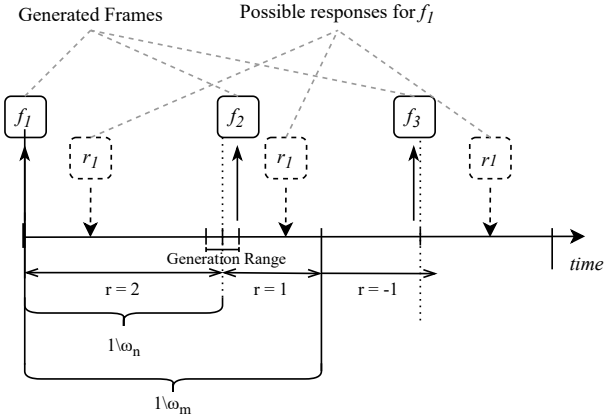

Fig. 5. Diagram that illustrates how reward is assigned when the task is executed and the frame f_1 returns to client after being processed (r_1).

F. Performance Parameters

For understanding the performance of our 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;
- 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.b):

$$\tau = d_t - 1/\omega_n \tag{5}$$

IV. 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 [3]:

$$q_*(s,a) = \sum_{r,s'} p(s', r|s, a) \left[r - \max_{\pi} r(\pi) + \max_{a'} q_*(s', a') \right] \tag{6}$$

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 [3]:

$$\delta_t = R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \vec{w}_t) - \hat{q}(S_t, A_t, \vec{w}_t) \tag{7}$$

This form can be applied to any function approximation algorithm for estimating the q_*, in our case we used the tiling technique [3]. As already introduced, in our 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 1 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 III) 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 rejected. We 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 1 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 completed its execution (which means that 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") 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.

V. Results

In this section, we present the results of the simulation in the described environment of our proposed RL-based scheduling algorithm both in a single cluster (Section V-A) and in a multi-cluster setting (Section V-B). 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. Our approach not only make possible to reach the desired frame rate to each traffic flow, but also minimizes 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 II. Specifically, the service rate is normalized with respect to the highest clock speed in the group, e.g., the service rate of the Asus Tinker (1.8 GHz)

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

is $\S = \frac{1.8}{2.0} = 0.9$. The cloud instead runs always with speed equal to 1.0.

The traffic flows arriving to each cluster is defined in Table II. Table III, 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 II
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 III
THE SPECIFICATION OF THE NETWORK PARAMETERS IN THE SIMULATION.

A. Single Cluster

The setting of this series of experiments if depicted in Figure 6. We have one single cluster which receives exactly four flows of traffic, described in Table IV. 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.

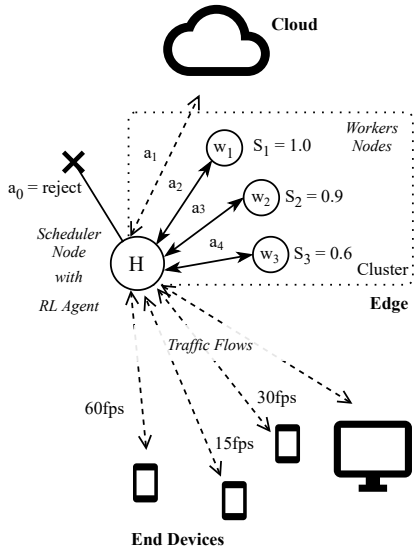

Fig. 6. The setting of the of the experiments on a single cluster with three workers nodes and four traffic flows (Section V-A).

	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 IV
THE LIST OF TRAFFIC FLOWS USED FOR THE SIMULATION, EACH TRAFFIC FLOWS TF_i GENERATES TASKS OF TYPE i.

1) Normal Operation: The Figure 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 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 (Section III-E) allows, at the same time, to reach the desired ω_n and the d_t and to reduce the lag time τ. We 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.

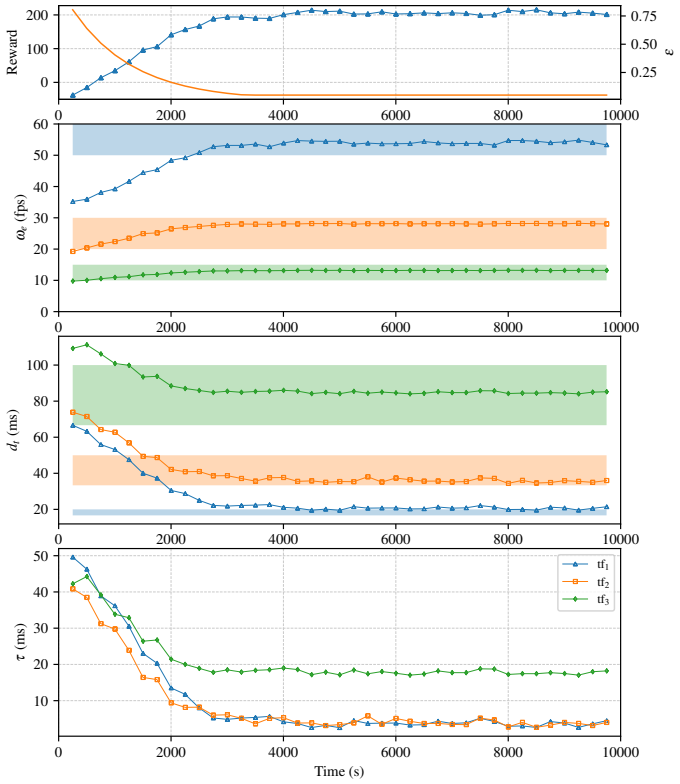

Fig. 7. Results of the simulation of a single cluster and three worker nodes (Section V-A), regarding, from top to bottom, the reward, the effective frame rate ω_e, the total response time d_t and the lag time τ.

The Figure 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 V shows the comparison of our 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, our approach allows to meet the user requirements and minimize the lag time τ, this because our objective is to

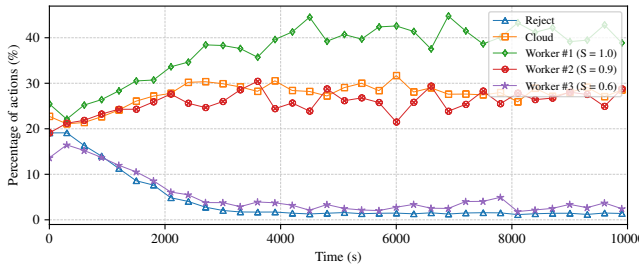

Fig. 8. The distribution of the actions made by the agent on the scheduler node in the experiments with a single cluster and three worker nodes (Section V-A).

maximize a reward based on the total response time of the tasks.

2) 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 10 shows the results of the simulation with the same structure of Figure 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.

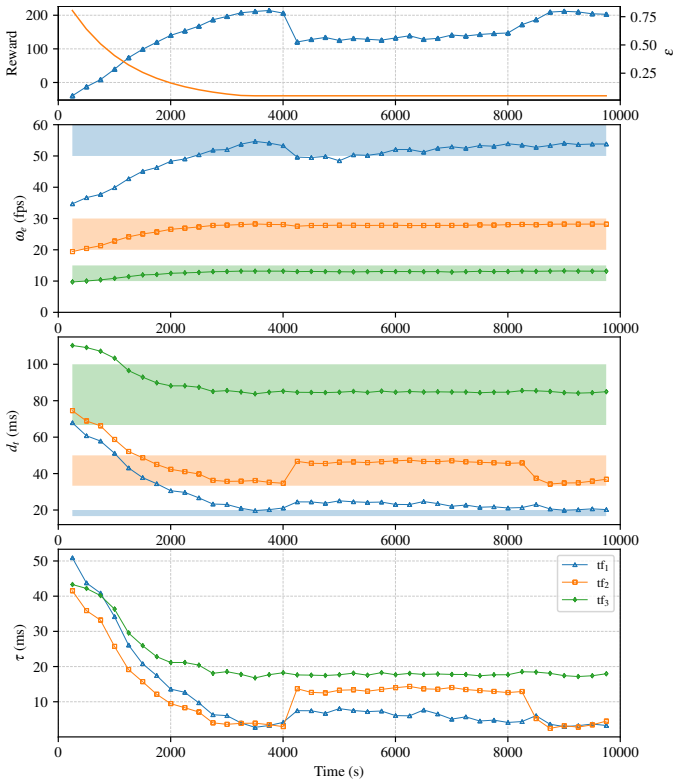

Fig. 9. Results of the simulation of a single cluster and three worker nodes (Section V-A), regarding, from top to bottom, the reward, the effective frame rate ω_e, the total response time d_t and the lag time τ. We assume that node #1 fails at time 4000.

In the Figure 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).

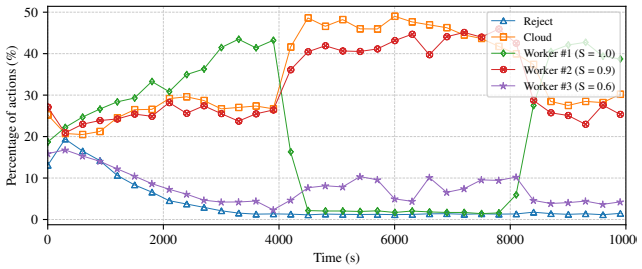

Fig. 10. The distribution of the actions made by the agent on the scheduler node in the experiments with a single cluster and three worker nodes (Section V-A). We assume that node #1 fails at time 4000.

B. 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 scheduler. We suppose that each cluster receives the flows described in the Table IV and we use our 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 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 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 expected. We 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 Section III.

VI. Conclusions and Future Work

In this paper, we presented an approach for solving the online task scheduling in the edge or fog to cloud continuum computing model by using the Reinforcement Learning. This

	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 V

COMPARISON REGARDING THE EFFECTIVE FRAME RATE (ω_e), THE LAG TIME (τ) AND THE TOTAL RESPONSE TIME d_t BETWEEN OUR ALGORITHM "SARSA TRAINED" AND OTHER TWO APPROACHES: SCHEDULING TO THE LEAST LOADED NODE AND RANDOM SCHEDULING.

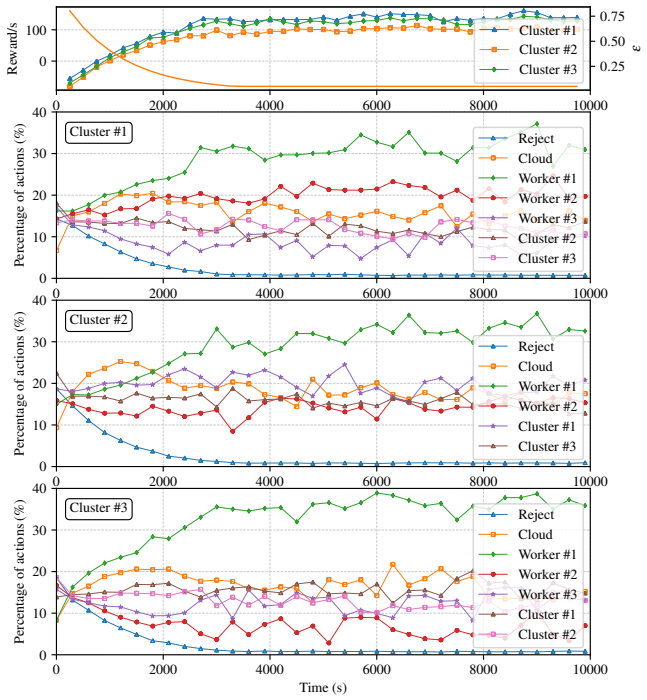

Fig. 11. Results of the simulation in a three clusters setting, regarding, from top to bottom, the reward per second, the percentage of the chosen action over time for the three clusters, in order.

approach is perfectly suiting the problems that regards this dynamic context, as for example, heterogeneity of the nodes, difficulties on 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). We showed the results of our approach both in a single cluster and in a multi-cluster environment, illustrating that in any of these case, given an 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 neighbor clusters.

As anticipated in the paper, unfortunately some points have been left open and will be further investigated, for example, in the experiments we hypothesized 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.

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] D. Kimovski, R. Mathá, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan, "Cloud, fog, or edge: Where to compute?" *IEEE Internet Computing*, vol. 25, no. 4, pp. 30–36, 2021.

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

[4] 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.

[5] 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.

[6] 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. [Online]. Available: https://www.mdpi.com/1424-8220/21/5/1666

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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. 5532410, May 2021. [Online]. Available: https://doi.org/10.1155/2021/5532410