IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 16, 2020, accepted October 19, 2020, date of publication October 23, 2020, date of current version November 3, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3033421

QoS-Aware Workload Distribution in Hierarchical Edge Clouds: A Reinforcement Learning Approach

CHUNGLAE CHO, (Member, IEEE), SEUNGJAE SHIN, (Member, IEEE), HONGSEOK JEON, AND SEUNGHYUN YOON
Telecommunications and Media Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon 34129, South Korea

Corresponding author: Seungjae Shin (sjshin0505@etri.re.kr)

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2017-0-00045, Hyper-connected Intelligent Infrastructure Technology Development).

ABSTRACT Recently, edge computing is getting attention as a new computing paradigm that is expected to achieve short-delay and high-throughput task offloading for large scale Internet-of-Things (IoT) applications. In edge computing, workload distribution is one of the most critical issues that largely influences the delay and throughput performance of edge clouds, especially in distributed Function-as-a-Service (FaaS) over networked edge nodes. In this paper, we propose the Resource Allocation Control Engine with Reinforcement learning (RACER), which provides an efficient workload distribution strategy to reduce the task response slowdown with *per-task* response time Quality-of-Service (QoS). First, we present a novel problem formulation with the per-task QoS constraint derived from the well-known *token bucket mechanism*. Second, we employ a problem relaxation to reduce the overall computation complexity by compromising just a bit of optimality. Lastly, we take the deep *reinforcement learning* approach as an alternative solution to the workload distribution problem to cope with the uncertainty and dynamicity of underlying environments. Evaluation results show that RACER achieves a significant improvement in terms of per-task QoS violation ratio, average slowdown, and control efficiency, compared to AREA, a state-of-the-art workload distribution method.

INDEX TERMS Deep reinforcement learning, edge computing, resource allocation, workload distribution.

I. INTRODUCTION

Thanks to the recent academic and industrial innovations of communications, networking, Internet-of-Things (IoT), and cloud computing, we are seeing that the era of edge computing is coming [1]–[3]. The main idea of the edge computing is to deploy edge cloud servers close to users nearby base stations (BS) or gateways for short-distance access, which is likely to result in a low delay and high throughput with energy efficiency. Therefore, edge computing is recognized as a promising paradigm that can provide users computation and data resources with an ultra-low delay response time. It is expected to enable various computation-intensive and delay-critical services such as virtual reality (VR), augmented reality (AR), real-time online IoT, and ultra-high-definition (UHD) video streaming. Leading cloud vendors are attracting users to use their edge cloud solutions (e.g., AWS IoT Greengrass [4], Azure IoT Edge [5]) with smart speakers (e.g., Echo [6], Google Home [7]) and light-weight cloud servers (e.g., Snowball [8], Azure Data Box Edge [9]).

In addition to the above progress, the edge computing is also expected to accommodate *Function-as-a-Service* (FaaS) such as AWS Lambda [10], Azure Functions [11], and Open-Whisk [12]. FaaS is an emerging paradigm that enables serverless computing to provide a fast, flexible, and convenient way to build customized information technology (IT) applications. This is done by assembling fine-grained function blocks over cloud platform without worrying about low-level development, resource provisioning, and management & operation (MANO). The cloud platform with FaaS itself transparently does those laborious jobs.

In order to make the FaaS come true in edge computing environments, we should consider resource allocation and scheduling issues. Due to the fine-grained nature of the FaaS,

The associate editor coordinating the review of this manuscript and approving it for publication was Chuan Heng Foh.

the size of the resource allocation and scheduling problems mainly increases. Moreover, since edge cloud servers are likely to have limited resources compared to conventional data centers, we should consider the *workload distribution* functionality: workloads are distributed among edge and remote computing nodes based on their execution characteristics such as Quality-of-Service (QoS), cost, and other constraints. For example, the tasks of a *delay-sensitive* or *mission-critical* application should be executed at the edge node. However, it is less critical whether to execute the tasks of a *delay-tolerant* application at the edge node or not. As another example, it may be desirable to execute a task near the node where its I/O data are located, no matter whether the node is edge or remote.

There have been many studies that provide delay, energy, or cost-efficient workload distribution approaches for mobile edge clouds. Most of the studies model the workload distribution issues as optimization problems and relax the problems to more tractable ones to exploit widely used problem-solving techniques across convex optimization, graph theory, greedy, and heuristic methods. Although those works have achieved meaningful advances, the workload distribution issues still encounter considerable challenges as follows:

- Most of the existing works do not provide a high enough level of QoS-awareness, especially in terms of the task response time. In addition, they consider a limited level of QoS awareness rather than per-task response time QoS [13]–[30]. They either rely on the assumption about a specific task arrival process and service time distribution [19], [24].
- The workload distribution problem introduces a significant decision complexity. In workload allocation, we typically need to do the job for each pair of user equipment (UE) and task type. Therefore, the complexity is $O(U \cdot K)$, where U is the number of UEs and K is the number of task types, which must be tremendous as U and K increase. We need some approaches to reduce such complexity, especially for a large U.
- It is tough to precisely model the underlying edge cloud system due to the uncertainty of details. The workload arrival patterns are expected to become more dynamic and unpredictable when a large number of IoT applications are adopted. Such uncertainty and dynamicity make the workload distribution problem more difficult. However, the existing works still model the underlying system in a somewhat unsophisticated manner and assume the workload arrival pattern as a predictable stochastic process, which may result in a degraded performance under the real environment.

In this paper, we propose the Resource Allocation Control Engine with Reinforcement learning (RACER), a novel workload distribution approach to handle the above three challenges. Our contributions are as follows:

- First, to address the per-task response time QoS issue, we revise the conventional problem with the average task response time constraint to a novel problem with the per-task QoS constraint. We derive the constraint from the well-known *token bucket mechanism* [31], which provides a guaranteed upper bound on the task response time (Section IV-A).
- Second, we significantly reduce the problem complexity by removing the task distribution vector from the decision variables of the original workload distribution problem. Instead, the task distribution vector is determined implicitly by the token bucket mechanism over the hierarchical edge cloud architecture. Even though this reduction sacrifices the optimality of the solution, we show that we can still achieve a desirable suboptimal performance with some assumptions (Section IV-B).
- Third, by taking the deep *reinforcement learning* (RL) approach as an alternative solver of the workload distribution problem, RACER has an improved capability against the uncertainty and dynamicity of underlying environments. By repeating the *observe-action-reward* process in the distributed edge computing cloud environment, the agent learns a desirable action sequence for an efficient workload distribution. The learned action sequence is a strategy that decides the computing resource allocation and the token bucket shaping for the efficient resource provision with a per-task response time QoS (Section V).
- Lastly, through an extensive set of experiments simulating distributed FaaS on hierarchical edge clouds, we compare RACER's performance with the Application-awaRE workload Allocation method (AREA) [19], a state-of-the-art QoS-aware workload distribution algorithm. The experimental results show that RACER outperforms AREA in terms of per-task QoS violation ratio, average slowdown, and control efficiency. RACER achieves on average a 92 percent better per-task QoS violation ratio than AREA (Sections VI and VII).

The remaining part of this paper is organized as follows. In Section II, we review the existing workload distribution schemes for edge clouds. In Section III, we define the workload distribution problem as a minimization problem on 3-tier hierarchical edge clouds, and discuss the three challenging issues of the problem. Then, we present our approach to handle each of the challenging issues, i.e., the *token bucket mechanism* and *reducing decision complexity* in Section IV and *RL* in Section V. The implementation and evaluation results are explained in Section VI and VII, respectively. Then we conclude in Section VIII.

II. RELATED WORKS

We investigated several state-of-the-art workload distribution schemes for edge clouds and categorized them in terms of five features: target environment, decision variables, optimization objective, problem-solving methods, and QoS awareness, as listed in Table 1.

TABLE 1. Summary of researches on workload distribution of edge clouds.

Feature	Categories
Target environment	• Generic edge clouds [15], [16], [19], [21], [29], [30] • Hierarchical clouds [13], [14], [20], [22]–[24], [26] • Fog clouds [18], [25], [27], [28] • Other [17]
Decision variables	• Workload placement and resource allocation [14], [15], [19], [20], [22], [24], [26], [27], [29] • Workload placement [13], [16], [21], [23], [25], [28] • Other [17], [18], [30]
Optimization objective	• Minimizing average response time [13]–[17], [19]–[21], [26]–[29] • Maximizing throughput/utilization [18], [22], [30] • Other [23]–[25]
Problem solving methods	• Problem relaxation [13], [14], [17], [19], [20], [22], [25], [29] • Approximation [15], [18], [21], [23], [24], [26], [30] • Heuristic based on machine learning [27], [28] • Game theory [16]
QoS awareness	• Best-effort [13], [16], [21]–[23], [28], [29] • Response time QoS [15], [18]–[20], [24], [25], [30] • Other [14], [17], [26], [27]

A. TARGET ENVIRONMENT

Ordinary workload distribution schemes target generic edge clouds without any constraints related to network topology [15], [16], [19], [21], [29], [30]. Some other schemes consider hierarchical edge clouds, where a higher-tier cloud server plays a role of back-up for overloaded lower-tier servers [13], [14], [20], [22]–[24], [26]. Notably, such a hierarchical structure is proved to be more beneficial for peak load stability and resource utilization [13]. Some researches deal with fog clouds, where even user devices (e.g., laptops, on-board units on vehicles) dynamically contribute to form distributed elastic clouds [18], [27], [28]. In addition, Liu *et al.* [17] designed a workload distribution scheme for augmented reality applications in edge computing.

B. DECISION VARIABLES

Workload distribution schemes should naturally decide *workload placement*, which refers to determining which cloud nodes process an arriving workload. In addition, *resource allocation* is considered to be another essential decision variable because it is highly correlated with the workload placement in determining the system performance. Therefore, the majority of existing workload distribution schemes jointly control the workload placement and resource allocation [14], [15], [19], [20], [22], [24], [26], [27], [29]. Since such joint controls may worsen the problem tractability, many methods only decide the workload placement and then, accordingly, control the resource allocation by pre-defined equations or algorithms [13], [16], [21], [23], [25], [28]. Some other methods handle other decision variables related to specific problem domains. FACT jointly controls the workload placement and analytics accuracy for augmented reality applications [17]. Yu *et al.* [18] took only workload placement, but also data routing and network capacity allocation as decision variables. Xu *et al.* [30] proposed to jointly control the workload placement and virtual network function (VNF) allocation.

C. OPTIMIZATION OBJECTIVE

Most of the existing workload distribution schemes consider delay or response time as optimization objectives. The majority of them try to optimize the average response time [13]–[17], [19]–[21], [26], [28]. As a variant, some works try to minimize maximum response time [29] or computing delay [27]. Maximizing throughput or utilization is also chosen as an optimization goal in several methods [18], [22], [30]. Some researchers attempt to minimize computing and networking costs [23], [24] or maximize max-min fairness [25] along with their specific assumptions.

D. PROBLEM-SOLVING METHOD

The workload distribution is typically reduced to hard problems to obtain the optimal solution, sometimes *NP-hard* ones. Therefore, existing works mainly adopt two representative heuristic problem-solving strategies: *relaxation* and *approximation*. The relaxation-based approach firstly relaxes the original problem into a more tractable one while compromising a little bit of optimality. Then, it solves the relaxed problem via known optimization techniques such as the following:

- Linear programming [14], [22], [25]
- Coordinated descent method [17]
- A combination of greedy heuristic and convex optimization [19], [20]
- A combination of branch-and-bound, simulated annealing, and convex programming [13]
- A combination of greedy heuristic, particle swarm optimization, and semidefinite programming [29]

The approximation-based approach tries to solve the original difficult problem with a bounded time complexity while obtaining a plausible solution instead of an optimal one via known approximation algorithms as follows:

- Solving the workload distribution problem via $O(1+\epsilon)$-competitive approximation [15]
- Solving the workload distribution problem via a fully polynomial-time approximation scheme (FPTAS) [18]
- Solving the matrix chain ordering problem via a variant of the shortest path algorithm [21]
- Solving the graph embedding problem by a greedy heuristic [23]
- Solving the minimum weight maximum matching problem via a greedy heuristic [30]
- Solving the minimal capacity network optimization problem by a two-phase iterative optimization (TPIO) heuristic [24]
- Solving the workload distribution problem by using a greedy heuristic method that combines the shortest remaining time first (SRTF) and task preemption [26]

Sometimes, *game theory* is also exploited. Liu *et al.* [24] described the workload distribution as a variational inequality theory problem and solved it by an iterative proximal algorithm (IPA) to find out the *Nash equilibrium* solution. Along with the recent emergence of artificial intelligence technologies, several researchers propose heuristics based on *machine learning*. Liao *et al.* developed multi-armed bandit (MAB) control-based workload distribution schemes that cope well with information asymmetry and uncertainty in IoT and vehicular fog computing environments [27], [28].

E. QoS AWARENESS

QoS awareness is still an ongoing research issue in workload distribution problems for edge clouds. A number of existing approaches belong to the *best-effort* approach, which does not consider any QoS awareness [13], [16], [21]–[23], [28], [29]. Along with problem and application domains, several QoS metrics are used, such as communication cost [14], resource recruitment cost [27], and analytics accuracy [17]. The most popular QoS metric is response time, i.e., the delay or latency of tasks, which is addressed in a variety of service-level agreements such as the following:

- Take into account (but not guarantee) per-task response time [15]
- Take into account the aggregate utility of response time [25]
- Guarantee that the response time of a task is improved when it is offloaded [26]
- Guarantee per-task network delay [18], [30]
- Guarantee the success probability of per-task response time QoS [24]
- Guarantee the average response time QoS [19], [20]

Here, the notable thing is that any of the above works do not support a high enough level of per-task response time QoS.

Although the above schemes have made considerable achievements, they do not adequately handle all three challenging issues described in Section I, i.e., *QoS awareness*, *decision complexity*, and *uncertainty* related to the workload distribution problem in edge computing environments. Our goal is to design a more innovative solution that addresses these issues. In designing our proposed scheme, RACER, we considered three-tier hierarchical edge clouds to take advantage of the resource efficiency discussed in [13]. We adopt an average slowdown as the optimization objective from considering that a lower slowdown is likely to result in better throughput [32]. Primarily, by solving the joint decision problem of task assignment and resource allocation through the reinforcement learning method, RACER effectively handles the complex workload distribution problem. It provides a more improved per-task response time QoS than existing schemes in hierarchical edge cloud environments where the system dynamicity and uncertainty appear.

III. SYSTEM MODEL AND PROBLEM DESCRIPTION

A. SYSTEM MODEL

As illustrated in Fig. 1, we consider a hierarchically distributed edge computing infrastructure that was proved to be more beneficial for load stability and resource

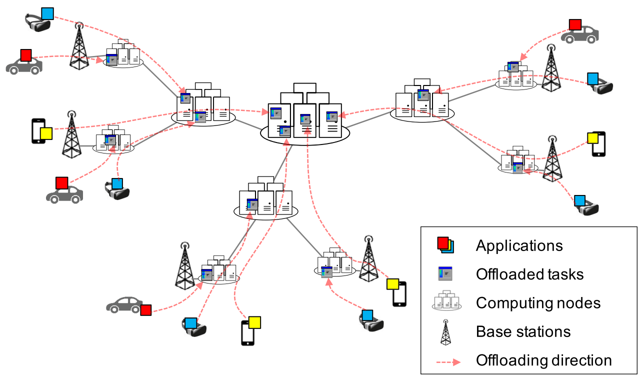


FIGURE 1. Hierarchically distributed edge computing infrastructure.

utilization [13]. We assume that user equipments (UEs) have tasks that should be offloaded to the computing nodes in the infrastructure for some reasons, such as energy savings or computation efficiency. Tier-1 nodes, which are colocated with a base station (BS), execute tasks offloaded from a group of UEs associated with the corresponding BS. Assume that a UE has a time-sensitive task that should be offloaded, and its response time is critical. A UE may prefer to offload the task to the tier-1 node colocated with its associated BS if it has sufficient computing resources because the network delay between the UE and tier-1 node is accounted only for the wireless link delay between the UE and BS. Despite the advantage in terms of network delay, tier-1 nodes should be used carefully because they generally have a limited computing capacity. A group of tier-1 nodes is connected to a tier-2 node with more computing capacity than tier-1 nodes, but much less than a tier-3 node (remote node). Tier-2 nodes can be realized as edge clouds such that each of them consists of a considerable number of computing and storage hosts. Network delays to tier-2 nodes experienced by UEs are more significant than those of tier-1 nodes, but are much smaller than that of a remote node. The remote node (e.g., a cloud computing center) has a nearly unlimited computing capacity. However, the network delay experienced by UEs is much more significant than those of the other nodes. Note that it is not always favorable to choose a lower-tier node to run offloaded tasks even though it has sufficient computing resources. We assume that each task should access input data, and the location of the data is dependent on the task type. Therefore, if a task needs to read sizable input data in the remote node, it is desirable to run it on the remote node rather than on lower-tier ones. This reduces the overall response time by shrinking the network delay required for reading the input.

Table 2 lists the symbols and notations used in our system model. We denote a set of computing nodes N and a set of directional logical links L. Each UE $u \in U$ is connected to a node $n \in N_1$ where N_1 is the set of tier-1 computing nodes. Each tier-1 computing node is connected to a node $n \in N_2$, where N_2 is the set of tier-2 computing nodes, and every tier-2 computing node is connected to a remote computing node n_r.

TABLE 2. **Summary of notations.**

Symbol	Definition
K	Set of task types
L	Set of directional logical links
N	Set of computing nodes
N_1	Set of tier-1 computing nodes
N_2	Set of tier-2 computing nodes
$N(u)$	Set of computing nodes on the path from UE u to n_r
U	Set of UEs
b_l	Network bandwidth of link l
c_n	Computing capacity of node n
d_l^p	Propagation delay of link l
D_k	Response time QoS threshold of type k task
n_r	Remote computing node
n_k^{in}	Node storing the input data of type k task
w_k^{req}	Size of request message for offloading type k task
w_k^{res}	Size of response message from offloaded type k task
w_k^{com}	Size of workload for running type k task
w_k^{in}	Size of input data required for running type k task
ϕ_{nk}	Amount of available token for type k task at node n
$c_n(t)$	Available computing capacity of node n at time slot t
$d_{uk}(t)$	Average response time of type k tasks experienced by UE u at time slot t
$d_{unk}(t)$	Average response time of type k tasks requested from UE u and offloaded to node n at time slot t
$d_{unk}^{net}(t)$	Networking delay experienced by UE u when it offloads a type k task to node n at time slot t
$f_{unk}(t)$	Weighted task response slowdown of type k tasks requested from UE u and offloaded to node n at time slot t
$g_{uk}(t)$	Number of completed type k tasks requested from UE u at time slot t
$v_{uk}(t)$	Number of completed type k tasks requested from UE u that have violated the response time QoS constraint at time slot t
$x_{unk}(t)$	Fraction of type k tasks requested from UE u that are offloaded to node n at time slot t
$y_{nk}(t)$	Token bucket size for type k tasks at node n at time slot t
$z_{nk}(t)$	Fraction of the capacity of node n allocated to type k tasks at time slot t
$\lambda_{uk}(t)$	Mean arrival rate of type k tasks requested from UE u at time slot t
$\lambda_{nk}(t)$	Mean arrival rate of type k tasks offloaded to node n from all UEs at time slot t
$\lambda_{unk}(t)$	Mean arrival rate of type k tasks offloaded to node n from UE u at time slot t

UEs run applications, each of which consists of several *fine-grained* tasks that are offloaded to computing nodes and then remotely executed.[1] We assume that tasks are categorized into a set of types K based on their properties: request message size w_k^{req}, response message size w_k^{res}, demanded computing workload w_k^{com}, input data size w_k^{in}, location of input data n_k^{in}, and response time QoS threshold D_k, where k $(\in K)$ represents a task type.

Applications in a UE generate task requests, each of which is sent to the network and then offloaded to one of the computing nodes. Once a computing node is selected to handle the requested task, it allocates a fraction of its computing capacity to run the task. Then the task reads the input data from n_k^{in} and processes the workload. After it finishes the processing, the computing node returns a task response to the UE.

Same as in [13], a higher-tier node backs up its descendant lower-tier nodes in the edge hierarchy. Considering the scale of recent data centers, we assume that the remote computing node has enough capacity to back up every descendant node. This property can be represented as an infinite capacity of the remote node for modeling convenience. Therefore, we can offload every task among the computing nodes on the path from its originated UE to the remote node. By exploiting this property, we can decompose the problem into multiple smaller ones such that each smaller subproblem is defined for an edge region. For a task requested from u, the decision space for offloading is confined to $N(u)$. For example, as in Fig. 2, to offload the tasks requested from u_{11} or u_{12}, we should choose one among the nodes n_{11}, n_{21}, and n_r. Similarly, the tasks requested from u_{41} or u_{42} must be processed among nodes n_{14}, n_{22}, and n_r. Hereafter, we redefine N, N_1, N_2, and U to denote a set of nodes, tier-1 nodes, tier-2 nodes, and UEs within an edge region, respectively, for convenience in dealing with the decomposed problem.

[1]In FaaS, each task corresponds to a fine-grained executable routine referred to as *function*.

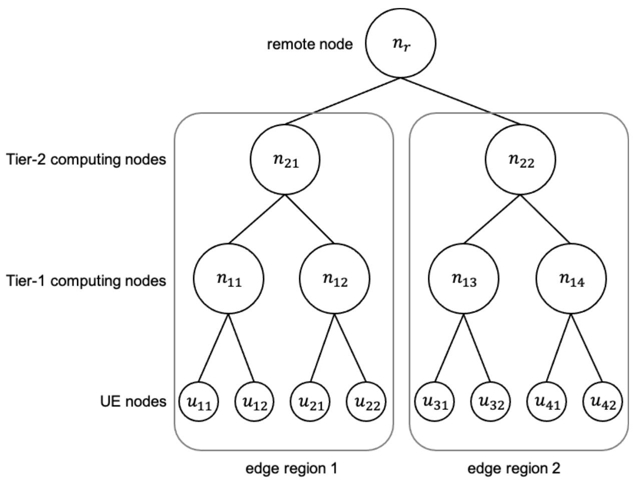


FIGURE 2. **Problem decomposition into edge regions.**

For each link $l \in L$, let b_l denote its bandwidth and d_l^p represent its propagation delay. The network delay for sending the data of size w on link l is the sum of the transmission delay w/b_l and the propagation delay d_l^p. Then the end-to-end network delay for sending data with size w from node n_i to node n_j for $n_i, n_j \in U \cup N$ is

$$d^{net}(n_i, n_j, w) \triangleq \frac{w}{\min_{l \in p(n_i,n_j)} b_l} + \sum_{l \in p(n_i,n_j)} d_l^p \tag{1}$$

where $p(n_i, n_j)$ is the set of links on the path from n_i to n_j.

Let z_{nk} be the computing capacity allocated to run a type k task at node n. Then the service time for computing a type k task with workload size w_k^{com} is

$$d_s^{com}(n, k) \triangleq \frac{w_k^{com}}{z_{nk}}. \tag{2}$$

We consider the system model with queues such that tasks offloaded to a node should wait for the earlier offloaded ones to finish if there is no sufficient computing resource. Then, to model the computing delay, we also need to consider how long a task waits in the queue until the requested computing resource is available. Let $d_w^{com}(n, k)$ be the queue waiting time for a type k task at node n. Then the overall computing delay is

$$d^{com}(n,k) \triangleq d_w^{com}(n,k) + d_s^{com}(n,k). \tag{3}$$

Many previous works ignore the waiting delay in a queue and simply consider the processing time as defined in (2). Such approaches fail to capture the accurate computing delay in real systems where offloaded tasks should wait for the earlier offloaded ones to finish their execution and release the resources. On the other hand, in [19], [24], the computing delay is modeled as a M/M/1 queueing system with Poisson traffic arrivals and exponentially distributed workloads as

$$d^{com}(n,k) \triangleq \frac{1}{\frac{z_{nk}}{w_k^{com}} - \lambda_{nk}} \tag{4}$$

where λ_{nk} is the arrival rate of type k tasks to node n. Although this model considers the queueing delay, it is limited in capturing real-world systems due to the assumptions on the fixed stochastic properties of the arrival and service processes. Later, we will propose a solution to overcome this limitation in Section IV.

The response time of a type k task with properties $(w_k^{req}, w_k^{res}, w_k^{com}, w_k^{in}, n_k^{in}, D_k)$ requested from UE u, and offloaded to node n, consists of four components such as

$$d_{unk} \triangleq d_{unk}^{req} + d_{nk}^{com} + d_{nk}^{in} + d_{unk}^{res} \tag{5}$$

where d_{unk}^{req} is the network delay required to deliver the task request message from u to n, d_{nk}^{com} is the computing delay for processing its workload, d_{nk}^{in} is the network delay for reading the input data, and d_{unk}^{res} is the network delay required to deliver the response message from n to u. They are defined as

$$d_{unk}^{req} \triangleq d^{net}(u, n, w_k^{req}), \tag{6}$$
$$d_{nk}^{com} \triangleq d^{com}(n,k), \tag{7}$$
$$d_{nk}^{in} \triangleq d^{net}(n_k^{in}, n, w_k^{in}), \tag{8}$$
$$d_{unk}^{res} \triangleq d^{net}(n, u, w_k^{res}). \tag{9}$$

B. PROBLEM DESCRIPTION

Here, we define the workload distribution problem for the system model described in Section III-A. The workload distribution handles the following two decision vectors:

- $\mathrm{x}(t) = [x_{unk}(t)]$: Type k tasks requested from UE u should be appropriately distributed among nodes $N(u)$ along with the ascending path from u to the remote node n_r in the edge cloud hierarchy. $x_{unk}(t)$ is the fraction of type k tasks requested from u that are offloaded to node n at time slot t.
- $\mathrm{z}(t) = [z_{nk}(t)]$: Each computing node n should appropriately distribute its computing capacity to each type k task. $z_{nk}(t)$ is the fraction of the capacity of node n allocated to type k tasks at time slot t.

As in [32], we consider minimizing the weighted average task response slowdown as the problem objective. For each type k task requested from UE u, and offloaded to node n, the task response slowdown is given by $d_{unk}(t)/D_k$. Let $\lambda_{uk}(t)$ be the expected number of type k task requests generated from u at time slot t, and denote $\lambda_{unk}(t) \triangleq x_{unk}(t) \cdot \lambda_{uk}(t)$ to be the arrival rate of type k task requests from u, and offloaded to n. Then the weighted task response slowdown is defined as[2]

$$f_{unk}(t) \triangleq \frac{\lambda_{unk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{d_{unk}(t)}{D_k}. \tag{10}$$

Then the *weighted average task response slowdown* is defined as $\sum_{u,n,k} f_{unk}(t)$. The workload distribution problem is that we try to minimize the weighted average task response slowdown experienced by all UEs in an edge region under several constraints, i.e.,

$$P1: \min_{\mathrm{x},\mathrm{z}} \sum_{u,n,k} f_{unk}(t) \tag{11}$$
$$s.t. \sum_{n \in N(u)} x_{unk}(t) = 1, \quad \forall u \in U, \ \forall k \in K \tag{12}$$
$$0 \le x_{unk}(t) \le 1, \quad \forall u \in U, \ \forall k \in K, \ \forall n \in N(u) \tag{13}$$
$$x_{unk}(t) = 0, \quad \forall u \in U, \ \forall k \in K, \ \forall n \notin N(u) \tag{14}$$
$$\sum_{k \in K} z_{nk}(t) \le c_n(t), \quad \forall n \in N \tag{15}$$
$$z_{nk}(t) \ge 0, \quad \forall n \in N, \ \forall k \in K \tag{16}$$
$$\lambda_{nk}(t) \cdot w_k^{com} \le z_{nk}(t), \quad \forall n \in N, \ \forall k \in K \tag{17}$$
$$d_{unk}(t) \le D_k, \quad \forall u \in U, \ \forall k \in K, \ \forall n \in N^+(u) \tag{18}$$

where (12)-(18) are the constraints to condition valid solutions. Constraints (12) and (13) suggest that, for each u and k, all the type k tasks from u should be offloaded to one of the computing nodes in $N(u)$. Constraint (14) specifies that every task should only be offloaded to one of the nodes along with the ascending path from its originated UE to the remote node in the edge cloud hierarchy. Constraints (15) and (16) provide conditions for the valid decision of resource allocation. That is, we cannot allocate more computing resources than the available capacity of a node. Constraint (17) expresses the condition that the aggregated workload of type k tasks should not exceed the computing capacity allocated by n for type k tasks. Lastly, (18) is the constraint that the average task response time should be no longer than the response time QoS threshold. In (18), $N^+(u)$ is the set of computing nodes such that $x_{unk}(t) > 0$. It is clear that the average response time $d_{unk}(t)$ depends on the computing capacity allocation $z_{nk}(t)$ according to (2). It is also influenced by the task distribution $x_{unk}(t)$ because it determines the aggregate task arrival rate to a computing node, which affects the waiting time $d_w^{com}(n,k)$ in (3).

Note that (18) supports the average QoS, not the per-task QoS, because $d_{unk}(t)$ is the average value that is derived from mean valued system parameters such as $\lambda_{uk}(t)$, $\lambda_{nk}(t)$, and $\lambda_{unk}(t)$. Therefore, it deals with the task response time in an average sense. Roughly speaking, it may result in the case

[2]We abuse the notation $\sum_{u \in U} \sum_{k \in K}(\cdot)$ by $\sum_{u,k}(\cdot)$, and similarly $\sum_{u \in U} \sum_{n \in N} \sum_{k \in K}(\cdot)$ by $\sum_{u,n,k}(\cdot)$.

where almost half of the requested tasks violate the required QoS, i.e., the task response time threshold. Undoubtedly, this will weaken the level of QoS awareness. The previous works described in Section II do not go beyond of this type of QoS awareness. In Section IV, we will adopt a well-known token bucket mechanism and reformulate $P1$ to a novel optimization problem that supports a more improved level of QoS such that we bound the per-task response time instead of the average one.

Note that the dimension of the decision vectors of $P1$ may be huge. The dimension of the offloading distribution vector $\mathrm{x}(t) = [x_{unk}(t)]$ is $|U| \times |N(u)| \times |K|$, i.e., $3 \times |U| \times |K|$ in the 3-tier hierarchy. The dimension of the computing resource allocation vector $\mathrm{z}(t) = [z_{nk}(t)]$ is $|N| \times |K|$. Therefore, the overall dimension of the decision space is $3 \times |U| \times |N| \times |K|^2$, which may be problematic, especially for a large U. To address this issue, we will exploit a problem relaxation technique to reduce the decision complexity by compromising a little bit of optimality in Section IV-B.

IV. PROBLEM REFORMULATION

A. SUPPORTING PER-TASK QoS WITH TOKEN BUCKETS

To devise a per-task response time QoS, we augment our system model with the well-known token bucket mechanism and transform $P1$ to a new optimization problem that supports the augmented model. For each task type k, each computing node n performs the admission control of whether it accepts executing the task requested from UEs. It maintains $|K|$ token buckets such that each token generation rate is set to be equivalent to the allocated capacity $z_{nk}(t)$. For each time, it decides the size of the token bucket y_{nk} for each k. When a type k task arrives, the node checks whether sufficient tokens are available in the bucket. If the amount of tokens ϕ_{nk} is greater than the workload of the requested task w_k^{com}, the offloading is accepted, and the task is added to a waiting queue for type k workloads. Then the amount of tokens decreases as $\phi_{nk} \leftarrow \phi_{nk} - w_k^{com}$. Otherwise, if $\phi_{nk} < w_k^{com}$, the task is rejected and then redirected to the parent node of n in the hierarchy. This implies that the remote node n_r is the last resort of all redirected tasks rejected by lower-tier nodes. Since the queueing workloads are served at the rate of z_{nk}, the computing delay must be bounded by y_{nk}/z_{nk}, which is strictly held by the token bucket mechanism. Note that this is valid no matter what probability distributions are followed by the task arrival processes and workload sizes.

From the augmented system model with the token bucket, our workload distribution problem should also determine $\mathrm{y}(t) = [y_{nk}(t)]$. Let us define $d_{unk}^{net}(t)$ to be the networking delay experienced by UE u if a type k task is offloaded to node n at time slot t as

$$d_{unk}^{net}(t) \triangleq d_{unk}^{req}(t) + d_{nk}^{in}(t) + d_{unk}^{res}(t). \tag{19}$$

Then (18), i.e., the average task response time constraint, can be rewritten as

$$d_{nk}^{com}(t) \le D_k - d_{unk}^{net}(t), \quad \forall u \in U,\ \forall k \in K,\ \forall n \in N^+(u). \tag{20}$$

According to the guaranteed delay bound property of the token bucket mechanism [31], we have

$$d_{nk}^{com}(t) \le \bar{d}_{nk}^{com}(t) \le \frac{y_{nk}(t)}{z_{nk}(t)} \tag{21}$$

where we denote $\bar{d}_{nk}^{com}(t)$ to be the *maximum experienced* computing delay at time slot t. And then, we can tighten (21) harder such that the maximum experienced computing delay should be bounded as

$$\frac{y_{nk}(t)}{z_{nk}(t)} \le D_k - d_{unk}^{net}(t),\ \forall u \in U,\ \forall k \in K,\ \forall n \in N^+(u) \tag{22}$$

$$y_{nk}(t) \ge 0,\ \forall k \in K, \forall n \in N^+(u). \tag{23}$$

Constraint (22) can be rewritten as

$$y_{nk}(t) \le (D_k - d_{unk}^{net}(t)) \cdot z_{nk}(t), \quad \forall u \in U, \forall k \in K, \forall n \in N^+(u) \tag{24}$$

which is a new constraint that specifies the task response time QoS threshold by which we can specify the per-task QoS instead of the average one. Now, we revise $P1$ to a new one $P2$ that controls $y_{nk}(t)$ as well as $x_{unk}(t)$ and $z_{nk}(t)$ with new constraints (23) and (24) as

$$P2: \min_{\mathrm{x,y,z}} \sum_{u,n,k} f_{unk}(t) \quad \text{s.t. } (12), (13), (14), (15), (16), (17), (23), \text{ and } (24). \tag{25}$$

B. REDUCING PROBLEM COMPLEXITY

The decision complexity of $P2$ becomes larger than that of $P1$ due to the introduction of new decision variables $y_{nk}(t)$. Since the dimension of the decision vector $\mathrm{y}(t) = [y_{nk}(t)]$ is $|N| \times |K|$, that of the decision space for $P2$ is $3 \times |U| \times |N|^2 \times |K|^3$ $(= (|N| \times |K|) \times (3 \times |U| \times |N| \times |K|^2))$.

Inspired from the assumption that $|U| \gg |N| > |K|$, we remove $x_{unk}(t)$ from the decision variables of $P2$ in order to eliminate $|U|$ from the dimension of the decision vector. Instead, $x_{unk}(t)$ is accordingly determined as

$$x_{unk}(t) \cdot \lambda_{uk}(t) = \frac{\hat{\lambda}_{unk}(t)}{\sum_{\breve{u}} \hat{\lambda}_{\breve{u}nk}(t)} \cdot \bar{\lambda}_{nk}(t), \quad \forall u \in U, \forall k \in K, \forall n \in N^+(u), \tag{26}$$

where $\bar{\lambda}_{nk}(t) = z_{nk}(t)/w_k^{com}$, and $\hat{\lambda}_{unk}(t)$ is the amount of type k tasks from user u arrived at node n, but is not yet offloaded. Note that $\hat{\lambda}_{unk}(t)$ includes also the redirected tasks rejected by lower tier nodes.

Condition (26) suggests that for each pair of node and task type, the tasks requested from different UEs are accepted proportionally to the arrival rates from each of the UEs to n. Assigning $z_{nk}(t)/w_k^{com}$ to be $\bar{\lambda}_{nk}(t)$ implies that we try to control the aggregate workload of type k tasks offloaded to n (i.e., $\lambda_{nk}(t) \cdot w_k^{com}$) to be the same as the allocated capacity (i.e., $z_{nk}(t)$).

The following lemma shows that the solution of $P2$ with (26) is not far from the original solution under some minor assumptions. Let f^* and $\Delta D_{nk}^{net}(t) \triangleq \max_u d_{unk}^{net}(t) - \min_u d_{unk}^{net}(t)$ denote the objective value of the optimal solution of $P2$ and the largest difference of network delays experienced by u for the type k task offloaded to n, respectively.

Lemma 1: The objective value achieved by an optimal solution of $P2$ with the additional constraint (26) is bounded by $f^* + \sum_{n,k} \Delta D_{nk}^{net}(t)/D_k$.

Proof: See Appendix. □

According to Lemma 1, if the network delays experienced by UE nodes are not significantly different, i.e., $\Delta D_{nk}^{net}(t)$ is small, then our approach can achieve a desirable suboptimal solution. While compromising a bit of optimality, we can take advantage of a significant reduction of problem complexity. Since we do not have to control $x_{unk}(t)$ anymore, the dimension of the decision vector is remarkably reduced to $|N|^2 \times |K|^2$. The trade-off between the complexity and optimality becomes better as $|U|$ increases.

Eventually, we reformulate $P2$ to a new one $P3$ with (26) as

$$P3: \min_{\mathrm{y,z}} \sum_{u,n,k} f_{unk}(t)$$
$$\text{s.t. } (12), (13), (14), (15), (16), (17), (23), \text{and } (24) \quad (27)$$

where $x_{unk}(t)$ is determined by the condition (26).

V. LEARNING-BASED APPROACH

Inspired from some recent works applying RL to mobile edge computing researches [27], [28], [33], we consider RL as a solution to cope with the system dynamicity and uncertainty discussed as the third challenge in Section I.

RL is a type of interaction-driven learning paradigm consisting of two entities: agent and environment [34]. For each interaction at time slot t, the agent observes the state $s(t)$ of the environment and decides an action $a(t)$ based on its policy function. As a result of the action, the environment issues the agent with a reward $r(t)$ as feedback to evaluate the policy function's quality. Repeating this observe-action-reward process, the agent incrementally learns a desirable action strategy by improving its policy function to maximize the expected cumulative discounted reward $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(t)]$ where $\gamma \in [0, 1)$ is the discount factor. Recently, deep neural network technologies are exploited for boosting policy improvement procedures.

We exploit RL to sequentially control the token bucket sizes $y_{nk}(t)$ and the resource allocations $z_{nk}(t)$ such that it solves $P3$ in a long time average sense. Therefore, we try to minimize

$$\lim_{T\to\infty} \frac{1}{T} \sum_{t=0}^{T-1} \gamma^t \sum_{u,n,k} f_{unk}(t). \quad (28)$$

In our experiment, we set γ to be 0.99. Note that when $\gamma = 0$, the objective is the same as (27).

A. CONTROL AND LEARNING ARCHITECTURE

Our proposed system, RACER, consists of two entities: *controller* and *agent* for each edge region. The RACER controller maintains $|N_1 \cup N_2| \cdot |K|$ token buckets, each of which corresponds to a pair of node $n \in N_1 \cup N_2$ and task type $k \in K$ to control the admission of tasks. The tokens for a type k task are generated at the rate of $z_{nk}(t)$. The maximum amount of tokens is bounded by the bucket size $y_{nk}(t)$. Each node n has $|K|$ queues that correspond to $|K|$ task types, respectively. Once node n receives a request for the offloading of type k task from a UE, the controller checks whether the token for type k is enough. If the token size ϕ_{nk} is greater than the workload of the task w_k^{com}, the request is admitted. Then the task is pushed to the queue for its corresponding type. The tasks in a queue are served in FIFO fashion with the same service rate as $z_{nk}(t)$. If the request is rejected due to a lack of tokens, it is redirected to the parent node of n in the hierarchy. Then the parent repeats the same admission process. With exception, the remote node n_r does not conduct the admission control because it is assumed that $c_{n_r} \gg c_n$ for $n \in N_1 \cup N_2$ in Section III-A. Note that a large c_{n_r} leads to a small computing delay, but it does not mean the satisfaction of response time QoS, possibly due to the long network delay from UE.

The RACER agent controls $\mathrm{y}(t) = [y_{nk}(t)]$ and $\mathrm{z}(t) = [z_{nk}(t)]$ for every time slot t. Before the agent starts to play its role in the edge region, it is *trained* using RL, rather than *programmed* as a handcrafted algorithm. The RL agent repeats the observe-action-reward process for every t. For each t, the agent receives the observation data $s(t)$ representing the current state of the edge region and then gives an action $a(t) = (\mathrm{y}(t), \mathrm{z}(t))$ to the RACER controller. Then the agent receives the reward $r(t)$ that is used to evolve its policy from the controller.

Note that we can make sure the constraints from (12) to (17) and (23) are satisfied under the above control architecture. However, it is somewhat challenging to meet (24), the QoS constraint, because the network delay $d_{unk}^{net}(t)$ is generally assumed to be unknown at the time of control. One possible way is to estimate the $d_{unk}^{net}(t)$ for all the tuples of $u \in U, n \in N, k \in K$. However, doing this is burdensome and difficult to be accurate. Rather than enforcing the constraint directly, we introduce the violation cost to the reward $r(t)$ to train the agent trying to satisfy it, which is elaborated in Section V-D.

We summarize our approach to learn the control policy in Algorithm 1. The detailed descriptions on how to build $s(t)$, $\tilde{a}(t)$, $a(t)$, and $r(t)$ are in the following subsections. The detail of the PPO algorithm is omitted since it is not a focus of this paper.

B. STATE DESIGN

At the start of each time slot t, the agent observes a state input vector $s(t) = (\tilde{\Lambda}(t), \mathrm{b}(t), \mathrm{c}(t), \tilde{\mathrm{c}}(t), \mathrm{q}(t), \mathrm{o}(t))$. $\tilde{\Lambda}(t) = [\tilde{\lambda}_{nk}(t)]$ is a $(4\times|N_1|\times|K|)$-dimensional vector that represents a measured statistics tuple, (mean, max, min, variance), of the task arrival rate for each $n \in N_1$ and $k \in K$. $\mathrm{b}(t) = [b_{ij}(t)]$ is

Algorithm 1 RACER Approach to Learn the Control Policy

```
RACER controller:
1: Set the initial state $s(0)$.
2: $t = 0$.
3: repeat
4:     Collect the state $s(t)$ and pass it to the agent.
5:     Get the action $a(t)$ from the agent and update the
       token bucket sizes and resource allocations of the nodes.
       (Then, the nodes run the workload distribution.)
6:     Compute the reward $r(t)$ and pass it to the agent.
7:     $t = t + 1$.
8: until the agent stops learning.
RACER agent:
1: repeat
2:     Receive the state $s(t)$ from the controller.
3:     Determine the nomalized action vector $\tilde{a}(t)$ from $s(t)$
       using the policy neural network model.
4:     Compute the action $a(t)$ and pass it to the controller.
5:     Receive the reward $r(t)$ from the controller.
6:     Update the policy and value neural network models
       using the PPO algorithm [35].
7: until the mean episode reward is converged.
```

a $(4 \times (4 \times |N_1| + 2))$-dimensional vector that represents the measured statistics tuple of the network bandwidth between the nodes in an edge region, including the groups of UEs and the remote node. We assume that a group of UEs under the same BS experience the same dedicated bandwidth. Then for each pair of a UE group and the corresponding tier-1 node in N_1, we only measure two statistics tuples, each one for uplink and downlink, respectively. $\mathrm{c}(t) = [c_n(t)]$ is a $(4 \times |N|)$-dimensional vector that has the measured statistics tuple of the available computing capacity of each $n \in N$. $\tilde{\mathrm{c}}(t) = [\tilde{c}_{nk}(t)]$ is a $(4 \times |N| \times |K|)$-dimensional vector that represents the measured statistics tuples of how much computing resources have been used for each $n \in N$ and $k \in K$. $\mathrm{q}(t) = [q_{nk}(t)]$ and $\mathrm{o}(t) = [o_{nk}(t)]$ are $(|N| \times |K|)$-dimensional vectors that specify the current lengths of waiting queues and amounts of tokens, respectively, for each $n \in N$ and $k \in K$.

A notable thing here is that we exploit the measured statistics tuples in constructing $s(t)$. For each time slot t, the tuples are computed from the statistics measured at the previous time slot, $t - 1$. With an LSTM[3] layer in our neural network design, it makes the agent more robust to any uncertainties caused by changes in task arrival rates, network bandwidths, and resource utilization.

C. ACTION DESIGN

For each time slot t, after receiving $s(t)$, the agent yields an action vector $a(t) = (\mathrm{y}(t), \mathrm{z}(t))$ where both the token bucket size vector $\mathrm{y}(t) = [y_{nk}(t)]$ and computing capacity allocation vector $\mathrm{z}(t) = [z_{nk}(t)]$ are $(|N_1 \cup N_2| \times |K|)$-dimensional vectors. Then the controller behaves as described in Section V-A by using the updated token bucket sizes and computing capacity allocations.

The feasible ranges of $y_{nk}(t)$ and $z_{nk}(t)$ are likely to be varied due to the changes in environment parameters such as D_k, $d^{net}_{unk}(t)$, $c_n(t)$, and so on. Such variability is generally harmful to the learning performance and flexibility of the agent. For this reason, we designed the internal neural network model such that the agent yields a normalized action vector $\tilde{a}(t) = (\tilde{\mathrm{y}}(t), \tilde{\mathrm{z}}(t))$, where $\tilde{\mathrm{y}}(t) = [\tilde{y}_{nk}(t)]$ and $\tilde{\mathrm{z}}(t) = [\tilde{z}_{nk}(t)]$ have values between [0, 1]. Then $a(t)$ is computed by postprocessing $\tilde{a}(t)$ as

$$y_{nk}(t) = \tilde{y}_{nk}(t) \cdot D^{com} \cdot c_n(t) \tag{29}$$
$$z_{nk}(t) = \tilde{z}_{nk}(t) \cdot c_n(t) \tag{30}$$

where D^{com} is a predefined constant that specifies the desirable upper bound on the computing delay when we assume that all the computing resources of a node n are allocated to that task type.[4]

D. REWARD DESIGN

At the end of each time slot t, the agent receives the reward $r(t) = -(h(t)+v(t))$, where $h(t)$ is the weighted average task response slowdown, and $v(t)$ is the violation cost substituted for the per-task QoS constraint (24). Note that we use the *negative* reward because $P3$ is a minimization problem.

To compute $r(t)$, the controller collects $d_{uk}(t)/D_k$, $g_{uk}(t)$, and $v_{uk}(t)$ for each UE $u \in U$ and task type $k \in K$. $d_{uk}(t)$ is the average response time of type k tasks experienced by u, and therefore, $d_{uk}(t)/D_k$ is the average slowdown experienced by the UE. $g_{uk}(t)$ is the number of type k tasks requested from u and completed during time slot t, and $v_{uk}(t)$ is the number of completed tasks that violated the QoS constraint (24). Then $h(t)$ and $v(t)$ are computed as

$$h(t) = \sum_{u,k} \frac{g_{uk}(t)}{\sum_{\breve{u},\breve{k}} g_{\breve{u}\breve{k}}(t)} \cdot \frac{d_{uk}(t)}{D_k} \tag{31}$$
$$v(t) = \sum_{u,k} \xi_k \cdot v_{uk}(t) \tag{32}$$

where ξ_k is a unit cost of each violation. Note that $\mathbb{E}[h(t)]$ is the same as the objective function of $P3$. The signaling overheads for constructing $r(t)$ are not significant. For each time slot, each UE just needs to deliver $O(|K|)$ messages to the controller for reporting $d_{uk}(t)/D_k$, $g_{uk}(t)$, and $v_{uk}(t)$. Such overheads may be enveloped into those of ordinary requests for task offloading.

E. LEARNING ALGORITHM

We used the proximal policy optimization (PPO) algorithm to train the agent [35]. It is one of the most widely used policy gradient-based deep RL methods for continuous control tasks

[3]long short term memory

[4]We can set D^{com} by letting D^{com} be sufficiently large such that $D^{com} \geq \max_{u,n,k} D_k - d^{net}_{unk}$.

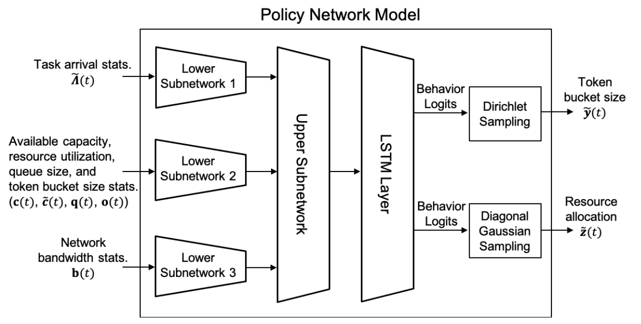

Policy Network Model
Task arrival stats. $\tilde{\Lambda}(t)$
Available capacity, resource utilization, queue size, and token bucket size stats. $(\mathbf{c}(t), \tilde{\mathbf{c}}(t), \mathbf{q}(t), \mathbf{o}(t))$
Network bandwidth stats. $\mathbf{b}(t)$
Lower Subnetwork 1
Lower Subnetwork 2
Lower Subnetwork 3
Upper Subnetwork
LSTM Layer
Behavior Logits
Dirichlet Sampling
Behavior Logits
Diagonal Gaussian Sampling
Token bucket size $\tilde{y}(t)$
Resource allocation $\tilde{z}(t)$

FIGURE 3. Policy network model of our approach.

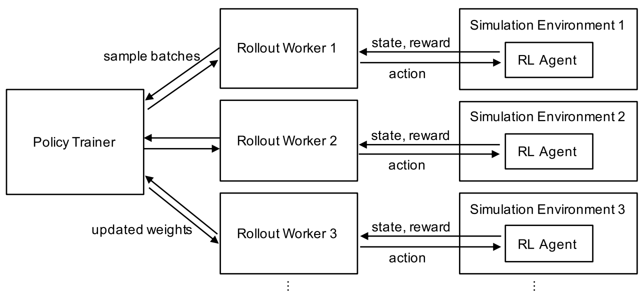


FIGURE 4. Architecture for training RL agent.

such as robotics [36] and games [37]. To reason why we chose PPO, we provide two arguments as follows:

- Obviously, $P3$ is a continuous control task for which the policy gradient methods are more preferred than value-based methods.
- Traditional policy gradient methods such as A3C [38] and DDPG [39] suffer from a *destructive policy update problem* [35], which refers to the case where a newly trained policy abruptly changes the policies previously trained. It means that an agent may forget actions learned in the past due to learning a new action. PPO overcomes this problem by clipping excessive policy gradients so that an agent is much more likely to preserve learned policy in the past [40].

F. NEURAL NETWORK DESIGN

Since we chose PPO as a training method, we designed a policy approximator function that is a neural network model, as depicted in Fig. 3. The model consists of three lower subnetworks, an upper subnetwork, a LSTM layer, and two sampling logics.

The three lower subnetworks take $s(t)$ as inputs and feed their outputs to the upper subnetwork. The first lower subnetwork receives the task arrival statistics vector $\tilde{\Lambda}(t)$. The second one takes four statistics vectors $(\mathrm{c}(t), \tilde{\mathrm{c}}(t), \mathrm{q}(t), \mathrm{o}(t))$, each of which corresponds to the available computing capacity, resource utilization, waiting queue backlogs, and available tokens, respectively. The last one receives the network bandwidth statistics vector $\mathrm{b}(t)$. Each subnetwork has a single layer with 128 perceptrons. One may recognize that the lower subnetworks extract the latent features about task arrivals, resource allocation status, and network status, respectively.

The upper subnetwork is a single layer network consisting of 256 perceptrons. It takes as inputs the latent features produced by the lower subnetworks, aggregates them, and then produces a new latent feature vector that captures the correlations between the elements of $s(t)$. The output of the upper subnetwork is fed into the LSTM layer with 64 cells, which recognizes temporal correlations that appeared in the 30 consecutive state inputs, i.e., $\{s(t), s(t-1), \ldots, s(t-29)\}$. The LSTM generates two behavior logit vectors that are passed to the samplers.

The two sampling logics are responsible for generating $\tilde{\mathrm{y}}(t)$ and $\tilde{\mathrm{z}}(t)$. The diagonal Gaussian sampler generates $\tilde{\mathrm{y}}(t)$ from the behavior logits for the token bucket size. Similarly, the Dirichlet sampler produces $\tilde{\mathrm{z}}(t)$ from the behavior logits for the computing resource allocation.

We also designed the value approximator function such that it shares all the components of the policy approximator, except for the last output layer. Instead of the samplers, the approximator uses a single linear perceptron that produces an expected return value for the last 30 state inputs.

VI. IMPLEMENTATION

In this section, we describe how we implemented RACER using *RLlib* [41] and *DFaaSCloud* [42], [43]. RLlib is an RL framework that supports massively distributed parallel learning, which has proven scalability such that we can use thousands of CPUs and hundreds of GPUs for large-scale RL. The DFaaSCloud is a discrete event-driven simulator for simulating distributed FaaS on edge cloud systems. It is an extension of CloudSim [44], a well-known simulation framework for cloud computing systems.

Following the RLlib APIs concept, we developed RACER using the training architecture illustrated in Fig. 4, where there are three types of entities: policy trainer, rollout worker, and simulation environment. The policy trainer and rollout workers are blocks for training the RL agent. The simulation environment is responsible for simulating the system model. By concurrently executing multiple pairs of the rollout worker and simulation environment, RLlib provides a significant speed-up for training an agent.

A. POLICY TRAINER AND ROLLOUT WORKERS

Using RLlib, we implemented the policy trainer and the rollout worker blocks including the PPO algorithm and neural network model.[5] Each rollout worker generates sample batches of experiences by repeating the observe-action-reward process in its corresponding environment block. A rollout worker interacts with its corresponding environment using REST APIs on an HTTP connection because they are implemented in different frameworks, RLlib and DFaaSCloud.

The policy trainer coordinates multiple rollout workers to train the RL agent. It collects sample batches of experiences

[5]In RLlib, the rollout worker can be seen as an *avatar* that exploits the policy copied from the trainer block.

TABLE 3. Network parameters.

	Bandwidth (Mbps)		Delay (ms)	
Link	Downlink	Uplink	Downlink	Uplink
UE to tier-1	1000	1000	1	1
Tier-1 to tier-2	500	500	10	10
Tier-2 to remote	100	100	100	100

Mbps = megabits per second, ms = milliseconds.

from the multiple rollout workers and improves the policy and value approximator functions. The improved neural approximators are copied to the rollout workers periodically for more improved experiences.

By repeating the sample batch collection and the policy improvement process, the policy trainer evolves the neural network model for the agent. We finish the training if the mean episode reward is converged to a desirable level.

B. SIMULATION ENVIRONMENT AND RL AGENT

Using DFaaSCloud, we implemented the simulation environment block, including the RACER controller and the RL agent. The RACER controller actually controls the token bucket sizes and resource allocations based on the $a(t) = (\mathrm{y}(t), \mathrm{z}(t))$ received from the agent. The agent plays a kind of broker that is in charge of interacting with its corresponding rollout worker, which has the neural network model for the agent. For each time slot t, it delivers state $s(t)$, received from the controller, to the rollout worker, and then, receives action $\tilde{a}(t) = (\tilde{\mathrm{y}}(t), \tilde{\mathrm{z}}(t))$. From $\tilde{a}(t)$, the agent computes $a(t)$ by using (29) and (30) and gives it to the controller. It is also responsible for passing reward $r(t)$ to the rollout worker. As mentioned earlier, the exchanges of $s(t)$, $\tilde{\mathrm{a}}(t)$, and $r(t)$ are conducted through the REST APIs on an HTTP connection.

VII. EVALUATION

We conducted a simulation-based evaluation to compare the performance of RACER with those of two other approaches: *AREA* and a *static* workload distribution. AREA [19] is a state-of-the-art algorithm that efficiently distributes the workloads from UEs to a networked edge cloud system while guaranteeing the average response time QoS. The static method distributes workloads according to a deterministic rule that maps a pair of UE and task type $(u, k) \in (U, K)$ to a node $n \in N$. For example, if we specify a rule $r : U \times K \rightarrow N = \{(u_0, 0, n_1), (u_1, 1, n_2), (u_2, 2, n_0)\}$, type 0 (1 or 2) tasks from u_0 (u_1 or u_2) are always offloaded to node n_1 (n_2 or n_0). Later, we will summarize their specific settings in Section VII-B.

A. SIMULATION SETUPS

As noted earlier, we implemented the simulation environment using the DFaaSCloud framework. We built a hierarchical edge cloud that consists of three tier-1 computing nodes, one tier-2 node, and one remote node. Each tier-1 node is attached to a BS by which UEs access the edge cloud. Each BS provides broadband access to 100 UEs that are the sources of tasks. Table 3 lists the average bandwidth and propagation delay of the logical link experienced by each task. To reflect the advantage of the short delay on near-client edges, we make the lower-tier links provide a higher bandwidth and shorter propagation delay. Each link has a symmetric bandwidth and propagation delay for the sake of easy evaluation. Note that every task from a UE can be offloaded to one of the UE's ancestor nodes, as discussed in Section III-A. The network parameters and the offloading location of a task significantly affect the network delay experienced by the task.

Each tier-1 node has 10 CPU cores, and the tier-2 and the remote nodes have 40 and 1000 cores, respectively. The computing capacity of each CPU core is 10^4 *million instructions per second* (MIPS). Thus, the computing capacity, c_n, of the tier-1, tier-2, and remote node becomes 10^5, 4×10^5, and 10^7 MIPS, respectively.

We consider five task types whose parameters are listed in Table 4. We assume that all the task types have the per-task response time QoS bound D_k such that any response exceeding the bound is considered a task failure. Therefore, we set task parameters with somewhat deterministic computing workloads so that the response time QoS bound should be met if the task is offloaded to a proper node in a low load utilization. For every task type k, we fix the computing workload size and input data size to be the same values as $w_k^{com} = 1000$ MIs and $w_k^{in} = 10$ MB for ease of evaluation. We also assume that each task offloaded to a node can use only one CPU core. That is, the amount of CPU resource that a task uses is predetermined, which is an ordinary constraint in the current FaaS services [10]–[12].

We vary the aggregate workload of each task type by adjusting its arrival rate. For each k, the per-task response time QoS bound, input data location, and preferred node set for the QoS are highly correlated with each other. In Table 4, d_{1k}, d_{2k}, and d_{3k} are the per-task response time experienced by type k tasks when they are offloaded to a tier-1, tier-2, and remote node, respectively, in a low load utilization. According to this setup, one can easily identify that type 2, 3, and 4 tasks should be offloaded to the remote, tier-1, and tier-2 nodes, respectively, to meet the QoS constraint. Type 0 and 1 tasks have both tier-1 and tier-2 nodes as their preferred node set for QoS. The difference between them is that the most preferred node of a type 0 (1) task is tier-1 (tier-2) because its input data location is UE (tier-2). From this, one can recognize that it is better to offload a task near its input data location. Note that the QoS constraint cannot be satisfied, even when the task is offloaded to a preferred node if it suffers from an excessive workload and large queueing delay.

Every UE generates tasks. For each UE u, task type k, and time slot t, tasks are generated as the *Poisson* arrival process with an average rate $\lambda_{uk}(t)$. For ease of evaluation, we define eight profiles of the task arrival rates, as shown in Table 5, each of which represents a specific resource utilization scenario. In the table, λ_k denotes the aggregate arrival rate of type k tasks from 100 UEs attached to each tier-1 node (actually, its corresponding BS). We assume that the users generate

TABLE 4. Task parameters.

$k \in K$	w_k^{com} (MIs)	w_k^{in} (MB)	D_k (ms)	Input data location	d_{1k} (ms)	d_{2k} (ms)	d_{3k} (ms)	Preferred node set for QoS
0	1000	10	400	UE	208	352	1584	{tier-1, tier-2}
1	1000	10	400	tier-2	314	154	1562	{tier-2, tier-1}
2	1000	10	600	remote	1218	1234	482	{remote}
3	1000	10	300	UE	208	352	1584	{tier-1}
4	1000	10	200	tier-2	314	154	1562	{tier-2}

MIs = million instructions, MB = megabytes, ms = milliseconds.

TABLE 5. Arrival rate profiles.

Profiles	λ_0	λ_1	λ_2	λ_3	λ_4	ρ_1	ρ_2
p_0	36.6	33.3	33.3	33.3	33.3	0.70	0.5
p_1	46.7	33.3	33.3	33.3	33.3	0.80	0.5
p_2	46.7	33.3	33.3	43.5	33.3	0.90	0.5
p_3	47.4	33.3	33.3	47.6	33.3	0.95	0.5
p_4	50.0	33.3	33.3	50.0	33.3	1.00	0.5
p_5	55.2	33.3	33.3	50.0	33.3	1.05	0.5
p_6	55.2	33.3	33.3	54.9	33.3	1.10	0.5
p_7	61.3	25.0	33.3	58.8	25.0	1.20	0.38

tasks uniformly for simplicity. ρ_1 and ρ_2 are the aggregate loads for the tier-1 and tier-2 node, respectively, assuming every task is offloaded to their most preferred node. Each profile is designed to observe whether RACER performs the desired behavior or not. For example, when the task arrivals follow the profile p_5, RACER should control $a(t)$ such that every type 3 task is preferentially offloaded to the tier-1 node. Type 0 tasks are also carefully distributed to tier-1 and tier-2 nodes in order to prevent an excessive computing delay at the tier-1 node. On the other hand, in the case of p_0, RACER must feel free to offload every task to its most preferred node. We increase the load on the tier-1 nodes as we increase the profile number. We set ρ_2 to be lower in profile p_7 than the other profiles so that the tier-2 node has sufficient room to accommodate the overloaded tasks in tier-1 nodes.

We set the simulation time and control interval to 40 minutes and 1 second, respectively, which means that the time slot t increases by 1 second from 0 to 2400 seconds. In order to cause uncertain and dynamic system behavior, we realize a kind of non-stationarity by using different arrival rate profiles for every 5 minutes during the simulation. The arrival rate profiles in Table 5 arise one after another in the ascending order of their indices in the simulation.

B. ALGORITHM-SPECIFIC SETUPS

In addition to RACER, we implemented two other workload distribution algorithms: The AREA and a static method. To provide a better evaluation of the simulation results, we note some algorithm-specific settings for each implementation.

1) RACER

We trained the RACER agent using the following settings:

- We used a single policy trainer, 20 rollout workers, and 20 simulation environments.
- The length of each episode is 1000 time slots.
- In each episode, for each task type k, we set the aggregate task arrival rate from a group of 100 UEs, λ_k, to a uniformly random value between 20 and 70. Note that, in training, we did not provide any information about the arrival rate profiles in Table 5.
- We trained a different RACER agent with varying violation unit costs, ξ_k (see Section V-D): 0, 5, 10, 50, 100, and 500. Later, we will discuss the impact of ξ_k on performance.

With the above settings, we conducted the training for 5000 episodes to get converged policies.

2) AREA

Referring to [19], we implemented AREA to control $\mathrm{x}(t) = [x_{unk}(t)]$ and $\mathrm{z}(t) = [z_{nk}(t)]$ of $P1$ taking as inputs $\lambda_{uk}(t)$, $c_n(t)$, d_{nk}^{com}, d_{nk}^{in}, d_{unk}^{req}, and d_{unk}^{res}, for each $u \in U$, $n \in N$, and $k \in K$. Among these inputs, d_{nk}^{com}, d_{nk}^{in}, d_{unk}^{req}, and d_{unk}^{res} should be computed from other system parameters. In particular, we have to recompute d_{nk}^{com} for each time slot t, since it is determined by the resource allocation control of the previous time slot, i.e., $\mathrm{z}_{nk}(t-1)$. Therefore, we also implemented a preprocessing module and added it to the original AREA algorithm.

AREA is a heuristic approach that consists of three sub-algorithms. Firstly, it greedily controls $\mathrm{x}(t)$ based on only the network delay (*i.e.*, $d_{nk}^{in} + d_{unk}^{req} + d_{unk}^{res}$). Then, the initial $\mathrm{x}(t)$ is improved by repeatedly changing some assignments, i.e., $x_{unk}(t)$, while considering the computing delay d_{nk}^{com}. Lastly, as a result of the greedy swaps, when the distribution reaches an equilibrium, where any swapping cannot improve the objective, it determines $\mathrm{z}(t)$ through a nonlinear convex optimization, for which we use the *trust-region constrained* solver imported from the *SciPy* library.[6]

It may be difficult for AREA to support real-time workload distribution when U becomes large. For each time slot t, its greedy swapping requires $O(|U| \times |N| \times |K|)$ iterations, and the nonlinear convex optimizer also causes large execution overheads.

3) STATIC METHOD

As we discussed earlier, the static method controls $\mathrm{x}(t) = [x_{unk}(t)]$ of $P1$ according to a static rule that maps a pair of UE and task type $(u, k) \in (U, K)$ to a node $n \in N$. In the

[6]https://scipy.github.io/devdocs/tutorial/optimize.html

simulation, we specify a rule r such that every type k task is offloaded to its most preferred node as

$$r : U \times K \rightarrow N = \{(u, k, n_{uk}^{pref}) | \forall u \in U, \forall k \in K\} \quad (33)$$

where $n_{uk}^{pref} \in N(u)$ is the most preferred node for task type k specified in Table 4. In fact, for each task type k, the UEs attached to the same BS have the same most preferred node. Thereby, we need to specify just $|N_1| \times |K|$ rules, since the number of BS is the same as that of tier-1 nodes.

After $\mathrm{x}(t)$ is decided, the static allocation method controls $\mathrm{z}(t) = [z_{nk}(t)]$ for each node n and task type k as

$$z_{nk}(t) = \frac{\sum_u w_k^{com} \cdot \lambda_{uk}(t) \cdot x_{unk}(t)}{\sum_{\breve{u},\breve{k}} w_{\breve{k}}^{com} \cdot \lambda_{\breve{u}\breve{k}}(t) \cdot x_{\breve{u}n\breve{k}}(t)} \cdot c_n(t), \quad (34)$$

which allocates the resource of a node to each task type proportionally to its aggregate workload.

As one expected, the static method is not appropriate for time-varying task arrivals. We used this as a baseline to emphasize the importance of handling the dynamicity of system behavior.

C. RESPONSE TIME QoS

In this section, we evaluate the response time QoS of RACER, AREA, and the static method through the violation ratio, which is the ratio of the number of QoS violations to that of the total requested tasks. The QoS violation ratios of RACER, AREA, and the static method are measured with varying task arrival profiles from p_0 to p_7 shown in Table 5. Hereafter, we categorize the task arrival profiles into two types by the load utilization of the tier-1 node for ease of presentation. Profiles p_0 to p_3 belong to low load utilization profiles since $\rho_1 < 1$. On the other hand, profiles p_4 to p_7 are high load utilization profiles because $\rho_1 \geq 1$.

Fig. 5(a) and 5(b) show the results for the low and high load utilization profiles, respectively. For all the methods, as the load utilization increases, the violation ratio increases.

The static method achieves a good violation ratio between 0.02 and 9.65 percent with the low load utilization profiles. However, it suffers from a much worse QoS in the high load utilization scenarios where its violation ratio is between 47.61 and 54.49 percent. This is an inevitable result because the static method does not handle system dynamicity.

AREA achieves a violation ratio between 0.3 and 17.95 percent by its QoS awareness. The performance for the low load utilization profiles is comparable with that of the static method. With the high load utilization profiles, it outperforms the static method.

RACER shows the best QoS awareness. The RACER-vc100[7] trained with the violation unit cost of 100 achieves a violation ratio between 0.01 and 4.42 percent. It outperforms AREA for all the task arrival profiles by reducing the violation ratio by 92 percent on average. It keeps the violation ratio below 0.09 percent in low utilization. Even under high load situations, the achieved violation ratio is much better than that of AREA. We also measured RACER's performance without considering the violation cost by setting the violation unit cost to 0, i.e., RACER-vc0. Interestingly, the QoS awareness of RACER-vc0 is still better than AREA. It reduces the violation ratio by 54.7 percent compared to AREA on average for all task arrival profiles.

The above difference in response time QoS between RACER and AREA comes from their design principles. While AREA considers the *average* response time QoS, RACER adopts the token bucket mechanism for task admission control to bound *per-task* response time. In addition to the token bucket, RACER reinforces its QoS awareness from rewards, including the penalty of QoS violation in training. From this, one can recognize RACER-vc0 as an agent that uses only the token bucket mechanism for QoS awareness without reinforcement from violation penalizing rewards.

D. WEIGHTED AVERAGE TASK RESPONSE SLOWDOWN

Fig. 6 shows the results for the weighted average response slowdown. As defined in Section III-B, slowdown is a normalized response time that is the ratio of the response time of a task to its QoS bound, i.e., d_{unk}/D_k. For example, if the response time equals the QoS bound of the task, the slowdown becomes 1.

In low load utilization, as shown in Fig. 6(a), all the methods show a desirable slowdown performance. All of them achieve a weighted average response slowdown between 0.65 and 0.75. It implies that the tasks are completed in 65 to 75 percent of their per-task response time-bound on average. In high load utilization, as shown in Fig. 6(b), the slowdown performance of the static method becomes much worse. It suffers from enormous slowdowns with task arrival profiles p_5, p_6, and p_7. On the other hand, RACER and AREA successfully maintain the slowdown between 0.71 to 0.86. It suggests that their control mechanisms work well to maintain a desirable level of slowdown performance as they try to solve the problem $P1$.

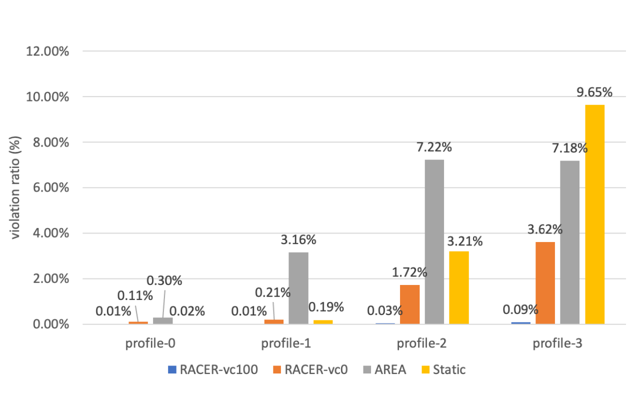
As opposed to the case of response time QoS, RACER-vc100 achieves a little worse slowdown performance than that of RACER-vc0. It is because the RACER-vc100 agent considers the response time QoS more than the RACER-vc0 agent, which compromises the slowdown performance. An encouraging result is that the RACER-vc100 agent reduces the QoS violation ratio by 82.3 percent while increasing the weighted average response slowdown just by 6.3 percent compared to those of RACER-vc0 on average for all the eight task arrival profiles.

E. CONTROL EFFICIENCY

To evaluate the control efficiency, we measured the control decision time of RACER, AREA, and the static method during the simulation.[8] We define the control decision time as

[7]We denote a RACER agent by suffixing the violation unit cost used in training.

[8]We used RACER-vc100 as a representative.


(a)

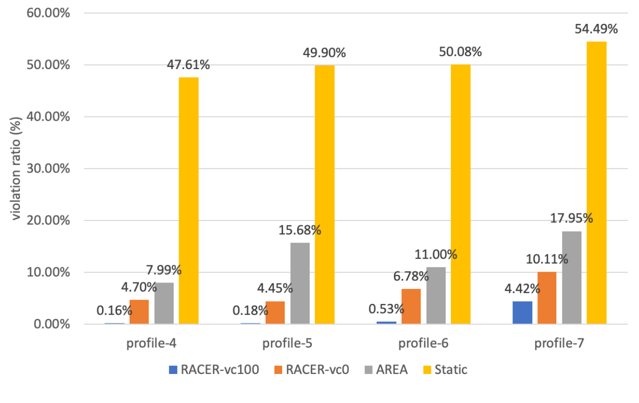
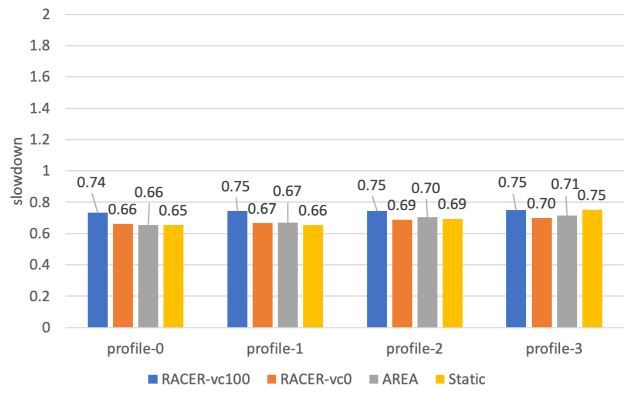

(b)

FIGURE 5. QoS violation ratio: (a) in low load utilization (b) in high load utilization.


(a)

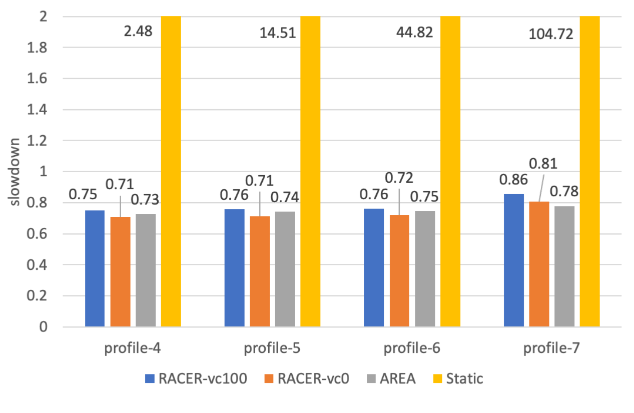

(b)

FIGURE 6. Weighted average slowdown: (a) in low load utilization (b) in high load utilization.

the duration between the time of receiving inputs and the time of producing control outputs. Note that it is not a simulation time, but a wall clock time. We implemented the control decision modules of RACER, AREA, and the static method as Python script and conducted the simulation on a 2.2GHz, 128GB RAM Intel Xeon processor. Additionally, RACER uses an NVIDIA Titan XP GPU since its decision is made with a neural network.

Fig. 7 shows that the control decisions of the static method, RACER, and AREA require 5.5, 66.6, and 3497.3 milliseconds (ms), respectively, on average for all the task profiles. We believe that the result is highly related to the decision complexity of the problems each method solves. The static method has the decision complexity of $O(1)$ because it controls decision variables by not solving $P1$, but just by deriving from pre-specified rules. In the case of RACER, the decision complexity of the problem is $O(|N| \times |K|)$, which affects the size of the neural network. Then the control decision time in RACER is dominated by a single forward computation on the neural network. On the other hand, the problem of AREA has the decision complexity of $O(|U| \times |N| \times |K|)$ (see Section VII-B). Considering that $|U| = 300$, $|N| = 5$, and $|K| = 5$ in the simulation, it is obvious that AREA has the largest decision complexity.

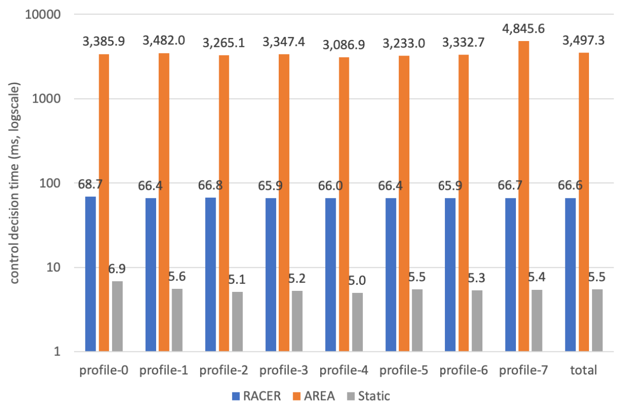


FIGURE 7. Average control decision time.

The control decision modules of all the methods may be improved by code optimization techniques such as complied execution and loop parallelism. Nonetheless, we believe that the relative performance remains a similar trend to the above results.

F. IMPACT OF VIOLATION UNIT COST

To investigate the effect of the violation unit cost, we evaluated the performance of RACER with various violation unit

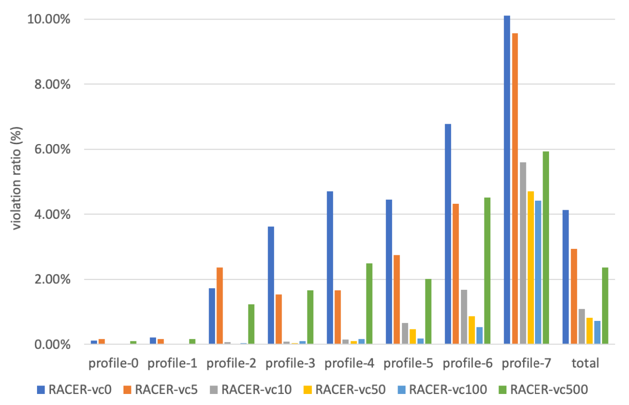


FIGURE 8. **Violation ratios according to violation unit cost.**

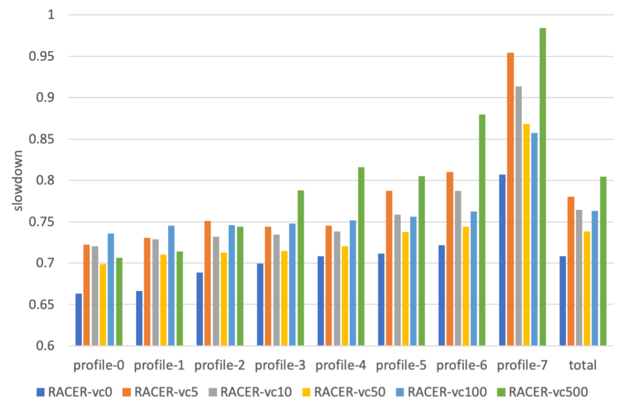


FIGURE 9. **Slowdowns according to violation unit cost.**

costs: $\xi_k = 0, 5, 10, 50, 100$, and 500. Fig. 8 presents the QoS violation ratios of the six RACER agents with varying task arrival profiles. As ξ_k increases, the violation ratio decreases until $\xi_k = 100$. However, RACER-vc500 shows a higher violation ratio than those of RACER-vc10, vc50, and vc100. From the observations in the training phase, we found that when $\xi_k = 500$, the agent had not been trained enough to converge the mean episodic reward.

We present the weighted average slowdowns of every RACER agent in Fig. 9. RACER-vc0 shows the best slowdown performance. It is in agreement as RACER-vc0 does not compromise the optimality for QoS consideration. However, an interesting result is that the slowdown does not monotonically increase as ξ_k increases. On average, RACER-vc50 shows the second-best performance. Except for the case of RACER-vc0, it is hard to find any tendency about the correlation between the slowdown and ξ_k. Nevertheless, an encouraging thing here is that RACER succeeds in retaining the slowdown below 1 for all the task arrival profiles, even high load utilization cases, for every ξ_k.

G. EVALUATION SUMMARY

We summarize the simulation results in terms of three questions as follows:

1) *Does RACER support the per-task response time QoS?* The RACER-vc100 agent reduces the QoS violation ratio by 92 percent, compared to AREA, a state-of-the-art QoS-aware workload distribution scheme (see Fig. 5).
2) *How much does RACER sacrifice the optimality for per-task QoS?* The RACER-vc100 agent succeeded in reducing the QoS violation ratio by 82.3 percent while sacrificing the weighted average response slowdown by a little increase (6.3 percent), compared to RACER-vc0 (see Fig. 6).
3) *Is RACER fast and adaptive enough for real-time control environments with temporal dynamics?* Although implemented in Python script, RACER achieves a control decision time of a few tens of milliseconds, which is 52 times faster than that of AREA. Further speed-up may be possible by code optimization techniques like complied execution (see Fig. 7).

In our evaluation, we used a somewhat controlled simulation setup to clearly recognize the effects of workload dynamicity on system performance. However, we believe that this is enough to evaluate the potential capability of our approach. Evaluating the performance of our approach in more realistic and application-specific environments may be interesting future work.

VIII. CONCLUSION

In this paper, we presented RACER, a novel workload distribution approach that supports per-task response time QoS in hierarchical edge clouds. RACER is an elaborate combination of token bucket mechanism and reinforcement learning for dynamic control to balance the trade-off between the task response slowdown and response time QoS under non-stationary task arrival scenarios that imply uncertainty of system behavior. The simulation results show that RACER achieves a competitive performance in terms of task response slowdown, response time QoS, and control efficiency.

APPENDIX

PROOF OF LEMMA 1

Proof: Let $(\mathrm{x}^*(t), \mathrm{y}^*(t), \mathrm{z}^*(t))$ be an optimal solution of problem $P2$ and $(\hat{\mathrm{x}}(t), \hat{\mathrm{y}}(t), \hat{\mathrm{z}}(t))$ be an optimal solution of $P2$ with an additional constraint (26). Let $(\mathrm{x}'(t), \mathrm{y}^*(t), \mathrm{z}^*(t))$ be the solution of $P2$ satisfying (26) such that its aggregate type k task traffic to node n is equal to that of the optimal solution $(\mathrm{x}^*(t), \mathrm{y}^*(t), \mathrm{z}^*(t))$ of $P2$. That is, we have

$$\sum_u x'_{unk}(t) \cdot \lambda_{uk}(t) = \sum_u x^*_{unk}(t) \cdot \lambda_{uk}(t) \tag{35}$$

for all n and k.

Let $f(\mathrm{x}'(t), \mathrm{y}'(t), \mathrm{z}'(t))$ and $f(\mathrm{x}'(t), \mathrm{y}^*(t), \mathrm{z}^*(t))$ be the objective function value of $(\mathrm{x}'(t), \mathrm{y}'(t), \mathrm{z}'(t))$ and $(\mathrm{x}'(t), \mathrm{y}^*(t), \mathrm{z}^*(t))$, respectively. Then, we obtain

$$f(\hat{\mathrm{x}}(t), \hat{\mathrm{y}}(t), \hat{\mathrm{z}}(t)) - f^* \le \sum_{n,k} \frac{\Delta D^{net}_{nk}(t)}{D_k} \tag{36}$$

which is derived from (37) to (43), as shown at the top of the next page.

$$f(\hat{x}(t), \hat{y}(t), \hat{z}(t)) - f^* \leq f(x'(t), y^*(t), z^*(t)) - f^* \tag{37}$$
$$= \sum_{u,n,k} \frac{x'_{unk}(t) \cdot \lambda_{uk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{d'_{unk}(t)}{D_k} - \sum_{u,n,k} \frac{x^*_{unk}(t) \cdot \lambda_{uk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{d^*_{unk}(t)}{D_k} \tag{38}$$
$$\leq \sum_{u,n,k} \frac{x'_{unk}(t) \cdot \lambda_{uk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{\max_{\breve{u}} d'_{\breve{u}nk}(t)}{D_k} - \sum_{u,n,k} \frac{x^*_{unk}(t) \cdot \lambda_{uk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{d^*_{unk}(t)}{D_k} \tag{39}$$
$$= \sum_{u,n,k} \frac{x^*_{unk}(t) \cdot \lambda_{uk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{\max_{\breve{u}} d'_{\breve{u}nk}(t)}{D_k} - \sum_{u,n,k} \frac{x^*_{unk}(t) \cdot \lambda_{uk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{d^*_{unk}(t)}{D_k} \tag{40}$$
$$\leq \sum_{u,n,k} \frac{x^*_{unk}(t) \cdot \lambda_{uk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{\Delta D^{net}_{nk}(t)}{D_k} \tag{41}$$
$$= \sum_{n,k} \frac{\lambda^*_{nk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \cdot \frac{\Delta D^{net}_{nk}(t)}{D_k} \tag{42}$$
$$\leq \sum_{n,k} \frac{\Delta D^{net}_{nk}(t)}{D_k}. \tag{43}$$

Inequality (37) holds because $(\hat{x}(t), \hat{y}(t), \hat{z}(t))$ is an optimal solution of $P2$ with (26). Inequality (39) holds because $d'_{unk}(t) \leq \max_{\breve{u}} d'_{\breve{u}nk}(t)$. Equality (40) holds due to (35). Inequality (41) holds because

$$\max_{\breve{u}} d'_{\breve{u}nk}(t) - d^*_{unk}(t)$$
$$= \max_{\breve{u}}(d^{com}_{nk}(t) + d^{net}_{\breve{u}nk}(t)) - (d^{com}_{nk}(t) + d^{net}_{unk}(t)) \tag{44}$$
$$= \max_{\breve{u}} d^{net}_{\breve{u}nk}(t) - d^{net}_{unk}(t) \tag{45}$$
$$\leq \Delta D^{net}_{nk}(t). \tag{46}$$

Inequality (43) holds because $\frac{\lambda^*_{nk}(t)}{\sum_{\breve{u},\breve{k}} \lambda_{\breve{u}\breve{k}}(t)} \leq 1$ for all n and k. □

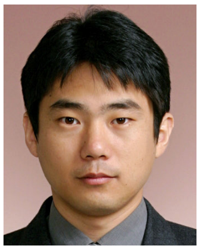
ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions, which helped them improve the quality of the paper.

REFERENCES

[1] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—A key technology towards 5G," *ETSI White Paper*, vol. 11, no. 11, pp. 1–16, 2015.
[2] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018.
[3] E. Ahmed, A. Ahmed, I. Yaqoob, J. Shuja, A. Gani, M. Imran, and M. Shoaib, "Bringing computation closer toward the user network: Is edge computing the solution?" *IEEE Commun. Mag.*, vol. 55, no. 11, pp. 138–144, Nov. 2017.
[4] (2020). *AWS IoT Greengrass*. [Online]. Available: https://aws.amazon.com/greengrass/
[5] (2020). *Microsoft Azure IoT Edge*. [Online]. Available: https://docs.microsoft.com/azure/iot-edge/
[6] (2020). *Amazon Echo*. [Online]. Available: https://www.youtube.com/channel/UCz2-0uvBJt-AwoiFSyMb_yQ
[7] (2020). *Google Home*. [Online]. Available: https://store.google.com/product/google_home
[8] (2020). *Amazon Snowball Edge*. [Online]. Available: https://aws.amazon.com/snowball-edge
[9] (2020). *Microsoft Azure Data Box Edge*. [Online]. Available: https://azure.microsoft.com/en-us/services/databox/
[10] (2020). *AWS Lambda*. [Online]. Available: https://aws.amazon.com/lambda/
[11] (2020). *Microsoft Azure Functions*. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-deploy-function
[12] (2020). *OpenWhisk*. [Online]. Available: https://openwhisk.apache.org/
[13] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.
[14] L. Yang, J. Cao, G. Liang, and X. Han, "Cost aware service placement and load dispatching in mobile cloud systems," *IEEE Trans. Comput.*, vol. 65, no. 5, pp. 1440–1452, May 2016.
[15] H. Tan, Z. Han, X. Li, and F. C. M. Lau, "Online job dispatching and scheduling in edge-clouds," in *Proc. IEEE INFOCOM*, May 2017, pp. 1–9.
[16] C. Liu, K. Li, and K. Li, "A game approach to multi-servers load balancing with load-dependent server availability consideration," *IEEE Trans. Cloud Comput.*, early access, Jan. 15, 2018, doi: 10.1109/TCC.2018.2790404.
[17] Q. Liu, S. Huang, J. Opadere, and T. Han, "An edge network orchestrator for mobile augmented reality," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 756–764.
[18] R. Yu, G. Xue, and X. Zhang, "Application provisioning in FOG computing-enabled Internet-of-Things: A network perspective," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 783–791.
[19] Q. Fan and N. Ansari, "Application aware workload allocation for edge computing-based IoT," *IEEE Internet Things J.*, vol. 5, no. 3, pp. 2146–2153, Jun. 2018.
[20] Q. Fan and N. Ansari, "Workload allocation in hierarchical cloudlet networks," *IEEE Commun. Lett.*, vol. 22, no. 4, pp. 820–823, Apr. 2018.
[21] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, and G. Agha, "DROPLET: Distributed operator placement for IoT applications spanning edge and cloud resources," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 1–8.
[22] T. He, H. Khamfroush, S. Wang, T. La Porta, and S. Stein, "It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources," in *Proc. IEEE ICDCS*, Jul. 2018, pp. 365–375.
[23] J. Gedeon, M. Stein, L. Wang, and M. Muehlhaeuser, "On scalable in-network operator placement for edge computing," in *Proc. 27th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2018, pp. 1–9.
[24] Y.-D. Lin, Y.-C. Lai, J.-X. Huang, and H.-T. Chien, "Three-tier capacity and traffic allocation for core, edges, and devices for mobile edge computing," *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 3, pp. 923–933, Sep. 2018.
[25] Y. Jiang and D. H. K. Tsang, "Delay-aware task offloading in shared fog networks," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4945–4956, Dec. 2018.

[26] L. Lin, P. Li, X. Liao, H. Jin, and Y. Zhang, "Echo: An edge-centric code offloading system with quality of service guarantee," *IEEE Access*, vol. 7, pp. 5905–5917, 2019.
[27] H. Liao, Z. Zhou, S. Mumtaz, and J. Rodriguez, "Robust task offloading for IoT fog computing under information asymmetry and information uncertainty," in *Proc. IEEE ICC*, May 2019, pp. 1–6.
[28] H. Liao, Z. Zhou, X. Zhao, B. Ai, and S. Mumtaz, "Task offloading for vehicular fog computing under information uncertainty: A matching-learning approach," in *Proc. 15th Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, Jun. 2019, pp. 2001–2006.
[29] X. Niu, S. Shao, C. Xin, J. Zhou, S. Guo, X. Chen, and F. Qi, "Workload allocation mechanism for minimum service delay in edge computing-based power Internet of Things," *IEEE Access*, vol. 7, pp. 83771–83784, May 2019.
[30] Z. Xu, W. Liang, M. Jia, M. Huang, and G. Mao, "Task offloading with network function requirements in a mobile edge-cloud network," *IEEE Trans. Mobile Comput.*, vol. 18, no. 11, pp. 2672–2685, Nov. 2019.
[31] A. Leon-Garcia and I. Widjaja, *Communication Networks: Fundamental Concepts and Key Architectures*, 2nd ed. New York, NY, USA: McGraw-Hill, 2006.
[32] M. Guo, Q. Guan, and W. Ke, "Optimal scheduling of VMs in queueing cloud computing systems with a heterogeneous workload," *IEEE Access*, vol. 6, pp. 15178–15191, 2018.
[33] H. Liao, Z. Zhou, X. Zhao, L. Zhang, S. Mumtaz, A. Jolfaei, S. H. Ahmed, and A. K. Bashir, "Learning-based context-aware resource allocation for edge-computing-empowered industrial IoT," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4260–4277, May 2020.
[34] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.
[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," Jul. 2017, *arXiv:1707.06347*. [Online]. Available: http://arxiv.org/abs/1707.06347
[36] (2019). *Learning Dexterity*. [Online]. Available: https://openai.com/blog/learning-dexterity/
[37] (2019). *OpenAI Five*. [Online]. Available: https://openai.com/five/
[38] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, Jun. 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html
[39] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *Proc. ICLR*, 2016. [Online]. Available: http://arxiv.org/abs/1509.02971
[40] (2019). *OpenAI Spinning Up: Proximal Policy Optimization*. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html#
[41] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," Dec. 2017, *arXiv:1712.09381*. [Online]. Available: http://arxiv.org/abs/1712.09381
[42] H. Jeon, C. Cho, S. Shin, and S. Yoon, "A CloudSim-extension for simulating distributed functions-as-a-service," in *Proc. 20th Int. Conf. Parallel Distrib. Comput., Appl. Technol. (PDCAT)*, Dec. 2019, pp. 386–391.
[43] (2019). *DFaaSCloud: Distributed Function-as-a-Service Simulator*. [Online]. Available: https://github.com/etri/DFaaSCloud
[44] (2019). *CloudSim: A Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services*. [Online]. Available: http://www.cloudbus.org/cloudsim/



CHUNGLAE CHO (Member, IEEE) received the B.S. and M.S. degrees in computer science from Pusan National University, South Korea, in 1994 and 1996, respectively, and the Ph.D. degree in computer engineering from the University of Florida, Gainesville, in 2011. He is currently with the Electronics and Telecommunications Research Institute (ETRI), Daejeon, South Korea. His research interests include computer networks, software-defined networking, cloud computing, edge computing, and machine learning.

SEUNGJAE SHIN (Member, IEEE) received the B.S. degree in electrical and computer engineering from Chung-Nam National University, Daejeon, South Korea, in 2007, and the M.S. and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 2009 and 2017, respectively. He is currently with the Electronics and Telecommunications Research Institute (ETRI), Daejeon. His research interests include computer networks, cloud computing, and reinforcement learning.

HONGSEOK JEON received the B.S. degree in industrial engineering from Sungkyunkwan University, Seoul, South Korea, in 2002, and the M.S. degree in engineering from Information and Communications University (ICU), Daejeon, South Korea, in 2004. He is currently with the Electronics and Telecommunications Research Institute (ETRI), Daejeon. His primary research interests include computer networks, network functions virtualization, and reinforcement learning. He has also made several contributions to IETF and IEEE standardization.

SEUNGHYUN YOON received the B.S., M.S., and Ph.D. degrees in industrial engineering from Sungkyunkwan University, Seoul, South Korea, in 1991, 1993 and 1997, respectively. He is currently with the Electronics and Telecommunications Research Institute (ETRI), Daejeon, South Korea. He is interested in the computer networks, cloud computing, and optimization in network and computer.