Collaborative Mobile Edge Computing Through UPF Selection

Yuanzhe Li(✉), Ao Zhou, Xiao Ma, and Shangguang Wang

Beijing University of Posts and Telecommunications, Beijing, China
{buptlyz,aozhou,maxiao18,sgwang}@bupt.edu.cn

Abstract. The distributed deployment and the relatively limited resource of one edge node make it quite challenging to effectively manage resources at the edge. Inappropriate scheduling may result in a quality of service deterioration and brings significant cost. In this paper, we propose a per-user level management mechanism for joint scheduling of user requests and container resources at the edge and study how to minimize average cost as well as satisfy delay constraints. The cost model of the system consists of operating cost, switching cost and delay violation cost. The key idea is to deploy a deep reinforcement learning-based scheduler in the core network to conduct joint network and computation management. To evaluate the performance, we build a test bed namely MiniEdgeCore that contains a full user plane protocol stack and deploy a real-time video inference application on it. A real-world dataset is used as the workload sequence to conduct experiments. The results show that the proposed method can reduce average costs effectively.

Keywords: Mobile edge computing · 5G · Request dispatching · Container management

1 Introduction

With the rapid development of the 5G network and Mobile Edge Computing (MEC), the traditional end-cloud computation is evolving into the end-edge-cloud mechanism. Thanks to this change, end devices are released from heavy computation tasks by offloading tasks to edge nodes. As a result, end devices could be more light and portable, providing more powerful services. This creates several emerging big markets for the next generation of killer applications [33,34]. For example, mobile AR and VR are supposed to create a market of USD 766 billion by 2025, with compound annual revenue growth of 73.3% from 2018 to 2025 [28].

Guaranteeing the Quality of Service (QoS) of computation-intensive and delay-sensitive services in MEC requires dynamic provisioning of computational resources. That is, when the request number per slot increases, the edge node should increase the number of container instances to avoid the long processing delay resulting from requests queuing on the server side. When the request number per slot decreases, the edge node needs to appropriately reduce the number of

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2022
Published by Springer Nature Switzerland AG 2022. All Rights Reserved
H. Gao et al. (Eds.): CollaborateCom 2022, LNICST 461, pp. 345–362, 2022.
https://doi.org/10.1007/978-3-031-24386-8_19

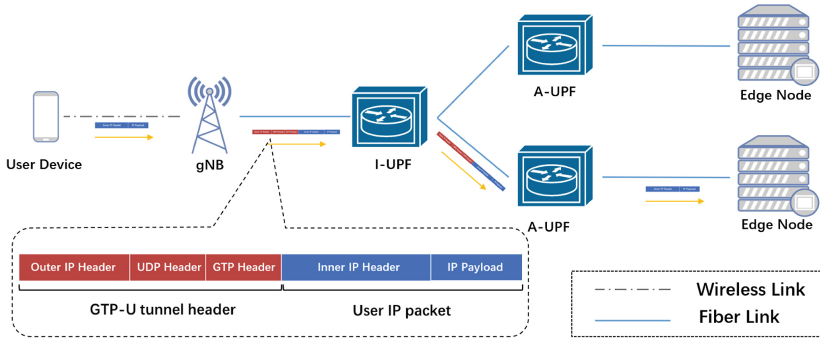


Fig. 1. GTP-U Tunnel

running containers, thereby saving computing resources and improving resource utilization. This is an intuitive method but still far from satisfactory. It has two disadvantages: 1) Starting or terminating instances brings time costs. This process lasts several seconds, during which the service is not available. 2) As each instance serves more than one user, such an adjustment cannot achieve per-user level management. Therefore, changing the number of running container instances is only suitable for coarse granularity and low-frequency management.

In traditional cloud data centers, the per-user level management is achieved by a load balance server. However, owing to the resource limitation of edge nodes, it is very likely that only a few instances are kept for one service. As a result, load balancing within one edge node is not enough. When container instances within one edge node cannot handle all the requests, dispatching user requests among different edge nodes is a practical way [17]. However, how to implement such dispatching is quite challenging. First, traditional load balance in cloud data centers relies on a centralized load balance server, which is not applicable for request dispatching among different edge nodes. Second, using the DNS mechanism in edge systems is not as efficient as it is in cloud computing [16]. As the DNS records are updated periodically according to the Time to Live (TTL) configured by the administrator, the DNS resolving results may remain the same during this time. This means a per-user level dispatching is not applicable and the resolving result may not match the state of the highly dynamic environment. Finally, and most importantly, user request dispatching relies on the selection of User Plane Function (UPF). User packets in the communication network (no matter 4G or 5G) are transferred through GPRS Tunneling Protocol User plane (GTP-U) tunnel, where original IP packets are encapsulated into GTP-U protocol data units (shown in Fig. 1). The original IP address is not used in packet routing within the communication network between gNB and Anchor UPF (A-UPF). Therefore, per-user level management in MEC should rely on the session management and traffic steering provided by the communication network architecture to realize the request dispatching.

Given the above facts, it is clear that achieving a per-user level matching between user requirements and computation resources is essential for MEC to realize its full potential. Thus, we pose a critical question: how to properly dispatch user requests and manage containers at edge nodes in 5G MEC to meet the latency constraints and minimize the total cost. The total cost comes from container operation cost, latency violation cost and container mismatching cost.

Currently, it has become a consensus that resource provision and request dispatching are highly interdependent and should be considered jointly as two levels of QoS guarantee methods [15,24,30]. That is, adjusting computation resource provision to cope with workload change in a long period and dispatching requests of each user to different edge nodes to deal with instantaneous changes. However, prior arts either provide a pure theoretic method based on specific delay models [10,39] or only consider the scheduling of computing resources at the service deployment level [11,24,31,32,36]. In addition, the aforementioned works neglect the key role of the 5G core network in request dispatching. To pave the way for deploying emerging applications such as cloud AR/VR at the edge, this paper focuses on how to achieve joint management of user request dispatching and container instance number scheduling in 5G architecture. The target is to minimize the total operating cost while satisfying the delay constraint.

In this paper, we study how to reduce total operation cost as well as satisfy the low delay requirement. The main contributions are listed as follows:

1) We propose a per-user level user request dispatching and container instance scheduling mechanism. This mechanism takes full consideration of 5G core network architecture and achieves the joint management of network and computation resources.
2) A deep reinforcement learning-based scheduling algorithm is proposed to jointly manage user request dispatching and container instance scheduling. The container switching delay during starting or terminating a container instance is considered so that the proposed method is more practical in real-world scenarios.
3) We build a test bed, namely MiniEdgeCore, to emulate the complicated request dispatching scenarios in MEC. MiniEdgeCore provides a full-stack 5G core network user plane that can dispatch user requests by setting up different GTP-U tunnels. It also has a Docker-based edge node system that works under the guidance of the core network.
4) Extensive experiments are conducted on MiniEdgeCore with a real-world dataset as the workload sequence input. The experiment results show that the proposed method can minimize the average total cost as well as guarantee a low access delay.

2 System Model

2.1 Scenario

As is shown in Fig. 2, we consider a mobile network consisting of base stations, edge nodes and UPFs. The topology of the network is an undirected graph $G =$

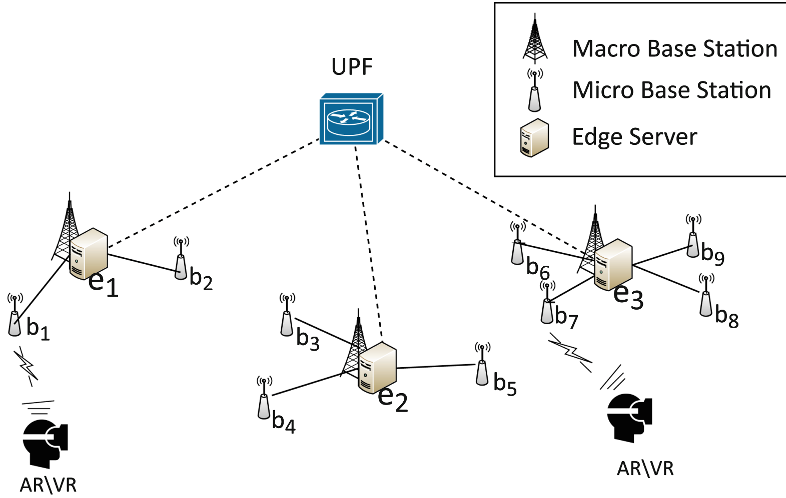


Fig. 2. System topology.

$(B \bigcup E \bigcup U, L)$. Let $B = \{b_1, b_2, \cdots, b_{n_b}\}$ denote the set of base stations, where $n_b(n_b = 1, 2, 3, \dots)$ is the total base stations number. Let $E = \{e_1, e_2, \cdots, e_{n_e}\}$ denote the set of edge nodes, where $n_e(n_e = 1, 2, 3, \dots)$ represents the edge node number. Let $U = \{u_1, u_2, \cdots, u_{n_u}\}$ denote the set of UPFs, where $n_u(n_u = 1, 2, 3, \dots)$ represents the UPF number. L denotes the physical communication links among base stations, edge nodes and UPFs. The operation of the system is described in a set of time slots T, indexed by $t = 1, 2, 3, \dots, \mathcal{T}$ with a slot length τ.

At time slot t, there are several user requests. Let R^t denote the set of all requests that exist at time slot t. n_r^t represents the number of user requests in time slot t, i.e., $n_r^t = |R^t|$. r_x^t represents the request of user x at time slot t. If user x is requesting a service at t, $r_x^t \in R^t$. Otherwise, $r_x^t \notin R^t$. Let $\mathcal{B}(r_x^t)$ denote the base station that user i links to. $\mathcal{E}(r_x^t)$ denotes the edge node that is serving r_x^t. $\mathcal{U}(r_x^t)$ denotes the UPFs that is in the connection of r_x^t. As a result, the offloading information of a request is determined by $(\mathcal{B}(r_x^t), \mathcal{E}(r_x^t), \mathcal{U}(r_x^t))$. Each User Equipment (UE) has two types of network connections. The first one is the physical link with the base station. It is determined by user location. Let $\alpha_{xi}^t \in \{0, 1\}$ denote such a physical connection. If $\mathcal{B}(r_x^t) = b_i$, $\alpha_{xi}^t = 1$. Otherwise, $\alpha_{xi}^t = 0$. The other one is the logical link between UE and edge node, which connects the request producer and request consumer. Such a relationship is denoted by $\beta_{xj}^t \in \{0, 1\}$. If $\mathcal{E}(r_x^t) = e_j$, then $\beta_{xj}^t = 1$. Otherwise, $\beta_{xj}^t = 0$.

Generally, a certain number of container instances need to be started to serve these requests in each edge node. Current container platforms support CPU core number limitation for one container [20,23]. In this paper, it is supposed that each container instance works with only one CPU core assigned to it. As a result, the computing resources provided by each edge node at time slot t can

be represented by the number of running container instances. Let $m_i^t(m_i^t = 0, 1, 2, \cdots M_i)$ denote the number of running container instances of edge node e_i at time slot t, where M_i represents the maximum number of container instances that can be started at edge node e_i.

In ideal situations, all user requests are dispatched to the nearest edge node to achieve the lowest access delay and best quality of service. However, the nearest edge node is not always the best choice in real world [6,17,35]. Considering computing resources are limited at edge nodes, if the nearest edge node gets overloaded, the total delay may exceed the upper limit as a result of long computation delay. However, the spatial and temporal distribution of user requests is uneven in cities. On the one hand, user number in different regions differs a lot, which makes the request number of different edge nodes at the same time slot vary a lot. On the other hand, user request number changes dynamically at one edge node because of user movement. Therefore, user requests may be routed to other edge nodes to prevent the nearest edge node from getting overloaded.

2.2 Delay

In mobile edge computing, access delay is one of the most important metrics. In most cases, access delay comes from transmission, processing, and backhaul [27]. Transmission delay refers to the delay brought by wireless communication between UE and base stations. Processing delay refers to the time needed for a processor to finish the task and the time consumed by tasks waiting in the queue. Backhaul delay refers to the total time for a packet to wait in the queues of network equipment when it traverses the distance between edge servers and base stations. As request dispatching does not affect the transmission delay of wireless communication, the service delay in this paper involves backhaul delay and processing delay.

Let φ_i and φ_j denote network devices located at the two ends of one link. $d(\varphi_i, \varphi_j)$ denotes the delay between these to devices. Then, the backhaul delay of a user request consists of two parts, i.e., the delay between the base station and UPF as well as the delay between UPF and the edge node. The backhaul delay can be defined as follows:

$$D_{\mathrm{N}}(r_x^t) = d(\mathcal{B}(r_x^t), \mathcal{U}(r_x^t)) + d(\mathcal{U}(r_x^t), \mathcal{E}(r_x^t)), \tag{1}$$

where $D_{\mathrm{N}}(r_x^t)$ denotes the backhaul delay of service request r_x^t.

The computation delay of request r_x^t is defined as the total time it takes from the request's arriving at the edge node to the edge node's sending back the result.

$$D_{\mathrm{C}}(r_x^t) = T_{\mathrm{e}}(r_x^t) - T_{\mathrm{s}}(r_x^t), \tag{2}$$

where $T_{\mathrm{s}}(r_x^t)$ denotes the time edge node receives the request and start to process it. $T_{\mathrm{e}}(r_x^t)$ denotes the time that the process ends.

The total delay consists of communication delay and computation delay. It is given as follows

$$D(r_x^t) = D_{\mathrm{N}}(r_x^t) + D_{\mathrm{C}}(r_x^t). \tag{3}$$

In real-world scenarios, average delay is not a good indicator of the system because it can be easily affected by extreme values. Besides, pursuing extremely short delays cannot improve the quality of service if delays have already been lower than a specific threshold. As a result, statistical delay guarantee is a more practical way to evaluate the quality of service [19]. We adopt delay ratio as the key metrics to evaluate the network state. To get the delay ratio, the delay of each user request is sampled N_{d} times in a time slot. Supposing that user x has n_d^t samples satisfying the delay constraint in time slot t, then the delay ratio is defined as follows:

$$\mathcal{R}_x^t = \frac{n_d^t}{N_d}. \tag{4}$$

2.3 System Cost

For a time period between t_a and t_b ($t_a, t_b \in T$), the total system cost contains three parts, i.e., operating cost, switching cost and delay violation cost. Operating cost refers to the rental cost that tenants are charged according to the number of containers they are using in each time slot. There are many charging systems. Tenants can pay by the year, by month, pay as you use, etc. To simplify the problem, we adopt a charging system of pay by time slot.

$$C_{\mathrm{r}}(t_a, t_b) = p_{\mathrm{run}} \sum_{t=t_a}^{t_b} \sum_{i=1}^{n_e} m_i^t, \tag{5}$$

where $C_{\mathrm{r}}(t_a, t_b)$ denotes the total operating cost, p_{run} represents the cost of one container running for one time slot, m_i^t is the total container number of edge node e_i running in time slot t.

Switching cost comes from operations of starting or terminating a container in an edge node. Changing the number of running containers according to the variation of total workload can prevent operating cost waste but will introduce extra system overhead [4]. Besides, the switching operation cannot take effect immediately because of the startup and termination time of a container. In order to prevent frequent switching operations, switching cost is introduced, which is defined as follows:

$$C_{\mathrm{s}}(t_a, t_b) = p_{\mathrm{switch}} \sum_{t=t_a}^{t_b} \sum_{i=1}^{n_e} |m_i^t - m_i^{t-1}|, \tag{6}$$

where $C_{\mathrm{s}}(t_a, t_b)$ denotes the total switching cost and p_{switch} is the price for one single switching operation.

Delay violation cost results from the potential access delay violation. The violation may come from the long network delay stemming from improper request

dispatching. It could also come from the extra computation delay caused by improper resource scheduling. Access delay violation will lead to unacceptable quality of service, and, as a result, the edge operator has to compensate users. Thus, a delay violation cost is introduced.

$$C_{\mathrm{d}}(t_a, t_b) = p_{\mathrm{delay}} \sum_{t=t_a}^{t_b} \sum_{r_x^t \in R^t} \eta_x^t, \tag{7}$$

$$\eta_x^t = \begin{cases} 1, \mathcal{R}_x^t < \mathcal{R}_{\min} \\ 0, \mathit{otherwise} \end{cases}, \tag{8}$$

where $C_{\mathrm{d}}(t_a, t_b)$ denotes the total delay violation cost, p_{delay} is the punishment for one delay violation, $\mathcal{R}_x^t$ is the delay ratio of user x at time slot t, $\mathcal{R}_{\min}$ is the lowest acceptable delay ratio according to service level agreement.

Therefore, the total cost of the system during the period from t_a to t_b can be denoted as follows:

$$C_{\mathrm{total}}(t_a, t_b) = C_{\mathrm{r}}(t_a, t_b) + C_{\mathrm{s}}(t_a, t_b) + C_{\mathrm{d}}(t_a, t_b). \tag{9}$$

Then the average cost per request per slot is defined as follows:

$$C_{\mathrm{req}} = \frac{C_{\mathrm{total}}(t_a, t_b)}{\sum_{t=t_a}^{t_b} n_r^t}. \tag{10}$$

2.4 Problem Formulation

In mobile edge computing, edge service provider changes the number of running containers at different edge nodes to dynamically adjust the computing resource provision to the computing demands of users. Besides, request dispatching is used to route user requests to proper edge nodes in a fine-grained manner to prevent frequent container switching operation and achieve a quick response. It is quite challenging to achieve a joint management of request dispatching and container scheduling with a low cost. This paper studies how to minimize the average cost per request per slot from the perspective of edge service providers.

$$\mathit{Minimize} \;\; C_{\mathrm{req}}, \tag{11}$$

$$s.t. \quad \sum_{i=1}^{n_b} \alpha_{xi}^t = 1, \quad \forall 0 \le x \le n_r^t, \tag{12}$$

$$\sum_{j=1}^{n_e} \beta_{xj}^t = 1, \quad \forall 0 \le x \le n_r^t, \tag{13}$$

$$m_i^t \le M_i, \tag{14}$$

where Eq. 12 represents that each UE connects to only one base station. Equation 13 guarantees that each user request is responded by only one edge node. Equation 14 represents that the running container number of one edge node cannot exceed the upper limit.

3 Algorithm Design

User requests dispatching and container scheduling in mobile edge computing are typical sequential decision-making problems, which are often modeled as Markov decision processes [5,29]. This is a challenging problem because it involves joint management of network and computing resources. In this paper, Advantage Actor Critic (A2C) algorithm is adopted to solve this problem. First, we map the problem into Markov decision process. A Markov decision process $M = (S, A, P, R)$ consists of a finite set of state S, a finite set of actions A, a state transition probability P and a reward function R. In this problem, the effect of actions on the system is deterministic, so the key points of this Markov decision process are the state, action and reward function, which are defined as follows:

At the beginning of each time slot t, the state of the system is constructed as the input of the A2C agent, which consists of: 1) The maximum number of container instances that each edge node can launch, i.e., M_i; 2) The number of running container instances of each edge node in the current time slot, i.e., $M_i - m_i^t$; 3) The dispatching relationship of user requests that are launched in previous time slots and haven't been terminated; 4) The information on new user requests that will be launched in this time slot.

Every action $a_t \in A$ in the action set consists of two parts: the information for container instance management and the information for user request dispatching. Supposing that there are n_e edge nodes in the system and at most n_q new user requests in a time slot. Then a_t will be an array with n_e+n_q bits. The first n_e bits correspond to the container instance operations of each edge node. This paper assumes that in each time slot, each edge node can only have three container operations: adding a container, reducing a container, or keeping the number of containers unchanged. The value of each bit could be 1,0 or -1, respectively. The last n_q bits correspond to the request dispatching decision and its value refers to the ID of the target edge node. Since the actual number of user requests in each time slot satisfies $n_r^t \leq n_q$, the first $n_e + n_r^t$ bits of the entire action array are valid, and the rest bits are filled with 0 by default. The output of A2C's policy network is a continuous action array a_t'. The value of each bit of a_t' is limited between -1.5 and 1.5. Then a_t' is discretized to a_t through an action discreteness algorithm shown in Algorithm 1.

The reward function is defined as $w^t - c^t$, where $w^t = w_0(n_r^t - \sum_{r_x^t \in R^t} \eta_x^t)$ is the reward brought by requests whose delay ratio meets the requirements. w_0 is the reward for a single request. $c^t = C_{\text{total}}(t-1, t)$ is the cost of the system in one time slot.

In 5G architecture, the orchestrator in MEC, acting as an Application Function (AF), can interact with the core network to provide application influence on traffic routing [1,8]. This mechanism makes it applicable to achieve joint management of 5G network and edge nodes. Therefore, this paper uses a centralized control method and deploys the A2C agent in the core network to schedule user requests and edge-side containers jointly.

Algorithm 1. Action Discreteness Algorithm

Input: a'_t
Output: a_t
1: Clamp each bit of a'_t to [-1.5, 1.5]
2: **for** $i = 0; i < n_e; i++$ **do**
3:     **if** $a'_t[i] > 0.5$ and $2m_i^t - \sum_{r_x^t \in R^t} \beta_{xi}^t < 2$ **then**
4:         $a_t[i] = 1$
5:     **else if** $2m_i^t - \sum_{r_x^t \in R^t} \beta_{xi}^t \geq 4$ **then**
6:         $a_t[i] = -1$
7:     **else**
8:         $a_t[i] = 0$
9:     **end if**
10: **end for**
11: **for** $i = n_e; i < n_e + n_q; i++$ **do**
12:     **if** $i < n_e + n_r^t$ **then**
13:         $a_t[i] = \lfloor n_e(a'_t[i] + 1.5)/3 \rfloor + 1$
14:     **else**
15:         $a_t[i] = 0$
16:     **end if**
17: **end for**
18: **return** a_t.

4 Experiment

4.1 5G MEC Experiment Platform

We build a test bed called MiniEdgeCore. The test bed consists of two parts. The first part is a 5G core network system consisting of a simplified control plane and a full-stack user plane. The control plane only implements the necessary functionalities related to traffic steering such as session management and UPF selection. The user plane implements UPF with a full GTP-U protocol stack [2, 3]. GTP-U is widely used in protocol which creates a UDP-based tunnel between gNB and Protocol data unit Session Anchor (PSA) to enable interconnection between UE and external packet data networks such as the Internet and local data network. Generally, there are two roles for UPFs. It can either serve as an Uplink Classifier (UL CL) or as PSA. For the convenience of expression, in the following, we use Intermediate UPF (I-UPF) to refer to the UPF working as ULCL,, and A-UPF to refer to the UPF that serve as PSA.

The second part of the system is edge nodes composed of several servers deployed with Docker [20] system. In each edge node, an AF is implemented based on Docker Python SDK[1], which is in charge of interacting with the core network and starting or terminating a container instance.

To achieve a flexible network architecture, we run network functions in Mininet [21] hosts. Mininet is a Linux-based system that consists of virtual networks, switches and applications running in a real kernel. By leveraging Mininet,

[1] https://docs.docker.com/engine/api/.

we can create different net topologies and test realistic network streams in one network node. Besides, the wireless network between UE and gNB is replaced by Ethernet because we were not concerned with the communication status of the wireless terminal. This can help to save the cost of software-defined radio devices and make the system capable of emulating large-number user scenarios.

One network node together with several edge servers composes one cluster that emulates one service area in the city. Multiple clusters linking with each other can emulate complicated request dispatching scenarios in MEC. For one user request, the dispatching is achieved by setting up a GTP-U tunnel. The I-UPF in this tunnel will route request data packets to the target A-UPF, which is connected with an edge node. If the target A-UPF locates in the same cluster, user requests are routed to the local edge node, otherwise, user requests will be processed by edge nodes located in another service area.

4.2 Experiment Setup

The experiment is conducted in the MiniEdgeCore system mentioned above. The application used in the experiment is a live stream real-time action inference. A UE node in MiniEdgeCore pushes a pre-recorded hand motion video to the edge server using an RTSP [25] stream set up by FFMPEG [12]. Then, a processing service based on Mediapipe [14] detects hand locations frame by frame and return hand-knuckle coordinates to the user. In the process of video streaming, the system records the time stamp when a frame is sent, received, and processed so that the backhaul delay and processing delay can be calculated. The system clocks of the servers are synchronized using Network Time Protocol [22].

This experiment is conducted by emulating the user request scenarios in the whole city. We use Edge Computing Dataset[2] as input to provide workload sequence. The dataset records the mobile Internet access log of users in Beijing. Each record in the dataset provides information including phone number (encrypted to protect privacy), location area code and cell identification code of the connected base station, access point name, international mobile device identification code (the first six digits), the start time and end time of network access, upstream and downstream traffic, and gateway information. One example of user request workload sequences is shown in Fig. 3. In this paper, all base stations are divided into six service areas. Each service area is equipped with an edge server. According to the location information of the user's access to the base station, the user's access requests can be mapped to each service area as the input workload of the area.

In this experiment, MiniEdgeCore deploys six service areas, corresponding to the six service areas of the dataset respectively. Each service area has four base stations, and each base station has five UEs. Each UE is in a dormant state by default. MiniEdgeCore activates a different number of dormant UEs in each time slot according to the workload sequence recorded in the dataset. UEs start to upstream video after it is activated. UE requests in one area are offloaded to

[2] https://github.com/BuptMecMigration/Edge-Computing-Dataset.

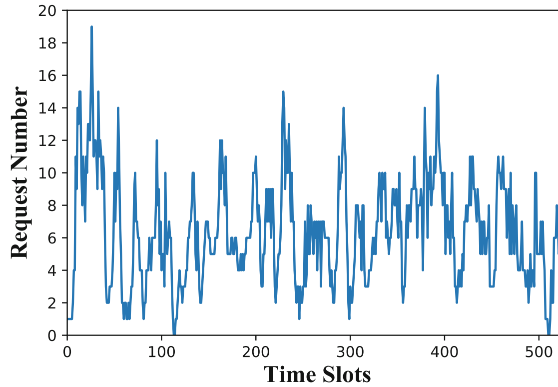


Fig. 3. User request number per time slot.

A-UPFs in different areas by I-UPF. A-UPF is connected to an edge server by binding the physical network card through the Open vSwitch[3]. The edge server is a Lenovo ThinkCenter M910t with a 3.40 GHz Intel Core i7-6700 CPU and a 16G DDR4 memory. Intel Core i7-6700 has four cores. One CPU core is used to run AF which is in charge of communicating with the core network. The remaining three CPU cores are used to run container instances. Each container instance is mapped to one CPU core.

4.3 Benchmark Algorithms

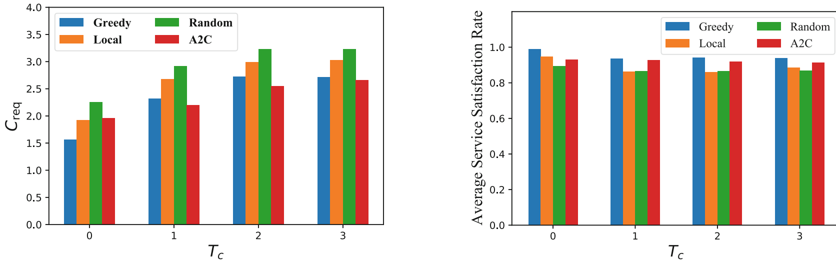
In this section, C_{req} and Average Service Satisfaction Rate are selected as the main metrics. Average Service Satisfaction Rate is defined as the average proportion of users whose request delay ratio meets the requirements. The benchmark algorithms are listed as follows:

1. **Local** All user requests are processed in the edge node within the service area. The edge node will start a new container when the processing delay reaches the upper limit. It terminates a container if instances work in an idle state.
2. **Random** User's service requests are randomly dispatched to different edge nodes. Each edge node adjusts the number of container instances according to its workload. When the upper limit is reached, a new container is started, and the container is terminated when the container is idle.
3. **Greedy** User requests in each time slot are dispatched to the edge node with the most sufficient computing resources in the time slot. Each edge server opens a new container when this server's processing delay reaches the upper limit and closes one container instance when it is idle.

[3] https://www.openvswitch.org/.

4.4 Experiment Results

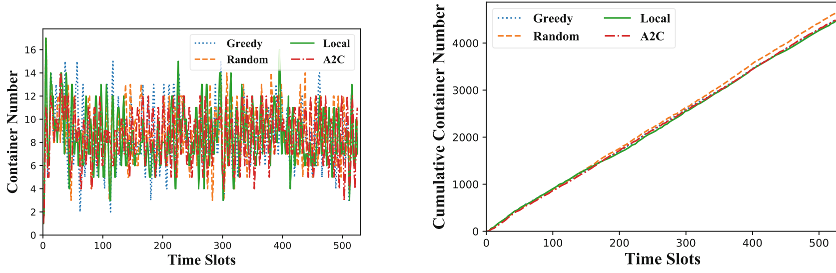
In this experiment, the time slot length is set to 5 s. The maximum service satisfaction delay is 220 ms. The minimum delay ratio $\mathcal{R}_{\min}$ is 0.9. p_{run} is set to 1. p_{switch} is set to 3, and p_{delay} is set to 3. As for parameters used in A2C training, the cross-entropy coefficient is 0.01. The reward discount is 0.9, and the learning rate is 0.001.



(a) Average cost per request per slot. (b) Average service satisfaction rate.

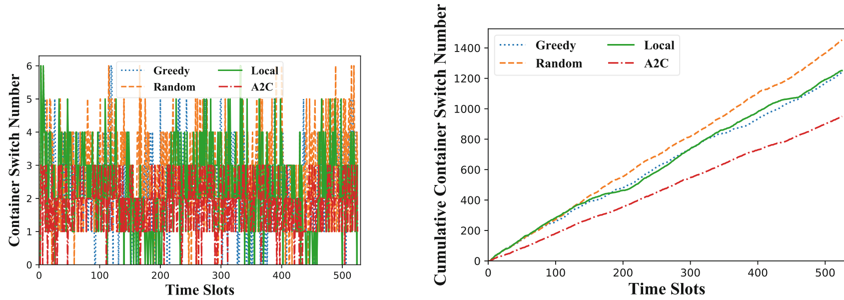
Fig. 4. Algorithm performance with different container switching delay.

Figure 4 shows the average cost per request per slot C_{req}(Fig. 4a) and the average service satisfaction rate (Fig. 4b) of each algorithm in the experiment. The horizontal axis T_c in the figure represents container switching delay, that is, the adjustment operation on the number of containers needs to pass T_c time slots to take effect. As shown in the figure, when $T_c = 0$ (container operation takes effect immediately), Greedy achieves the highest service satisfaction rate (98.96%) and the lowest C_{req} (1.56). Local achieves a delay satisfaction rate of 94.68% with a C_{req} of 1.92, and A2C achieves a delay satisfaction rate of 93.03% with a C_{req} of 1.96. The performance of Random is the poorest, and its delay satisfaction rate is less than 90%. However, the performance of A2C begins to stand out when $T_c \neq 0$. As T_c increases from 1 time slot to 3 time slots, A2C maintains the lowest C_{req}, which is on average 4.51% less than Greedy, 14.96% less than Local, and 21.17% less than Random. In terms of average service satisfaction rate, when container switching delay is non-zero, only Greedy and A2C can maintain a service satisfaction rate of above 90%. Both Local and Random are lower than 90%. In the ideal case where the container operation takes effect immediately, Greedy is the optimal strategy. Because the system can immediately adjust the number of running container instances according to the change of workload and dispatch the user's service requests to the edge node with the most abundant computing resources. This can avoid the processing timeout caused by the overload of a single container. However, in real-world scenarios, it takes a certain amount of time for the container instance to start or terminate [4]. In this situation, although Greedy can ensure a high service satisfaction rate, its container operations bring extra costs due to the existence of container



(a) Real-time running container number. (b) Cumulative running container number.

Fig. 5. Running container number.

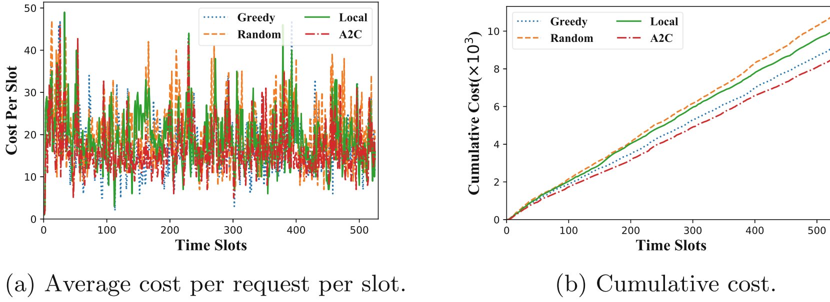


(a) Real-time container switching times. (b) Cumulative container switching times

Fig. 6. Container switching times

switching delays. A2C has a forward-looking decision-making process through the training of the agent. As a result, the cost of container scheduling is lower.

In order to further explain the reason why A2C achieves the lowest C_{req} in the presence of container switching delay, a complete epoch is analyzed in the case of $T_c = 2$. As shown in Fig. 5, during the whole experiment, the number of running container instances and the number of user requests (see Fig. 3) have a similar fluctuation pattern. Among all algorithms, the real-time running container number of A2C (shown in Fig. 5a) has a smaller fluctuation range than other algorithms. Its cumulative number of containers (shown in Fig. 5b) is basically the same as that of Greedy and Local, and less than that of Random. However, the performance of each algorithm on container switching times is significantly different. As shown in Fig. 6a, the real-time container switching times of A2C are significantly lower than that of other algorithms. The gap of cumulative container switching times (see Fig. 6b) is much more distinct. The cumulative container switching times of A2C are 23.38% less than that of Greedy, 24.06% less than that of Local, and 34.62% less than that of Random. Since A2C has smaller container switching times, the C_{req} of A2C is relatively lower (Fig. 7a),



(a) Average cost per request per slot.

(b) Cumulative cost.

Fig. 7. Total cost.

and the cumulative cost of A2C is 6.52% less than that of Greedy, which is 14.74% less than Local and 21. 07% less than Random.

5 Related Work

5.1 Joint Scheduling Methods

In mobile edge computing, dynamic resource scheduling in service offloading generally includes two levels, i.e., coarse-grained edge node computing resource scheduling and fine-grained request dispatching. The former mainly adjusts the provision of computing resources in the edge node dynamically to meet the user requirement in the service area. The latter mainly selects target edge node and chooses transmission path to guarantee that the user side delay meets the QoS requirement.

Existing works mainly focus on the joint optimization of request dispatching at network side and dynamic service placement in the edge node. Ting He et al. [31] study the joint service placement and request scheduling in mobile edge computing with consideration of both sharable resources (storage) and non-shareable resources (communication, computation). They develop a constant-factor approximation algorithm and evaluate performance of the algorithm using simulation. Vajiheh Farhadi et al. [11] try to maximize the expected requests served by edge nodes per slot by optimizing service placement and request scheduling. They leverage trace-driven simulation to evaluate the performance of their algorithm. Konstantinos Poularakis et al. [24] jointly consider storage, communication, computation resource constraints in service placement and request scheduling. Bo Yin et al. [37] introduce the concept of age of information in the study of scheduling in mobile edge computing. They leverage age of information to quantify the information freshness and propose two computationally tractable scheduling policies to minimize age of information. Yiwen Han et al. [15] study distributed request scheduling and dynamic service deployment in edge nodes. They proposed multi-agent reinforcement learning-based algorithm to improve system throughput while reducing system scheduling overhead.

These works are effective but still can be improved. Most of these works study the problem in two time scales, i.e., service deployment at a larger scale and request dispatching at a smaller scale. However, the computing resource management is still coarse-grained. Edge nodes not only have to decide what service to deploy, but also have to schedule how many container instances to run in each time slot. In addition, these works fail to take session management and traffic steering mechanism of 5G core network architecture into consideration.

5.2 Test Beds

Yoohwa Kang et al. [18] implement a test bed system to evaluate their multipath transmission control protocol based on multi-access traffic steering solution. In their test bed, UE get access to the data network through WLAN and 5G gNB. The 5G gNB uses software-defined radio to emulate an LTE gNB. Mingyuan Zang et al. [38] leverage the open source project OpenAirInterface to build an in-lab emulation test bed. They use this test bed to verify mobile edge cache in different network scenarios. Bhaskar Prasad Rimal et al. [26] design a two-level edge computing scheme in a fiber-wireless access network. In order to evaluate the performance of their proposed solution, they implement an experimental test bed with edge applications in optical fiber backhaul networks. Mona Ghassemian et al. share their experience in building a 5G test bed platform in [13]. Their 5G-VINNI project deploys 5G-NR radio as well as virtualized EPC outside to test performance in the 3.6 GHz (first implementation phase) and 5G mmWave (second implementation phase). Multiple use cases are tested including cloud-based gaming, connected care for assisted living, remote robotic control, etc., covering gaming, health and industry. However, such a test bed system is based on Samsung network equipment, which is expensive and may not be suitable for in-lab experiments. Mohammad Kazem Chamran et al. [7] study the independent decision-making of distributed nodes in 5G scenarios by implementing a distributed test bed. Different from traditional centralized decision-making systems, the proposed system consists of Universal Software Radio Peripheral nodes embedded with Raspberry Pi3 B+. That is, test bed nodes can communicate with each other as well as make decisions independently. Ali Esmaeily et al. [9] also leverage OpenAirInterface to implement a test bed for end-to-end network slicing.

The aforementioned systems take advantage of open source projects to implement network emulation. These test beds emulate various network scenarios and can get data close to the real scene. However, limited by the coverage area of software-designed radio, the access limit of total equipment number and the high price of software designed radio devices, these systems lack scalability. Evaluating large-scale research, such as service deployment, mobile edge offloading and user request scheduling, on these systems are difficult.

6 Conclusion

In order to guarantee the QoS of delay-sensitive applications, MEC relies heavily on the dynamic joint management of user request dispatching and edge-side container scheduling. In this paper, we first establish a cost model for service offloading scenarios. Then, we map the joint scheduling problem into a Markov decision process and proposed a reinforcement learning-based algorithm to solve it. Next, instead of conducting simulations leveraging mathematical delay models, we build a test bed called MiniEdgeCore, which provides a full-stack 5G core network user plane and a Docker-based edge node system. MiniEdgeCore implements user request dispatching by setting up GTP-U tunnels to different UPFs. Besides, it uses the interaction process between the core network and the edge nodes to control the starting and terminating of container instances. Finally, experiments are conducted on MiniEdgeCore. A real-time video inference application is deployed on MiniEdgeCore and a real-world dataset is used as workload sequence input. The experiment results show that the proposed method can reduce at least 4.51% of cost.

Acknowledgments. This work was supported in part by the National Key R&D Program of China (No. 2020YFB1805502) and NSFC (U21B2016, 62032003 and 61922017).

References

1. 3GPP: TS 23.501, system architecture for the 5G System. In: Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP). https://www.3gpp.org/ftp/Specs/archive/23_series/23.501/
2. 3GPP: TS 29.060, GPRS Tunneling Protocol (GTP) across the Gn and Gp interface. In: Technical Specification (TS) 29.060, 3rd Generation Partnership Project (3GPP). https://www.3gpp.org/ftp/Specs/archive/29_series/29.060/
3. 3GPP: TS 29.281, General Packet Radio System (GPRS) Tunneling Protocol User Plane (GTPv1-U). In: Technical Specification (TS) 29.281, 3rd Generation Partnership Project (3GPP). https://www.3gpp.org/ftp/Specs/archive/29_series/29.281/
4. Ahmed, A., Mohan, A., Cooperman, G., Pierre, G.: Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart. In: Proceedings of IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), pp. 55–62 (2020)
5. Bäuerle, N., Rieder, U.: Markov decision processes. Jahresber. Deutsch. Math.-Verein. **112**(4), 217–243 (2010)
6. Ceselli, A., Premoli, M., Secci, S.: Mobile edge cloud network design optimization. IEEE/ACM Trans. Netw. **25**(3), 1818–1831 (2017)
7. Chamran, M.K., Yau, K.L.A., Noor, R.M.D., Wong, R.: A distributed testbed for 5G scenarios: an experimental study. Sensors **20**(1), 18 (2020)
8. Contreras, L.M., et al.: MEC in 5G networks. Tech. rep., European Telecommunications techreports Institute
9. Esmaeily, A., Kralevska, K., Gligoroski, D.: A cloud-based SDN/NFV testbed for end-to-end network slicing in 4G/5G. In: Proceedings of IEEE Conference on Network Softwarization (NetSoft), pp. 29–35 (2020)

10. Fang, L., Liu, T., Zhu, Y., Yang, Y.: Task offloading and dispatching for MEC with selfish mobile devices and access points. In: Proceedings of IEEE Global Communications Conference (GLOBECOM), pp. 1–6 (2020)
11. Farhadi, V., et al.: Service placement and request scheduling for data-intensive applications in edge clouds. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), pp. 1279–1287 (2019)
12. FFmpeg: FFmpeg (2022). https://ffmpeg.org/
13. Ghassemian, M., Muschamp, P., Warren, D.: Experience building a 5G testbed platform. arXiv:2008.01628 (2020)
14. Google: Mediapipe (2021). https://google.github.io/mediapipe/
15. Han, Y., Shen, S., Wang, X., Wang, S., Leung, V.C.: Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), pp. 1–10 (2021)
16. Hsu, K.J., Choncholas, J., Bhardwaj, K., Gavrilovska, A.: DNS does not suffice for MEC-CDN. In: Proceedings of ACM Workshop on Hot Topics in Networks (HotNets), pp. 212–218. Association for Computing Machinery, New York, NY, USA (2020)
17. Jia, M., Cao, J., Liang, W.: Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks. IEEE Trans. Cloud Comput. **5**(4), 725–737 (2017)
18. Kang, Y., Kim, C., An, D., Yoon, H.: Multipath transmission control protocol-based multi-access traffic steering solution for 5G multimedia-centric network: design and testbed system implementation. Int. J. Distrib. Sensor Netw. **16**(2), 155014772090975 (2020)
19. Li, Q., Wang, S., Yang, F.: QoS driven task offloading with statistical guarantee in mobile edge computing. IEEE Trans. Mob. Comput. **21**(1), 278–290 (2020)
20. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
21. Mininet: Mininet (2022). http://mininet.org/
22. Network Time Foundation: NTP: the network time protocol (2014). http://www.ntp.org/
23. Podman: Podman (2022). https://podman.io/
24. Poularakis, K., Llorca, J., Tulino, A.M., Taylor, I., Tassiulas, L.: Joint service placement and request routing in multi-cell mobile edge computing networks. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), pp. 10–18 (2019)
25. Rao, A., Lanphier, R., Schulzrinne, H.: Real Time Streaming Protocol (RTSP). Tech. Rep. 2326 (1998). https://www.rfc-editor.org/info/rfc2326
26. Rimal, B.P., Maier, M., Satyanarayanan, M.: Experimental testbed for edge computing in fiber-wireless broadband access networks. IEEE Commun. Mag. **56**(8), 160–167 (2018)
27. Rodrigues, T.G., Suto, K., Nishiyama, H., Kato, N., Temma, K.: Cloudlets activation scheme for scalable mobile edge computing with transmission power control and virtual machine migration. IEEE Trans. Comput. **67**(9), 1287–1300 (2018)
28. Siriwardhana, Y., Porambage, P., Liyanage, M., Ylianttila, M.: A survey on mobile augmented reality with 5g mobile edge computing: architectures, applications, and technical aspects. IEEE Commun. Surv. Tutorials **23**(2), 1160–1192 (2021)
29. Sutton, R.S., Barto, A.G.: Reinforcement learning: an introduction. In: Adaptive Computation and Machine Learning Series, The MIT Press, Cambridge, Massachusetts, second edition edn (2018)

30. Tan, H., Han, Z., Li, X.Y., Lau, F.C.: Online job dispatching and scheduling in edge-clouds. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), pp. 1–9. IEEE, Atlanta, GA, USA (2017)
31. He, T., Khamfroush, H., Wang, S., La Porta, T., Stein, S.: It's hard to share: joint service placement and request scheduling in edge clouds with sharable and non-sharable resources. In: Proceedings of International Conference on Distributed Computing Systems (ICDCS), pp. 365–375. IEEE, Vienna (2018)
32. Tong, L., Li, Y., Gao, W.: A hierarchical edge cloud architecture for mobile computing. In: Proceedings of IEEE International Conference on Computer Communications (INFOCOM), pp. 1–9. IEEE, San Francisco, CA, USA (2016)
33. Xu, M., Qian, F., Zhu, M., Huang, F., Pushp, S., Liu, X.: DeepWear: adaptive local offloading for on-wearable deep learning. IEEE Trans. Mob. Comput. **19**(2), 314–330 (2020)
34. Xu, M., Xu, T., Liu, Y., Lin, F.X.: Video analytics with zero-streaming cameras. In: Proceedings of USENIX Annual Technical Conference (ATC), pp. 459–472. USENIX Association (2021)
35. Xu, Z., Liang, W., Xu, W., Jia, M., Guo, S.: Efficient Algorithms for Capacitated Cloudlet Placements. IEEE Trans. Parallel Distrib. Syst. **27**(10), 2866–2880 (2016)
36. Yang, L., Cao, J., Liang, G., Han, X.: Cost aware service placement and load dispatching in mobile cloud systems. IEEE Trans. Comput. **65**(5), 1440–1452 (2016)
37. Yin, B., et al.: Only those requested count: proactive scheduling policies for minimizing effective age-of-information. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), pp. 109–117 (2019)
38. Zang, M., Zhang, C., Yan, Y.: In-lab testbed for mobile edge caching with multiple users access. In: Proceedings of International Conference on Information and Communication Technology Convergence (ICTC), pp. 450–455 (2019)
39. Zeng, D., Gu, L., Guo, S., Cheng, Z., Yu, S.: Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system. IEEE Trans. Comput. **65**(12), 3702–3712 (2016)