Migration Modeling and Learning Algorithms for Containers in Fog Computing

Zhiqing Tang, Xiaojie Zhou, Fuming Zhang, Weijia Jia, *Senior Member, IEEE*, and Wei Zhao, *Fellow, IEEE*

Abstract—Fog Computing (FC) is a flexible architecture to support distributed domain-specific applications with cloud-like quality of service. However, current FC still lacks the mobility support mechanism when facing many mobile users with diversified application quality requirements. Such mobility support mechanism can be critical such as in the industrial internet where human, products, and devices are moveable. To fill in such gaps, in this paper we propose novel container migration algorithms and architecture to support mobility tasks with various application requirements. Our algorithms are realized from three aspects: 1) We consider mobile application tasks can be hosted in a container of a corresponding fog node that can be migrated, taking the communication delay and computational power consumption into consideration; 2) We further model such container migration strategy as multiple dimensional Markov Decision Process (MDP) spaces. To effectively reduce the large MDP spaces, efficient deep reinforcement learning algorithms are devised to achieve fast decision-making and 3) We implement the model and algorithms as a container migration prototype system and test its feasibility and performance. Extensive experiments show that our strategy outperforms the existing baseline approaches 2.9, 48.5 and 58.4 percent on average in terms of delay, power consumption, and migration cost, respectively.

Index Terms—Fog computing, user mobility, container migration, delay, power consumption, deep reinforcement learning

1 Introduction

With the development of cloud computing, cloud computing based mobile applications, such as real-time video streaming [1], real-time face recognition [2], have become popular recent years. Mobile users can offload some tasks to remote cloud data centers to gain larger computation capacity [3]. However, in many domain-specific applications, such as industrial application scenarios, cloud computing may not be able to respond mobile users on time, and the delay could be unacceptable. Besides, a centralized cloud is very hard to manage the various service requests from billions of mobile users. Moreover, cloud computing centralized data-centers are lacking in flexibility and unable to support mobility for mobile users [4].

To solve the problem, CISCO proposed fog computing (FC) architecture [5]. In FC, a significant number of light computation and storage infrastructures, called fog nodes, are deployed close to mobile users [6]. In this case, mobile application tasks can be offloaded to suitable nodes to shorten significant access delay. Besides, nodes are flexible, scalable, and capable of supporting the mobility for mobile users [7].

For better service provision and utilization of resource like CPU, memory, bandwidth, tasks are loaded in virtual machines (VM) and share the resource of the corresponding node with other VMs, instead of fully occupying all the resource [8]. Meanwhile, VMs can be migrated. On the one hand, the utilization of resource can be further improved and plenty of cost like power consumption can be saved [9]. Statistical results show that a lot of idle power is wasted when nodes run at a very low load [10]. Even at 10 percent CPU utilization, the consumed power exceeds 50 percent of peak power [11]. Therefore, if the VMs can be dynamically consolidated, plenty of nodes can be switched off and power consumption can be saved significantly [11]. On the other hand, the corresponding contents, which are located in VMs, could be migrated to the place close to mobile users. And the quality of service (QoS) is improved during the movement of the mobile users [4]. As a result, the quality of communication connection is upgraded, which further reduces the communication delay between mobile users and the nodes.

In the area of VM migration, recent studies mainly focus on assuming a prior distribution of resource requirements or learning resource requirements based on history utilization [12], [13], [14]. For assuming a prior distribution, the available resource is changed dynamically due to the unstable network and the mobility of the mobile users [15]. As a result, it is difficult to make an accurate assumption of prior distribution [13]. For learning resource requirements based on history utilization, some heuristic algorithms are proposed [13], [14]. However, most of these algorithms are suffered from the curse of dimensions when dealing with the

- *Z. Tang, X. Zhou, and F. Zhang are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {domain, szxjzhou, zhangfuming-alex}@sjtu.edu.cn.*
- *W. Jia is with the Centre of Data Science, University of Macau, SAR Macau 999078, China, and also with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: jia-wj@cs.sjtu.edu.cn.*
- *W. Zhao is with the American University of Sharjah, PO Box 26666, Sharjah 26666, UAE. E-mail: weizhao@umac.mo.*

Manuscript received 15 Dec. 2017; revised 29 Mar. 2018; accepted 8 Apr. 2018. Date of publication 16 Apr. 2018; date of current version 9 Oct. 2019. (Corresponding author: Weijia Jia.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSC.2018.2827070

large-scale migration problems, and such challenge results in long decision delay, which is unacceptable for many applications.

In FC scenarios, the management cost of traditional virtualization technology could be very high [16]. For this problem, a lightweight virtualization technology named Linux Container is a promising solution [17]. Compared with other heavyweight counterparts such as VM, containers have advantages in short time implementation, economical resource utilization, and low management cost [18]. Moreover, similar to VM, containers can also be migrated to improve service provision and resource utilization.

For the delay-sensitivity of mobile applications and the instability of actual network environment, the complexity and robustness of migration algorithms should be carefully designed to achieve fast migration decision-making [9]. Not only the delay and the power consumption, but also the migration cost and the mobility of mobile users need to be taken into consideration [4]. Meanwhile, the first-order transition probability of VMs' resource demands is also quasi-static for an extended period and non-uniformly distributed [13]. Moreover, the movement of mobile users is a sequential decision-making process and has memoryless property [19]. Therefore, it is suitable for us to adopt reinforcement learning algorithms in making migration decisions, which are algorithms with low computational complexity based on Markov chain model. In reinforcement learning algorithms, an agent takes charge of selecting the action according to the current state as well as specific strategy, keeping monitoring the system state, and updating the strategy [20]. Among varieties of reinforcement learning algorithms, Q-learning algorithm has the advantage in rapid decision-making [21]. Nevertheless, traditional Q-learning algorithm is infeasible to solve large-scale Markov decision process (MDP) problems for the enormous size of state set and action set. Thanks to the development of neural network technology, deep learning is capable of learning very complex functions and handling high-dimension data samples by using deep neural networks (DNNs) [22].

In this paper, we design and implement a container migration manager prototype system, where application tasks are laid in migratable containers and located in nodes. Besides, we consider the container migration in FC as a stochastic optimization problem. In this problem, the communication delay, the power consumption, and the movement of mobile users are taken as the transition parameters when we use Markov process to model the migration issues. Our algorithms are designed based on Q-learning and deep learning strategies, to handle large-scale container migration problem due to large MDP space for making the fast migration decisions. Such algorithms are implemented and tested in our prototype system. Our contributions can be summarized as follows:

(1) We consider each mobile application task can be hosted in a container of a corresponding node that can be migrated to another node in response to finding the best tradeoff between the total round-trip delay and the total power consumption. Differ from the existing algorithms, the task mobility requirements and the migration cost are taken into consideration, which are neglected in most of the previous migration algorithms for FC architectures.

(2) We further model such container migration strategy as multiple dimensional MDP spaces. To efficiently reduce the large MDP spaces in large-scale migration problem, robust and efficient deep reinforcement learning algorithms are devised to achieve fast migration decision-making.

(3) We implement the model and algorithms as a container migration manager prototype system and test its feasibility and performance. Real-world data experimental results show that our strategy outperforms the existing baseline approaches 2.9, 48.5 and 58.4 percent on average in terms of delay, power consumption, and the migration cost, respectively.

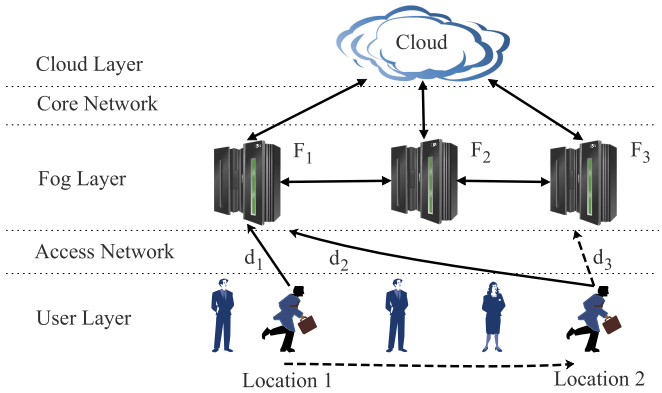


Fig. 1. System architecture.

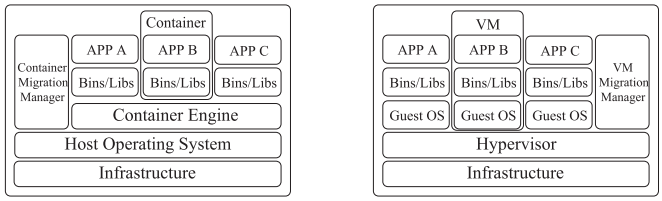
The remainder of this paper is organized as follows. In Section 2, we model the FC architecture, formulate and analyze the container migration problem. Then, the container migration algorithms are devised in Section 3. The experimental setup and experimental results of our algorithms are described in Section 4. We review the related work in Section 5, discuss some issues in Section 6, and conclude the paper in Section 7.

2 Modeling and Problem Definitions

We first present the system model in Section 2.1. Then, the container migration problem based on the FC architecture is formulated and analyzed in Sections 2.2 and 2.3, respectively.

2.1 System Model

We consider container based FC architecture with three layers: mobile user layer, fog layer, and cloud layer, as shown in Fig. 1. In FC, a container can serve multiple mobile users. Tasks are produced by mobile users, and parts of them are offloaded to corresponding containers via access networks [3]. Besides, each task is loaded in one corresponding container, which shares the physical resource with other containers in the same node [17]. Container migration manager in fog layer not only monitors the mobility behavior of mobile users, the delay, and the power consumption of nodes, as well as the resource requirement of containers, but also determines the container migration strategy [8]. For example, in Fig. 1, a mobile user moves from $Location_1$ to $Location_2$, resulting in long communication path between the mobile user and F_1. In this case, we assume the



(a) Container Migratioin (b) VM Migration

Fig. 2. Comparison between container and VM.

information about moving behaviors will be caught by the manager through GPS signal. According to pre-defined migration algorithms, the manager may migrate the corresponding container from F_1 to F_3, which shortens the distance to d_3, upgrades the quality of communication and reduces the round-trip delay. As illustrated in Fig. 2, compared with VM migration, only the necessary applications and the run-time libraries are migrated in container migration, except for the whole guest operating system (guest OS). The remote cloud data centers connect nodes via the core network. Instead of computing, cloud data centers mainly act for data replication and load balancing [8].

For better service provision and utilization of the resource, we design and implement container migration algorithms in this paper. Let $\boldsymbol{F} = \{F_1, F_2, ..., F_m\}, \boldsymbol{C} = \{C_1, C_2, ..., C_n\}, \boldsymbol{M} = \{M_1, M_2, ..., M_l\}$, which are the set of fog nodes with wired and wireless communication capability; the set of containers that reside in the nodes and run the user tasks; and the set of concurrent mobile users who may request connection to any of the nodes anytime, respectively.[1] Each node F_i has its location $F_i.l$ and resource capacity $F_i.c$. There exists four main kinds of resource: CPU, memory, storage, and bandwidth. Since scheduling of computation resource is the main factor, we can assume that there are sufficient memory, storage and network capacity for the containers in each node [13], [23]. We consider that different resource allocations to a mobile user will result in different delay and power consumption for the user's tasks. In this paper, containers are allocated to user tasks and migrated in order to reduce the power consumption while providing user tasks with satisfactory service regarding the delay. Container C_i's location at time t is denoted as $C_i.l(t) \in \{\boldsymbol{F}\}$. Different from node's identical resource capacity, the resource requirements and allocations of the containers are changed by time. Thus, let $C_i.r(t)$ and $C_i.a(t)$ denote C_i's resource requirement and allocation at t, respectively. Besides, unlike cloud computing, the moving of mobile users need to be taken into consideration. Consequently, we assume that $M_i.l(t)$ and $M_i.r(t)$ are the location and container request of mobile user M_i at t, respectively.

TABLE 1
Notations

$\boldsymbol{F}$	Fog node set
m	Number of nodes
F_i	i^{th} node ($i \in [1, m]$)
$F_i.l$	Location of F_i
$F_i.c$	Resource capacity of F_i
$\boldsymbol{C}$	Container set
n	Number of containers
C_i	i^{th} container ($i \in [1, n]$)
t	Real time
$C_i.l(t)$	Location of C_i at t ($C_i.l(t) \in \{\boldsymbol{F}\}$)
$C_i.r(t)$	Resource requirement of C_i at t
$C_i.a(t)$	Resource allocation of C_i at t
$C_i.m(t)$	List of mobile applications of C_i at t
$\boldsymbol{M}$	Mobile user set
l	Number of mobile user
M_i	i^{th} mobile user ($i \in [1, l]$)
$M_i.l(t)$	Location of M_i at t
$M_i.r(t)$	Container request of M_i at t
d_{total}	Total delay
d_{net}	Total network delay between mobile users and corresponding nodes
d_{comp}	Total computation delay of mobile application tasks
p_{total}	Total power consumption
$u_i(t)$	Resource utilization of F_i at t
m_{total}	Total migration cost
$\mathcal{C}$	Total cost
ω_1	Weight of delay in $\mathcal{C}$
ω_2	Weight of power consumption in $\mathcal{C}$
T_τ	τ^{th} time slice
τ	Serial number of T_τ
$\lvert T\rvert$	Length of T_τ
$\mathcal{X}(\tau)$	System state of delay during T_τ
S_τ	System state during T_τ
A_τ	Action set during T_τ
R_τ	Reward during T_τ
$Q(S_\tau, A_\tau)$	Q-value of S_τ and A_τ
Q_m	A best-action dictionary
α	Learning rate
γ	Discount parameter
Q_{m_i}	A key-value pair $\langle Q_{m_i}.s, (Q_{m_i}.a, Q_{m_i}.v)\rangle$
$Q_{m_i}.a$	Best action of state $Q_{m_i}.s$
$Q_{m_i}.v$	Corresponding Q-value of $Q_{m_i}.s$ and $Q_{m_i}.a$
th_{under}	Under-utilization threshold
th_{over}	Over-utilization threshold
$m_{total_{i,j}}(\tau')$	Estimated migration cost during $T\tau$
$\Delta\mathcal{R}_{i,j}(\tau)$	Migration revenue of migrating C_i to F_j during T_τ
$d_{total_{i,j}}(\tau')$	Estimated delay during $T\tau$
$p_{total_{i,j}}(\tau')$	Estimated power consumption during $T\tau$
$Fea_{i,j}(\tau)$	Feasibility of migrating C_i to F_j during $T\tau$
$L(\theta)$	Loss function of the DNN
D	Experience replay memory
θ	Weights of the DNN

2.2 Problem Formulation

The utilization of resource and the delay have the benefits of container migration. Plenty of nodes can be switched off, and lots of power consumption can be reduced through container consolidation [10]. However, if too many containers are merged into a node, the risk of resource over-requirement will be risen dramatically, leading to the performance degradation and longer delay for task processing [13]. Moreover, unsuitable migration strategy may extend the distance between mobile users and corresponding nodes, which results in weak network connection [4]. Therefore, there exists a tradeoff between the delay and the power consumption. With an objective to finding the best tradeoff, we define the delay and the power consumption in this section. In addition, non-negligible migration cost is incurred during the container migration, which is also defined in this section. Taking the delay, power consumption, and migration cost into account, our container migration problem can be formulated as shown below.

1. The major notations used in this paper are summarized in Table 1.

Delay. The total delay d_{total} consists of two parts: 1) the total network delay d_{net} between mobile users and corresponding nodes, 2) the total computation delay d_{comp} of the mobile application tasks. As for d_{net}, the delay in wireless access network is related to the path loss, which is defined as [24]:

$$d_{net} = \sum_{i=1}^{l} \int (k_{net} \log_{10} d_i(t) + b_{net}) dt, \tag{1}$$

where k_{net} and b_{net} are defined as:

$$k_{net} = 44.9 - 6.55 \log_{10}(h_b),$$
$$b_{net} = 46.3 + 33.9 \log_{10}(f) - 13.82 \log_{10}(h_b) - ah_m + c_m,$$

in which f is the signal frequency in MHz; $d_i(t)$ is the distance between M_i and corresponding node; h_b is the height of antenna of the node; c_m is set to 3 dB for normal circumstances and ah_m is defined as [25]:

$$ah_m = 3.20(\log_{10}(11.75 h_r))^2 - 4.97, f > 400\ MHz,$$

where h_r is height of the mobile users.

d_{comp} is influenced by the computation capacity of nodes and the degree of resource shortage. In this paper, we consider a heterogeneous FC architecture, where all the nodes have different computation capacity. The degree of resource shortage is modeled by SLA violation $SLAV$, which is defined as [14]:

$$SLAV = \frac{\sum_{i=1}^{n} \int (C_i.r(t) - C_i.a(t)) dt}{\sum_{i=1}^{n} \int (C_i.r(t)) dt}. \tag{2}$$

We further prove that the computation delay is proportional to SLA violation, as shown in Theorem 1.

Theorem 1. *Computation delay is proportional to SLA violation in FC architecture.*

Proof. If there exists no SLA violation, $C_i.a(t) = C_i.r(t)$, so $SLAV = 0$. For the general case, we usually pre-allocate resource to each container based on container's immediate CPU requirement. Assume that at time t, the immediate resource requirement of C_i is random variable X_i with mean μ_i. To avoid severe SLA violation, we would allocate more resource than X_i. Assume the resource allocation of C_i is $X_i + \varepsilon_i$. From Eq. (2) we can obtain:

$$SLAV = \sum_{i=1}^{n} \int \left(1 - \frac{C_i.a(t)}{C_i.r(t)}\right) dt.$$

Since resource requirement of each container at different time is independent with others [13], we can further obtain:

$$E[SLAV] = \sum_{i=1}^{n} E\left[\int \left(1 - \frac{C_i.a(t)}{C_i.r(t)}\right) dt\right]$$
$$= \sum_{i=1}^{n} \int \left(E\left[1 - \frac{C_i.a(t)}{C_i.r(t)}\right]\right) dt = \sum_{i=1}^{n} \int \left(1 - \frac{E[X_i + \varepsilon_i]}{E[X_i]}\right) dt$$
$$= \sum_{i=1}^{n} \int \left(1 - \frac{E[X_i] + E[\varepsilon_i]}{E[X_i]}\right) dt = -\sum_{i=1}^{n} \int \left(\frac{E[\varepsilon_i]}{\mu_i}\right) dt.$$

In addition, the expectation of delayed mobile application tasks is:

$$\sum_{i=1}^{n} \int (E[X_i - (X_i + \varepsilon_i)]) dt = -\sum_{i=1}^{n} \mu_i \int \left(\frac{E[\varepsilon_i]}{\mu_i}\right) dt.$$

As a result, this expectation is proportional to SLA violation. Since delayed time is proportional to the amount of delayed mobile application tasks, it is also proportional to SLA violation. □

Therefore, d_{comp} is defined as:

$$d_{comp} = k_{SLAV} \times SLAV$$
$$= k_{SLAV} \times \frac{\sum_{i=1}^{n} \int (C_i.r(t) - C_i.a(t)) dt}{\sum_{i=1}^{n} \int (C_i.r(t)) dt}, \tag{3}$$

where k_{SLAV} is the parameter controlling the weight of $SLAV$.

In short, based on Eqs. (1) and (3), d_{total} is obtained as:

$$d_{total} = d_{net} + k_{comp} \times d_{comp}, \tag{4}$$

where k_{comp} controls the weight of d_{comp}.

Power Consumption. The total power consumption p_{total} is equivalent to the addition of each node's power consumption. If F_i is switched off or set in sleep mode, its power consumption $p_{total_i} \approx 0$ [13]. Otherwise, its power consumption is proportional to resource utilization as indicated in [23]:

$$p_{total} = \int \left(\sum_{i=1}^{m} (p_{idle} + (p_{\max} - p_{idle}) \times u_i(t))\right) dt, \tag{5}$$

where $p_{idle}, p_{\max}$ represent the power consumption of 0% and 100% CPU utilization of F_i, respectively. Besides, $u_i(t)$ is the resource utilization of F_i at t, which is defined as:

$$u_i(t) = \min\left\{1, \frac{\sum_{1\{C_j.l(t)=F_i\}} C_j.a(t)}{F_i.c}\right\},$$

where $1\{\cdot\}$ is Iverson bracket, which is equivalent to 1 when the condition is satisfied. Otherwise, it is equivalent to 0.

Container Migration Cost. For C_i, when $C_i.l(t-1) \neq C_i.l(t)$, C_i is migrated from $C_i.l(t-1)$ to $C_i.l(t)$ at t. Otherwise, no migration happens to C_i at t. Migration can actually incur some non-negligible migration cost m_{total}, which is defined as:

$$m_{total} = \int \left(\sum_{i=1}^{n} \left(m_{mig_i} \times 1\{C_i.l(t) \neq C_i.l(t-1)\}\right)\right) dt, \tag{6}$$

where m_{mig_i} is the migration time of C_i, which includes the transmission delay. Compared with transmission delay, the migration decision-making delay can be ignored [26].

Problem Definition. Our algorithms aim to reduce the cost of power consumption and delay. Besides, the migration cost of containers is taken into account. In short, the total cost $\mathcal{C}$ is the weighted sum of d_{total}, p_{total}, and m_{total} defined in Eqs. (4), (5), and (6), respectively. The target is to find the best strategy which can minimize $\mathcal{C}$. Meanwhile, for each F_i, the resource constraint $0 \leq \sum_{1\{C_j.l(t)=F_i\}} \frac{C_j.a(t)}{F_i.c} \leq 1$ should be obeyed.

Therefore, the dynamic container migration problem of mobile applications in FC architecture is defined as follows:

Problem 1.

$$\begin{aligned} &\min \mathcal{C} = \omega_1 d_{total} + \omega_2 p_{total} + m_{total} \\ &s.t. \quad 0 \le \textstyle\sum_{1\{C_j.l(t)=F_i\}} \frac{C_j.a(t)}{F_i.c} \quad \le 1 \quad i = 1, 2, ..., m\,, \end{aligned} \tag{7}$$

where ω_1 and ω_2 are the parameters controlling the weight of d_{total} and p_{total}, respectively.

2.3 Problem Analysis

Problem 1 is an advanced bin-packing problem, which is NP-hard and can only be solved heuristically. However, most of existing heuristic algorithms are unstable in real network environment, and unable to handle large-scale problems which slowdown the decision-making. In this problem, d_{total}, p_{total}, and m_{total} can be represented as the addition of delay $d_{total}(\tau)$, power consumption $p_{total}(\tau)$, and migration cost $m_{total}(\tau)$ during each time slice $T_\tau(\tau = 0, ..., \kappa)$, which are shown as:

$$d_{total} = \sum_{\tau=0}^{\kappa} \left(\sum_{i=1}^{l} d_{net_i}(\tau) + k_{comp} \times \sum_{i=1}^{n} d_{comp_i}(\tau) \right), \tag{8}$$

$$p_{total} = \sum_{\tau=0}^{\kappa} \sum_{i=1}^{m} p_{total_i}(\tau), \tag{9}$$

$$m_{total} = \sum_{\tau=0}^{\kappa} \sum_{i=1}^{n} m_{total_i}(\tau), \tag{10}$$

where $d_{net_i}(\tau), d_{comp_i}(\tau)$ are the network delay and the computation delay of C_i, respectively. $p_{total_i}(\tau)$ is the power consumption of F_i, and $m_{total_i}(\tau)$ is the migration cost of C_i during T_τ, which are shown as:

$$d_{net_i}(\tau) = \int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|} (k_{net} \log_{10} d_i(t) + b_{net}) dt, \tag{11}$$

$$d_{comp_i}(\tau) = \frac{\int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|} (C_i.r(t) - C_i.a(t)) dt}{\int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|} (C_i.r(t)) dt}, \tag{12}$$

$$\begin{aligned} p_{total_i}(\tau) = \int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|} &(p_{idle} + (p_{\max} - p_{idle}) \\ &\times u_i(t)) dt, \end{aligned} \tag{13}$$

$$\begin{aligned} m_{total_i}(\tau) = \int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|} &\big(m_{mig_i} \\ &\times 1\{C_i.l(t) \neq C_i.l(t-1)\}\big) dt, \end{aligned} \tag{14}$$

where t_0 is the start time of the system, and $|T|$ is the length of time slice.

Assume that $\mathcal{C}(\tau)$ is the total cost until T_τ. From Eqs. (8)-(14), we can conclude that $\mathcal{C}(\tau)$ obeys first-order Markov process as:

$$\begin{aligned} \mathcal{C}(\tau) = 1\{\tau > 0\}\mathcal{C}(\tau - 1) + \omega_1 \Bigg(&\sum_{i=1}^{l} d_{net_i}(\tau) \\ &+ k_{comp} \sum_{i=1}^{n} d_{comp_i}(\tau) \Bigg) + \omega_2 \sum_{i=1}^{m} p_{total_i}(\tau) \\ &+ \sum_{i=1}^{n} m_{total_i}(\tau). \end{aligned}$$

In addition, the first-order transition probability of the containers' resource demands is also quasi-static for a long period and non-uniformly distribution by properly choosing the time slice duration [13]. Moreover, the movement of mobile users is a sequential decision-making process, and has memoryless property [19]. Therefore, we are able to adopt reinforcement learning algorithms in solving this problem as presented in the next session.

3 Container Migration Algorithms in Fog Computing Architecture

In this section, reinforcement learning algorithms are adopted to attain the migration decisions. Moreover, the deep Q-learning algorithm, combined with Q-learning algorithm and DNN, has the advantage in rapid decision-making, which is suitable for unstable FC environment. The reinforcement learning settings and deep Q-learning algorithm are introduced in 3.1. With the help of DNN, the significant amount of information is abstracted to a processable size. In addition, to avoid the 'evil' actions, the optimization of action selection is described in 3.2. Besides, the update of Q-network and the optimization of the training process are described in 3.3. Finally, the deep Q-learning based container migration algorithms are proposed and analyzed in 3.4 and 3.5, respectively.

3.1 Reinforcement Learning Settings

In reinforcement learning algorithms, for each T_τ, the agent collects system state S_τ, and calculates the reward during last time slice $R_{\tau-1}$. Then, the agent selects action A_τ according to pre-defined strategy. After performing the action, the system would transit to the new state $S_{\tau+1}$ in the next time slice. Similarly, the agent calculates reward R_τ and chooses new action $A_{\tau+1}$ according to $S_{\tau+1}$.

In this paper, the system state is based on the delay, the power consumption and the migration cost. Delay is related to $M_i.l(t)$ and $C_i.r(t)$. Thus, let

$$\mathcal{X}(\tau) = \begin{bmatrix} \mathcal{X}_{1,1}(\tau) & \cdots & \mathcal{X}_{1,n}(\tau) \\ \vdots & \ddots & \vdots \\ \mathcal{X}_{m,1}(\tau) & \cdots & \mathcal{X}_{m,n}(\tau) \end{bmatrix}$$

denotes the system state of delay, whose element $\mathcal{X}_{i,j}(\tau)$ represents the state of delay of F_i and C_j, which is defined as:

$$\mathcal{X}_{i,j}(\tau) = \begin{cases} \lfloor \sum_{M_k \in C_j.m(t_0+\tau|T|)} \frac{d_{net_k}(\tau) + k_{comp} \times d_{comp_j}(\tau)}{X_{scale}} \rfloor, & \\ & C_j.l(t_0+\tau|T|) = F_i \\ 0, & C_j.l(t_0+\tau|T|) \neq F_i, \end{cases}$$

where $C_j.m(t_0+\tau|T|)$ denotes the list of mobile application tasks running in C_j during T_τ, $\lfloor\cdot\rfloor$ is the floor function, and X_{scale} is the discretization parameter. $\lfloor\cdot\rfloor$ and X_{scale} map $d_{net_k}(\tau)+k_{comp}\times d_{comp_j}(\tau)$ to the corresponding delay level. Since the power consumption and the migration cost are related to $C.l(\tau)$ and $C.a(\tau)$, the system state during T_τ can be uniquely specified by:

$$S_\tau=\{\mathcal{X}(\tau),C.l(t_0+\tau|T|),C.a(t_0+\tau|T|)\}\in\mathbb{S},$$

where $\mathbb{S}$ denotes the space of the system states.

The action set during T_τ is $A_\tau=\{C.l(t_0+\tau|T|)\}\in\mathbb{A}$, where $\mathbb{A}$ is the set of all possible actions.

We focus on minimizing the total cost over time, so the reward during T_τ is defined as:

$$R_\tau=-(\omega_1 d_{total}(\tau)+\omega_2 p_{total}(\tau)+m_{total}(\tau)). \tag{15}$$

Among all kinds of reinforcement learning algorithms, Q-learning algorithm [21] has an advantage in fast computation, which is consistent with the requirement of rapid decision-making in FC. In such algorithm, the quality of each state-action pair is indicated by Q-value $Q(S_\tau,A_\tau)$, which is stored in Q-matrix. The Q-matrix is initialized to a zero matrix, and each element in Q-matrix indicates the $Q(S_\tau,A_\tau)$ of corresponding state-action pair.

Obviously, the large size of $\mathbb{S}$ and $\mathbb{A}$ could result in the large size of Q-matrix. One possible solution is to discretize $\mathbb{S}$ and store parts of the Q-values. However, it may lose plenty of information, which makes the result inaccurate. To solve this problem, abstracting and storing the information with neural network is considered as an effective solution [27]. Therefore, deep Q-learning algorithm, comprised of DNN phase and Q-learning phase, is adopted in this paper. In deep Q-learning algorithm, the Q-matrix is replaced by a Q-network, which is consisted of a DNN with weights θ and used to efficiently store the Q-value information.

The deep Q-learning algorithm is shown in Algorithm 1. For each episode, the agent first observes S_0. Then, for each T_τ, the agent selects A_τ according to pre-defined strategy, and observes the next state $S_{\tau+1}$. The reward R_τ is calculated by Eq. (15). After that, Q-network is updated and outputs A_τ finally. The action selection and the Q-network update are two main parts of the algorithm, whose details will be described in Sections 3.2 and 3.3, respectively.

3.2 Action Selection

In Q-learning algorithm, the action is selected based on ϵ-greedy algorithm, which consists of exploration and exploitation [28]. In ϵ-greedy algorithm, a threshold ϵ is set in advance, and a random number ϕ is generated each time. The details of action selection are described as below:

- When $\phi>\epsilon$, the action is selected by exploitation. The agent selects the best action $A_\tau=arg\max_{A_\tau}Q(S_\tau,A_\tau)$ according to the Q-values stored in Q-matrix.
- Otherwise, the action is selected by exploration. The agent attempts to get rid of local optimization by selecting a random action.

Algorithm 1. Deep Q-Learning

Input: θ
Output: A_τ
for episode = $1, M$ **do**
  Observe S_0
  for $\tau=1,\kappa$ **do**
    /* Action Selection */
    Select A_τ according to pre-defined strategy
    Observe $S_{\tau+1}$
    Calculate R_τ by Eq. (15)
    /* Q-network Update */
    Update Q-network
    Output A_τ
  end for
end for
end

Optimization of Exploitation. In exploitation, for S_τ, $A_\tau=arg\max_{A_\tau}Q(S_\tau,A_\tau)$ is selected by the agent. In traditional deep Q-learning algorithm, DNN outputs Q-values of all possible actions based on the input S_τ, which has lots of repeated calculations, and is impossible for large $\mathbb{A}$ in FC. To solve this problem, in addition to applying DNN, we maintain a best-action dictionary $Q_m=\{Q_{m_1},Q_{m_2},...,Q_{m_p}\}$. Since S_τ is unique, it can be used as key, and each entry Q_{m_i} is a key-value pair $\langle Q_{m_i}.s,(Q_{m_i}.a,Q_{m_i}.v)\rangle$, where $Q_{m_i}.a$ and $Q_{m_i}.v$ are the best action and corresponding Q-value of state $Q_{m_i}.s$, respectively. In such strategy, the best action of S_τ can be easily obtained by retrieving Q_m with S_τ.

Optimization of Exploration. in exploration, the agent of traditional deep Q-learning algorithm randomly selects an action. Besides, to satisfy the resource constraints in Eq. (7), each random action needs to follow the constraints:

$$\sum_{C_x\in\boldsymbol{C_k}}\frac{C_x.a(t_0+\tau|T|)}{F_j.c}+\sum_{C_x\in\boldsymbol{C_i}}\frac{C_x.a(t_0+\tau|T|)}{F_j.c}-\sum_{C_x\in\boldsymbol{C_h}}\frac{C_x.a(t_0+\tau|T|)}{F_j.c}\le 1,\qquad j=1,2,...,m,$$

where $\boldsymbol{C_k}$ is the set of containers located in F_j. $\boldsymbol{C_i}$ and $\boldsymbol{C_h}$ are the sets of containers that selected to be migrated into/out of F_j, respectively.

Nevertheless, in the most of FC scenarios, the migration system could be ruined by selection of 'evil' actions. For example, massive migration could bring about significant migration cost. Besides, migrating the containers far from users or consolidating too many containers into a node may result in unacceptable round-trip delay [9]. To solve this problem and select an acceptable action in a short time, the agent handles different nodes with different utilization level through different strategies, as shown in Algorithm 2.

In Algorithm 2, $\boldsymbol{F}$ is classified as under-utilization, normal-utilization, and over-utilization groups according to the under threshold th_{under} and over threshold th_{over}. 1) For under-utilization nodes, the purpose is to migrate all containers located in them to other nodes, without any over-utilization nodes addition. So that a lot of power consumption can be saved by powering off these nodes. 2) Meanwhile, for over-utilization nodes, the purpose is to reduce the utilization by migrating some of the containers located in them

to other nodes. Algorithm 2 includes two main steps. First, the nodes are classified into three groups according to resource usage, as shown in line 6 - 13. Then, in line 14 - 29, for each group, the selection policy and allocation policy of the containers are applied to determine the destination nodes of containers which need to be migrated.

Algorithm 2. $ActionSelection(S_\tau, Q_m)$

Input: S_τ, Q_m
Output: A_τ
1: Generate random number ϕ
2: **if** $\phi > \epsilon$ and $Q_m.S_\tau \neq \emptyset$ **then**
3: $A_\tau = Q_m.S_\tau.A$
4: **else**
5: $thUnderFlag = false$
6: **for** $F_j \in \boldsymbol{F}$ **do**
7: **if** $\sum_i C_i.r_{C_i.l(t_0+\tau|T|)=F_j} > th_{over}$ **then**
8: add F_j to $\boldsymbol{F}_{over}$
9: **else if** $\sum_i C_i.r_{C_i.l(t_0+\tau|T|)=F_j} < th_{under}$ **then**
10: add F_j to $\boldsymbol{F}_{under}$
11: $thUnderFlag = true$
12: **end if**
13: **end for**
14: **if** $thUnderFlag = true$ **then**
15: Randomly choose F_i from $\boldsymbol{F}_{under}$
16: **for** C_j in F_i **do**
17: Calculate $\mathcal{R}_{i,j}(\tau)$ by Eq. (17)
18: Select A_τ by Eq. (18)
19: **end for**
20: **else**
21: **if** $F_{over} \neq \emptyset$ **then**
22: Randomly choose F_i from $\boldsymbol{F}_{over}$
23: **else**
24: Randomly choose F_i from $\boldsymbol{F}$
25: **end if**
26: select $C = arg\min m_{total_{i,j}}(\tau')$ by Eq. (16)
27: Calculate $\mathcal{R}_{i,j}(\tau)$ by Eq. (17)
28: Select A_τ by Eq. (18)
29: **end if**
30: **end if**
31: Return A_τ
32: **end**

Selection Policy. It is applied to select the containers which need to be migrated out from each source node. For each under-utilization node, we try to migrate all containers located in it to other nodes. Meanwhile, for each over-utilization node, we calculate the estimated migration cost for each container located in it and then migrate the container with minimal estimated migration cost. The estimated migration cost $m_{total_{i,j}}(\tau')$ is defined as:

$$m_{total_{i,j}}(\tau') = \int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|} \Big(m_{mig_{i,j}}(t)\Big)dt, \tag{16}$$

where $m_{mig_{i,j}}(t)$ is the migration cost of migrating C_i to F_j, which is calculated as:

$$m_{mig_{i,j}}(t) = \frac{C_i.size}{\min(C_i.l(t).b, F_j.b)},$$

where $C_i.size$ is the size of C_i; $C_i.l(t).b$ and $F_j.b$ are the bandwidth of $C_i.l(t)$ and F_j, respectively. Without losing generality, we assume they are constants for simplicity [13].

Allocation Policy. After selecting the migrated containers, we apply allocation policy to determine the destination node of each container. We estimate the migration revenue $\Delta\mathcal{R}_{i,j}(\tau)$ of migrating C_i to $F_j, j = 1, ..., m$, which is defined as:

$$\begin{aligned}\Delta\mathcal{R}_{i,j}(\tau) = {} & \omega_1 d_{total}(\tau) + \omega_2 p_{total}(\tau) - \omega_1 d_{total_{i,j}}(\tau') \\ & - \omega_2 p_{total_{i,j}}(\tau') - m_{total_{i,j}}(\tau'),\end{aligned} \tag{17}$$

where $d_{total_{i,j}}(\tau')$, $p_{total_{i,j}}(\tau')$, and $m_{total_{i,j}}(\tau')$ are the estimated delay, power consumption, and migration cost of migrating C_i to F_j, respectively. In Eq. (17), $d_{total_{i,j}}(\tau')$ is defined as:

$$d_{total_{i,j}}(\tau') = \sum_{k=1}^{l}\big(d_{net_k}(\tau')\big) + \sum_{k=1}^{n}\big(k_{comp} \times d_{comp_k}(\tau')\big),$$

where $\sum_{k=1}^{l}(d_{net_k}(\tau'))$ is the estimated network delay, which is the original network delay plus the change due to migration, and is defined as:

$$\begin{aligned}\sum_{k=1}^{l}\big(d_{net_k}(\tau')\big) = {} & \sum_{k=1}^{l}\big(d_{net_k}(\tau)\big) \\ & + \sum_{M_k \in C_i.m(t_0+\tau|T|)} (k_{net}\log_{10}|M_k.l(t_0+\tau|T|) \\ & - F_j.l| + b_{net} - d_{net_k}(\tau)\big).\end{aligned}$$

And $\sum_{k=1}^{n}(d_{comp_k}(\tau'))$ is the estimated computation delay, which is defined as:

$$\begin{aligned}\sum_{k=1}^{n}\big(d_{comp_k}(\tau')\big) = {} & \sum_{C_k \in \boldsymbol{C}\backslash(\boldsymbol{C_x}\cup\boldsymbol{C_y})} (d_{comp_k}(\tau)) \\ & + \sum_{C_k \in \boldsymbol{C_x}\cup\boldsymbol{C_y}} \frac{\int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|}(C_k.r(t) - w_k(\tau)C_k.a(t))dt}{\int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|}(C_k.r(t))dt},\end{aligned}$$

where $\boldsymbol{C_x}$ and $\boldsymbol{C_y}$ are the sets of containers located in $C_i.l(t_0+\tau|T|)$ and F_j, respectively. $w_k(\tau)$ is used to estimate the allocated resource of C_k, which is defined as:

$$w_k(\tau) = \begin{cases} \frac{\sum_{C_z \in \boldsymbol{C_x}} C_z.r(t_0+\tau|T|)}{\sum_{C_z \in \boldsymbol{C_x}}(C_z.r(t_0+\tau|T|)) - C_i.r(t_0+\tau|T|)}, & C_k \in \boldsymbol{C_x} \\ \frac{\sum_{C_z \in \boldsymbol{C_y}} C_z.r(t_0+\tau|T|)}{\sum_{C_z \in \boldsymbol{C_y}}(C_z.r(t_0+\tau|T|)) + C_i.r(t_0+\tau|T|)}, & C_k \in \boldsymbol{C_y} \end{cases}.$$

If $C_k \in \boldsymbol{C_x}$, the estimated allocated resource of C_k should be incresed. Otherwise, the estimated allocated resource should be reduced. Besides, $p_{total_{i,j}}(\tau')$ is defined as:

$$\begin{aligned}p_{total_{i,j}}(\tau') = {} & \sum_{C_k \in \boldsymbol{C}\backslash(\boldsymbol{C_x}\cup\boldsymbol{C_y})} (p_{total_k}(\tau)) \\ & + \sum_{C_k \in \boldsymbol{C_x}\cup\boldsymbol{C_y}} \int_{t_0+\tau|T|}^{t_0+(\tau+1)|T|} \big(p_{idle} + (p_{\max} - p_{idle}) \times u'_k(t)\big)dt,\end{aligned}$$

where $u'_k(t)$ is the estimated resource utilization, which is defined as:

$$u'_k(t) = \begin{cases} \min\Big\{1, u_{\#\{C_i.l(t_0+\tau|T|)\}}(t) - \frac{C_i.a(t)}{C_i.l(t_0+\tau|T|).c}\Big\}, & C_k \in \boldsymbol{C_x}, \\ \min\Big\{1, u_j(t) + \frac{C_i.a(t)}{F_j.c}\Big\}, & C_k \in \boldsymbol{C_y} \end{cases}$$

where $\#\{C_i.l(t_0+\tau|T|)\}$ is the number of $C_i.l(t_0+\tau|T|)$, e.g., if $C_i.l(t_0+\tau|T|)=F_x$, then $\#\{C_i.l(t_0+\tau|T|)\}=x$. Since C_i is migrated from $C_i.l(t_0+\tau|T|)$ to F_j, if $C_k \in \boldsymbol{C_x}$, $u'_k(t)$ is the original resource utilization $u_{\#\{C_k.l(t_0+\tau|T|)\}}(t)$ minus the resource utilization of C_i. Otherwise, $u'_k(t)$ is $u_j(t)$ plus the resource utilization of C_i.

The total cost can be effectively reduced with larger $\Delta\mathcal{R}_{i,j}(\tau)$. By choosing the largest $\Delta\mathcal{R}_{i,j}(\tau)$, we can migrate the selected container to the corresponding node. Sometimes, some nodes could not load any other container. Thus, we measure the feasibility of migrating C_i to each F_j as:

$$Fea_{i,j}(\tau) = 1\{u_j(t_0+\tau|T|)F_j.c + C_i.a(t_0+\tau|T|) \leq th_{over}\} \cdot 1\{u_j(t_0+\tau|T|)F_j.c \neq 0\}.$$

If $Fea_{i,j}(\tau)=1$, it is feasible to migrate C_i to F_j. For nodes which are feasible to receive the container, we select the node with the largest $\Delta\mathcal{R}_{i,j}(\tau)$ as the destination. Finally, the destination node is selected as:

$$F_{d_i}(\tau) = arg\max_{F_j}\{\mathcal{R}_{i,j}(\tau)|Fea_{i,j}(\tau)=1\}.$$

If no feasible nodes available, we power on one node with minimal migration cost. Moreover, if no other nodes in shutdown mode, we randomly select a node as the destination.

After all containers are processed, A_τ is obtained by:

$$A_\tau = \{\mathcal{A}_1(\tau), \mathcal{A}_2(\tau), ..., \mathcal{A}_n(\tau)\}, \tag{18}$$

where $\mathcal{A}_i(\tau), i=1,...,n$ is defined as:

$$\mathcal{A}_i(\tau) = \begin{cases} C_i.l(t_0+\tau|T|), & C_i \text{ is not selected} \\ F_{d_i}(\tau), & C_i \text{ is selected.} \end{cases}$$

After the selection of A_τ, the agent observes $S_{\tau+1}$, and calculates R_τ, as shown in Algorithm 1. Then, the Q-network is updated, whose detail is described in the next sub-section.

3.3 Q-Network Update

In Q-learning algorithm, each $Q(S_{\tau-1}, A_{\tau-1})$ can be updated online with the learning rule:

$$Q(S_{\tau-1}, A_{\tau-1}) \leftarrow (1-\alpha)Q(S_{\tau-1}, A_{\tau-1}) + \alpha[R_{\tau-1} + \gamma \times \max_{A_\tau} Q(S_\tau, A_\tau)], \tag{19}$$

where α is the learning rate and γ is the discount parameter.

Meanwhile, in deep Q-learning algorithm, information of Q-values is abstractly stored in the DNN. As the state of the system is continually changed in each T_τ, it is necessary for DNN to be adaptively updated. Therefore, the training of DNN based on historical information stored in experience replay memory D is an essential part of deep Q-learning algorithm [27].

The traditional objective of training is to minimize the loss function $L(\theta)$, which is defined as:

$$L(\theta) = E[(y(\tau) - Q(S_\tau, A_\tau; \theta))^2],$$

where $Q(S_\tau, A_\tau; \theta)$ is the output of DNN with weights θ. However, when only one DNN works as a whole, each update to a point in the Q function also influences the whole area around that point. This leads to a problem that with each update, the target is likely to shift. To overcome this problem, a target network [29] is used. The target network provides stable $Q(S_\tau, A_\tau; \theta')$, and $y(\tau)$ is defined as:

$$y(\tau) = E[(1-\alpha)Q(S_{\tau-1}, A_{\tau-1}; \theta') + \alpha(R_{\tau-1} + \gamma \max Q(S_\tau, A_\tau; \theta'))|S_{\tau-1}, A_{\tau-1}],$$

where θ' is the weights of target network, which is reset to θ from time to time.

Algorithm 3. $QNetworkUpdate(S_\tau, A_\tau, R_\tau, S_{\tau+1}, D)$

Input: $S_\tau, A_\tau, R_\tau, S_{\tau+1}, D$
Output: θ

```
Store (S_τ, A_τ, R_τ, S_{τ+1}) in D[τ mod |D|]
Sample D_τ ⊂ D by Eq. (23)
for (S^(j)_{τ-1}, A^(j)_{τ-1}, R^(j)_{τ-1}, S^(j)_τ) in D_τ do
    Calculate Q(S^(j)_τ, A^(j)_τ; θ')
    Calculate y_d(τ) by Eq. (20)
    if Q_{m_i}.s^(j)_τ = ∅ or y(τ) > Q_{m_i}.v then
        Q_{m_i}.v = y_d(τ)
        Q_{m_i}.a = A^(j)_τ
    end if
    Compute error_{τ,j} by Eq. (21)
    Update pr_{τ,j} by Eq. (22)
end for
θ = arg min_θ L(θ)
From time to time reset θ' = θ
Return θ
end
```

In addition, to help the agent perform better with faster learning speed and more stability, we optimize the DNN training with double Q-learning and prioritized experience replay in Q-network update.

Optimization of DNN Training. The max operator $\max_{A_\tau} Q(S_\tau, A_\tau)$ in standard Q-learning and deep Q-learning uses the same Q-values both to choose and to evaluate an action. This makes it more likely to select overestimated values, which will result in overoptimistic estimates. To solve this problem, two Q-value functions are learned by assigning experiences randomly to update one of them in Double Q-learning [30]. One function is used to determine the maximizing action, and the other one is used to estimate its Q-value. With these two functions, the selection is decoupled from the evaluation.

In deep Q-learning algorithm, the target network provides a natural candidate for the second value function. Therefore, the training DNN can be used to evaluate the greedy policy, and the target network can be used to estimate its value, as proposed in Double DQN [22]. The target $y_d(\tau)$ is defined as:

$$y_d(\tau) = E[(1-\alpha)Q(S_{\tau-1}, A_{\tau-1}; \theta') + \alpha(R_{\tau-1} + \gamma Q(S_\tau, arg\max_{A_\tau} Q(S_\tau, A_\tau; \theta); \theta')|S_{\tau-1}, A_{\tau-1}]. \tag{20}$$

The update to the target network stays unchanged from deep Q-learning.

Optimization of Experience Replay. In deep Q-learning, the experience replay is used and all the samples in experience memory are treated equally. However, we can learn more from some transitions than from others. Prioritized experience replay [31] is one strategy that tries to leverage this fact

by changing the sampling distribution. The main idea is that we can learn most from the transitions that do not fit well to our current estimate of the Q function. The error of a transition $Tr_{\tau,j} = (S_{\tau-1}^{(j)}, A_{\tau-1}^{(j)}, R_{\tau-1}^{(j)}, S_{\tau}^{(j)})$ is defined as:

$$error_{\tau,j} = |Q(S_\tau, A_\tau; \theta') - y_d(\tau)|. \tag{21}$$

The error is then converted to priority $P_{\tau,j}$ based on $pr_{\tau,j}$, which is defined as:

$$pr_{\tau,j} = (error_{\tau,j} + \epsilon_{pr})^{\alpha_{pr}}, \tag{22}$$

where ϵ_{pr} is a small positive constant that ensures no transition has zero priority, and $0 \leq \alpha_{pr} \leq 1$ controls how much prioritization is used. Finally, the priority of $Tr_{\tau,j}$ is calculated by the probability of being chosen for replay, which is defined as:

$$P_{\tau,j} = \frac{pr_{\tau,j}}{\sum_k pr_{\tau,k}}. \tag{23}$$

The Q-network update algorithm is shown in Algorithm 3. In Algorithm 3, for each T_τ, the transition $(S_\tau, A_\tau, R_\tau, S_{\tau+1})$ is stored in D. Then, a subset D_τ is sampled from D and used to train the DNN. For each entry $(S_{\tau-1}^{(j)}, A_{\tau-1}^{(j)}, R_{\tau-1}^{(j)}, S_{\tau}^{(j)})$ in D_τ, the corresponding $Q(S_\tau^{(j)}, A_\tau^{(j)}; \theta')$ and $y_d(\tau)$ are calculated from the target network. Besides, due to $Q(S_\tau, A_\tau)$ of the state-action pair (S_τ, A_τ) is obtained through DNN, the best actions stored in Q_m are further updated according to $y_d(\tau)$ during the DNN training. Finally, a gradient descent step is performed in the training network, and the weights θ of the current network is copied to the target network from time to time. The network is offline trained and online updated.

Based on the action selection and Q-network update, our deep Q-learning based container migration algorithm is proposed in Section 3.4.

3.4 Deep Q-Learning Based Container Migration Algorithms

The deep Q-learning based container migration algorithm is proposed in Algorithm 4. The algorithm consists of three main parts: action selection, next state observation and reward calculation, and Q-network update. In action selection, Algorithm 2 is used to select an effective action. Algorithm 3 is used to efficiently train and update Q-network.

3.5 Convergence and Computational Complexity Analysis

For the efficient deep Q-learning based container migration algorithms, it has been proven that Q-learning will gradually converge to the optimal policy under stationary MDP system and sufficiently small learning rate [21]. In addition, the learning rule given in Eq. (19) converges to the optimal Q-function as long as $\sum_\tau \alpha_\tau(S, A) = \infty$ and $\sum_\tau \alpha_\tau^2(S, A) \leq \infty$ for all $(S, A) \in \mathbb{S} \times \mathbb{A}$ [32], where $\alpha_\tau(S, A)$ is the learning rate at T_τ. Hence, the deep Q-learning container migration algorithms will converge to the optimal policy when (1) the system evolves as a stationary memoryless MDP, (2) the learning rate is sufficiently small and (3) the DNN is sufficiently accurate to return the action with optimal $Q(S(\tau), A(\tau))$ estimate. In this paper, the MDP characteristics have been analyzed in 2.3. Besides, the learning rate is sufficiently small ($0 \leq \alpha_\tau \leq 1$) in our problem, and the convergence of deep Q-learning has been illustrated in [27]. In short, the deep Q-learning based container migration algorithms are convergent, and the experimental results will demonstrate the effectiveness of the algorithms.

Algorithm 4. Deep Q-Learning Based Container Migration

Input: θ, Q_m, D
Output: A_τ
1: $Q_m = \emptyset$
2: **for** episode = $1, M$ **do**
3: Observe S_0
4: $D = \emptyset$
5: **for** $\tau = 1, \kappa$ **do**
6: /* Call Algorithm 2 */
7: $A_\tau = ActionSelection(S_\tau, Q_m)$
8: Observe $S_{\tau+1}$
9: Calculate R_τ by Eq. (15)
10: /* Call Algorithm 3 */
11: $\theta = QNetworkUpdate(S_\tau, A_\tau, R_\tau, S_{\tau+1}, D)$
12: Output A_τ
13: **end for**
14: **end for**
15: **end**

For computational complexity, as illustrated in Section 2.1, there are m nodes, n containers, and l mobile users. The main process of the system consists of two parts: the implementation of the algorithms, and the update of the system state. For the implementation of Algorithm 2 and Algorithm 3, the main steps include getting system state, action selection, and reward calculation. The time complexity of getting system state is related to the size of the state, which is $O(m \cdot n)$. For reward calculation, the time complexity is $O(m + n + l)$. And for action selection, when $\phi > \epsilon$, the best action is selected by querying the dictionary Q_m, which is $O(1)$ [33]. Otherwise, the random action is selected by Algorithm 2, which has a complexity of $O(n^2 \cdot (l + n))$. For the update of the system state, it mainly includes the update of the lists maintained by the agent, which are the container list, user list, and node list. For the update of the container list, the time complexity is $O(n \cdot (m + n))$. And for the update of the user list, the time complexity is $O(l \cdot (m + n + n \cdot \log(n))) = O(l \cdot (m + n \cdot \log(n)))$. Besides, the time complexity for the update of node list is $O(l + l \cdot n/m + n) = O(l \cdot n/m)$. All the steps are executed in order. In short, the algorithms have polynomial-time complexity.

4 Experiments

In this section, we first introduce the experimental settings in Section 4.1. Then the experimental results of comparison between container migration and VM migration are shown in Section 4.2. The results of real-world driver trace are discussed in Section 4.3.

4.1 Experimental Settings

We explain the experimental settings of our container migration testbed and real-world driver trace in this sub-section.

Container Migration Testbed. To show the efficiency of containers, we compare the performance of CPU consumption and migration cost between Docker containers and VMs. The container migration testbed consists of a set of servers with four VMs as fog nodes and a set of laptops with two VMs as users. For each VM, its operating system is Ubuntu 14.04 LTS with Docker 1.9 and CRIU 2.2 [34]. CRIU is used to checkpoint and restore the containers. Each node has a container migration controller consisted of six components: migration daemon, system check, filesystem handler, memory handler, handoff, and restore. The migration daemon is responsible for receiving and sending the information of the node and maintaining the network connectivity with other nodes. The system check component is responsible for checking the feasibility of migration, e.g., the kernel, Docker, and CRIU requirements. The filesystem handler and memory handler components are used to handle the root filesystem and memory pages of the container, respectively. The handoff component is used to handle the migration process. And the restore component is used to restore the container.

Once the migration controller receives the migration task, it first checks the status and gets the container information. Second, it handles the root filesystem, compresses the files, and sends it to the destination node. Third, it pre-dumps the memory [35], compresses the memory with LZ4 [36], and sends the memory pages to the destination node. LZ4 is a lossless data compression technology used for rapid compression of the memory pages. Then, it dumps the memory and sends it to the destination node. After that, the container is restored in the destination node. The migration time is the time from the beginning of dumping the memory to the end of restoring the container.

Nginx web servers are run in the VMs in the computer, and three types of containers are run behind the Nginx web server: a WordPress website with MySQL database, a Ghost website with SQLite 3 database, and a website with static pages. Tasks of web pages refreshing are continuously submitted by Webbench [37] from laptops to the corresponding Nginx web servers with different request numbers as different workload. We use Linux traffic control [38] to control the delay between users and nodes. Besides, we use Sysstat [39] to get the CPU consumption.

Real-World Driver Trace. Moreover, we perform experiments using real-world San Francisco taxi traces obtained on May 31, 2008 [40] with Python. Similar to Wang et al. [26], we assume that the nodes are deployed in a grid structure in the central area of San Francisco. The locations of the taxi traces are collected via GPS, which vary from latitude 32.87 to latitude 50.31, and from longitude -127.08 to longitude -122.0. The area is separated into 10^8 cells. We consider 67 nodes located in some of the cells, and 200 mobile users (taxies) in total. The capacities of the nodes are a uniform distribution in $[50, 100]$. All mobile users are active, whose requests are changed when the passengers are on board or off the cab. A unique instance from the nearest node is allocated to each request of mobile users. At a time, at most one task is requested by the users.

As for parameters, for d_{net}, we set $f = 2.5MHz$, $h_b = 35m$, $h_r = 1m$, and $c_m = 3dB$ [25]. $d_i(t)$ is calculated by $d_i(t) = |M_i.l(t) - F_j.l|$, where F_j is the assigned node. Besides, $X_{scale} = 3$, and $th_{over} = 0.9$ [9]. In addition, for Q-learning settings, parameters α, γ, ϵ is set to 0.1,0.9 and 0.9, respectively. Moreover, the power models of each node are shown in Table 2 [9].

TABLE 2
The Power Consumption at Different CPU Utilization in Watts

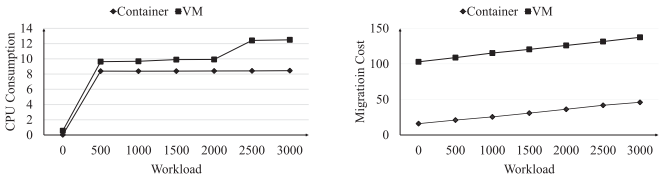
CPU Utilization(%)	**0%**	**10%**	**20%**	**30%**	**40%**	**50%**
HP ProLiant G4	86	89.4	92.6	96	99.5	102
HP ProLiant G5	93.7	97	101	105	110	116
CPU Utilization(%)	**60%**	**70%**	**80%**	**90%**	**100%**	
HP ProLiant G4	106	108	112	114	117	
HP ProLiant G5	121	125	129	133	135	

For the DNN architecture, we use a four-layer fully connected neural network. Our choices of four layers of neural networks are based on our experimental data. We have run many experiments with different depth of the neural networks, and the results show that the 4-layer network is good enough to handle this problem. However, due to space limitation, we are unable to put the detailed analysis here, and Liu et al. [41] have done the similar analysis. Besides, S_τ and A_τ are stored in a tuple and converted into a 12127×1 vector as the input of DNN. A Rectified Linear Unit (ReLU) activation function [42] is added between the first layer and the second layer. The output of DNN is corresponding to $Q(S_\tau, A_\tau)$ with tuple (S_τ, A_τ). Different nodes share the same DNN architecture and weights.

We compare the deep Q-learning container migration algorithms (DQLCM) with traditional deep Q-learning based algorithm (TDQL), discretization based Q-learning algorithm (QL) [21], static threshold algorithm (THR), median absolute deviation algorithm (MAD) and interquartile range regression algorithm (IQR) [43]. In QL, actions are selected by action exploration and exploitation. To handle the large state set and action set, discretization technique of state set and action set is adopted in QL. Other algorithms are based on the detection and resource allocation of under-utilization and over-utilization of the nodes. In these algorithms, some of the containers in over-utilized nodes, whose utilization is higher than th_{over}, are migrated to other nodes to decrease the number of over-utilized nodes. Besides, all the containers in under-utilized nodes, whose utilization is lower than th_{under}, are migrated to other nodes to increase the number of empty nodes. In THR algorithm, th_{under}, th_{over} are set statically for under-utilized and over-utilized nodes, while th_{under}, th_{over} are dynamically predicted in MAD and IQR algorithms. The MAD algorithm measures the statistical dispersion of each task, which is a more robust estimator of scale than the sample variance or standard deviation. Besides, IQR algorithm also measures the statistical dispersion of each task, which has a breakdown point of 25%.

4.2 Comparison between Container and VM

The monitoring results of CPU consumption and migration cost are shown in Fig. 3. The average CPU consumption when running applications within containers and VMs are shown in Fig. 3a. We can see that when workload grows, the CPU consumption of VM becomes much larger than the container. And Fig. 3b shows the average migration cost of containers and VMs. The migration cost mainly consists of the stop time, filesystem handling and memory handling time in



(a) CPU Consumption (b) Migration Cost

Fig. 3. Performance of Docker Container against VM.

the source node, the file transmission time between two nodes and the restore time, start time in the destination node. Besides, the file transmission time of VM is much more than container due to the transmission of extra guest OS files of VM. In short, containers are more efficient than VMs in FC.

4.3 Experimental Results of Real-World Driver Trace

The performance of algorithms with different ω_1 and ω_2 is shown in Figs. 4 and 5, respectively. Fig. 4 indicates when ω_1 increases, the advantage of the delay in DQLCM increases, which is reasonable since the larger ω_1, the greater influence from the delay. We also illustrate the delay of these algorithms in Fig. 4a. We can see that DQLCM has less delay than all other algorithms. Fig. 4b demonstrates the power consumption when ω_1 changes. The power consumption of DQLCM is better than most of the baselines. Since the IQR and MAD algorithms focus only on reducing power consumption, while our DQLCM algorithm focuses not only on power consumption, but also on delay and migration cost, and the performance of DQLCM is about the same with these baselines. As illustrated in Eq. (5), the power consumption is obtained through the resource utilization of nodes, which is the CPU utilization in this paper. So through the performance of power consumption, we can also conclude the performance of CPU for each node. As shown in Fig. 4c, for average migration cost, DQLCM outperforms all baselines. In Fig. 4d, the average total cost of all algorithms is shown, and it is obvious that the DQLCM is better than all other algorithms.

In Fig. 5, the performance with different ω_2 is shown. As shown in Fig. 5a, DQLCM has less delay than the baseline methods, i.e., DQLCM < TDQL < QL < THR < MAD < IQR. In Fig. 5b, while increasing ω_2, DQLCM has less power consumption than the three algorithms, which also gives DQLCM < IQR < MAD < THR < TDQL < QL. This reflects that the power consumption reduction in DQLCM is the highest. And the migration cost of DQLCM is less than all other baselines as shown in Fig. 5c. For the average total cost, as shown in Fig. 5d, the DQLCM outperforms all other algorithms.

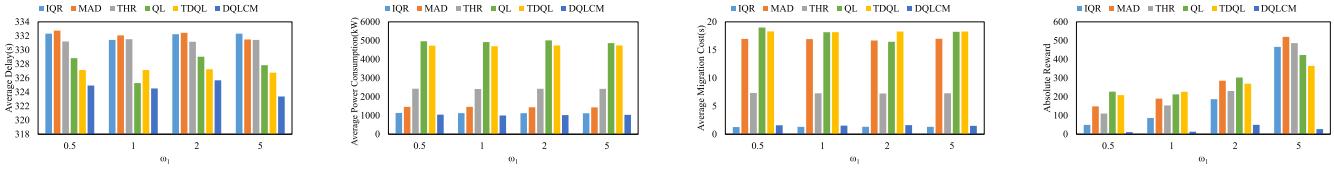
The performance of algorithms with different under threshold th_{under} is shown in Fig. 6. As shown in Figs. 6a and 6b, the lower the th_{under}, the more advantage DQLCM is in average delay and average power consumption. And the migration cost of DQLCM is less than all other algorithms, such that DQLCM < THR < MAD < TDQL < QL, as shown in Fig. 6c. Also, the average total cost is shown in Fig. 6d, which demonstrates that DQLCM is better than all other algorithms.

In short, DQLCM has an overall best performance than all baselines and outperforms the baseline approaches 2.9, 48.5 and 58.4 percent on average regarding the delay, power consumption, and the migration cost, respectively.

In Fig. 7, the performance of delay and power consumption in one iteration is shown. We can see that the performance of DQLCM outperforms the baselines when the iteration number grows. And it finally convergences to a stable state.

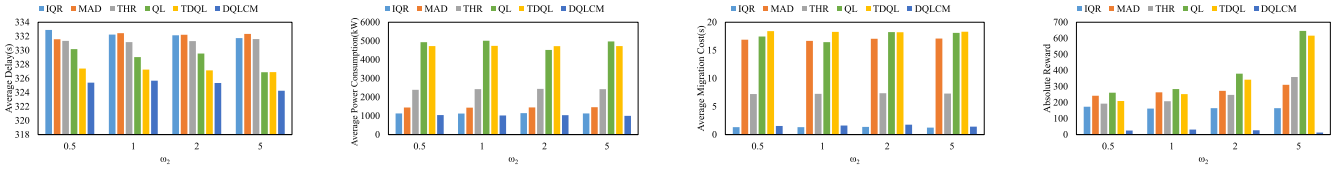
5 Related Work

Fog Computing: There have been many types of research of FC. Based on the architecture proposed by CISCO, HP labo-



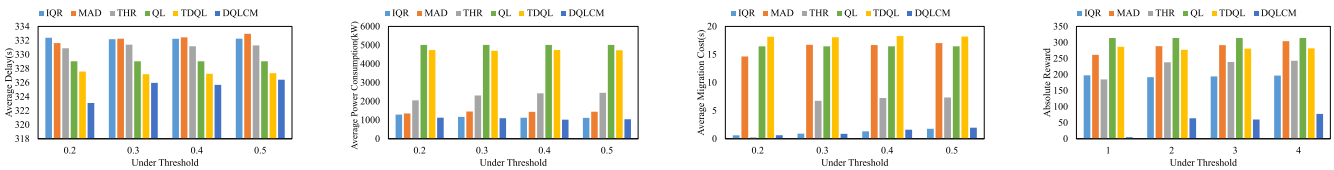
(a) Average Delay (b) Average Power Consumption (c) Average Migration Cost (d) Average Total Cost

Fig. 4. Performance with different ω_1.



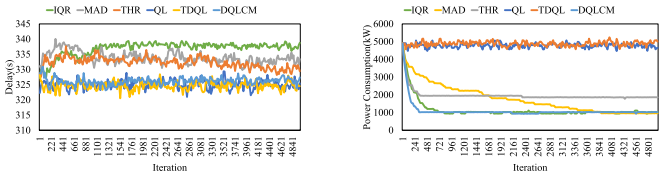
(a) Average Delay (b) Average Power Consumption (c) Average Migration Cost (d) Average Total Cost

Fig. 5. Performance with different ω_2.



(a) Average Delay (b) Average Power Consumption (c) Average Migration Cost (d) Average Total Cost

Fig. 6. Performance with different under threshold.



(a) Delay (b) Power Consumption

Fig. 7. Performance in one iteration.

ratory further proposes the concept of FC [44]. Hong et al. [45] design a mobile fog computing system which explained that the FC model could provide support for mobility. In addition, there are some concepts similar to FC such as mobile computing [46], mobile cloud computing [3], edge computing [47], etc. Compared with mobile computing, FC, while maintaining the support for mobility, combines with powerful computation and storage capabilities possessed by cloud computing [4]. Mobile fog computing specifically aims at mobile devices [48]. However, since the tasks are uploaded to remote cloud data center, the problem of delay could not be solved compared with FC. As for edge computing, it has many similarities with FC, which combines the advantages of mobile computing and cloud computing. Edge computing is more concerned with users, while FC is more concerned with the service providers of fog layer and the owners of all kinds of equipment in the fog layer [49].

The research of FC is mainly about the hardware virtualization technology [16], and the allocation and monitoring of hardware to reduce the delay and power consumption [14]. The virtual machine scheduling is the focus of FC. The purpose is to allocate the virtual machines to complete the task with minimal cost, and to achieve a higher QoS. Different algorithms have different definitions of task benefits and QoS, which results in a variety of scheduling algorithms. In general, the task benefits and QoS are defined as follows: storage capacity [15], delay [14], power consumption [13], utility, network utilization and migration time [12].

For mobility of mobile applications in FC, there is a trade-off between the QoS and container migration cost [26]. But only a few existing researches have studied this problem [26], [50], [51], [52]. A mobility-aware FC architecture is proposed in [50]. The performance of nodes with user mobility is studied in [51], but decisions on whether and where to migrate the service in the node are not considered. All of the approaches in [26], [52] do not explicitly consider multiple users and assume specific structures of the cost function that only related to the distance between users and nodes.

Container Migration. VM migration is a resource-intensive operation and is a critical issue in the data center. In FC, a general architecture to support VM migration was proposed in [8]. VM Handoff [53] is a technique for transferring VM-encapsulated execution when users move in FC.

Compared with VM, container is a lightweight virtualization technology [17] which is more suitable for FC. With the help of Docker, more researches of container migration are proposed [54]. But container migration in FC is a relatively new area which is first considered in [55] with no migration strategy. Besides, Ma et al. [56] implement a prototype system of container migration across edge servers.

Reinforcement Learning. Reinforcement learning attracts more and more attention recent years. It is often used in controlling and routing [57]. The DNN is successfully combined with reinforcement learning and achieved an excellent result in winning game scores [27], [29]. Mao et al. [58] use the deep reinforcement learning in network resource scheduling. Liu et al. [41] propose a novel hierarchical framework in cloud data center which weighs the overall resource allocation and power management issues. The best trade-off between power consumption and processing delay can be effectively achieved with deep reinforcement learning technology, combined with auto-encoding, LSTM, and other neural networks.

However, there are only a few researches of combining reinforcement learning with container migration management in FC. Some studies of VM migrations in data center [13], [41] are proposed, but they do not support the mobile application mobility.

6 Discussion

From the experimental results, we can see the effectiveness of our algorithms. The following issues deserved further investigations.

Battery-Powered Nodes. In FC, there are indeed many lightweight battery-powered nodes and clients. However, we consider those battery-powered nodes without enough computation resource to support containers will be shut down when the battery runs out, which is not our focus. Besides, we only focus on the power consumption of fog nodes as they are the main roles of container migrations. In our model, the weight of power consumption in Eq. (7) can be adjusted to reflect the different levels of power consumptions, and the results are shown in 4.3.

Container versus VM. Containers are lighter than VMs, as shown in Fig. 2. Moreover, a container is consisted of some read-only layers and one read and write layer. When migrating a container, only the read and write layer need to be migrated [56]. This can further leverage the unique features of containers so that the migration cost in Eq. (6) can be further reduced.

Revenue Estimation. In Section 3, to select a better action, we estimate the migration cost and the migration revenue. The resource requirements, resource allocations, the transmission size, and real-time bandwidth could be dynamically changing during the execution. However, the aim of revenue estimation is to select a better random action for faster convergence of the network. Since we aim to optimize the long-term revenue, even sometimes it is not so accurate, it does not affect the results in the long term.

Placement of Migration Decision Service. The placement of migration decision service remains a problem in FC. Truong et al. [59] place a fog controller with an SDN controller between the cloud and the fog nodes. The fog controller is connected to the cloud and the fog nodes with broadband communications. In this paper, we make the same assumption and place the service in the nearest cloud.

7 Conclusion

In this paper, we have modeled the container migration problem of mobile application tasks in FC as a large-scale MDP problem. We first define the system model, whose cost function consists of delay, power consumption, and

migration cost. Then we proposed the deep Q-learning based container migration algorithms. To achieve fast decision- making, we optimize random action selection in exploration and DNN training strategy in Q-network update. Experiments with real-world data trace have shown that our algorithms substantially reduce the delay, power consumption and migration cost as compared with the existing baselines. Future work will consider the container migration across the nodes and the cloud.

Acknowledgments

This work is supported by DCT-MoST Joint-project No. (025/2015/AMJ); Startup Funds of University of Macau Nos: CPG2018-00032-FST & SRG2018-00111-FST; Chinese National Research Fund (NSFC) Key Project No. 61532013; National China 973 Project No. 2015CB352401; Shanghai Scientific Innovation Act of STCSM No.15JC1402400 and 985 Project of Shanghai Jiao Tong University: WF220103001.

References

[1] H. Zhang, Q. Zhang, and X. Du, "Toward vehicle-assisted cloud computing for smartphones," *IEEE Trans. Veh. Technol.*, vol. 64, no. 12, pp. 5610–5618, Dec. 2015.

[2] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, "Foggy: A framework for continuous automated IoT application deployment in fog computing," in *Proc. IEEE Int. Conf. AI Mobile Serv.*, 2017, pp. 38–45.

[3] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Commun. Mobile Comput.*, vol. 13, no. 18, pp. 1587–1611, 2013.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. ACM SIGCOMM Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.

[5] F. Bonomi, "Connected vehicles, the internet of things, and fog computing," in *Proc. ACM Int. Workshop Veh. Inter-Netw.*, 2011, pp. 13–15.

[6] J. Li, J. Jin, D. Yuan, M. Palaniswami, and K. Moessner, "Ehopes: Data-centered fog platform for smart living," in *Proc. IEEE Int. Telecommun. Netw. Appl. Conf.*, 2015, pp. 308–313.

[7] X. Hou, Y. Li, M. Chen, D. Wu, D. Jin, and S. Chen, "Vehicular fog computing: A viewpoint of vehicles as the infrastructures," *IEEE Trans. Veh. Technol.*, vol. 65, no. 6, pp. 3860–3873, 2016.

[8] L. F. Bittencourt, M. M. Lopes, I. Petri, and O. F. Rana, "Towards virtual machine migration in fog computing," in *Proc. IEEE Int. Conf. P2P Parallel Grid Cloud Internet Comput.*, 2015, pp. 1–8.

[9] X. Zhou, K. Wang, W. Jia, and M. Guo, "Reinforcement learning-based adaptive resource management of differentiated services in geo-distributed data centers," in *Proc. IEEE Int. Symp. Quality Serv.*, 2017, pp. 1–6.

[10] Q. Zhang, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, "Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 510–519.

[11] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2008, pp. 337–350.

[12] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang, "Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1171–1181, Dec. 2016.

[13] Z. Han, H. Tan, G. Chen, R. Wang, Y. Chen, and F. C. Lau, "Dynamic virtual machine management via approximate markov decision process," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[14] F. Farahnakian, P. Liljeberg, and J. Plosila, "Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning," in *Proc. IEEE Int. Euromicro Conf. Parallel Distrib. Netw.-Based Process.*, 2014, pp. 500–507.

[15] S. Yi, C. Li, and Q. Li, "A survey of fog computing: Concepts, applications and issues," in *Proc. ACM Workshop Mobile Big Data*, 2015, pp. 37–42.

[16] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Proc. IEEE Workshop Hot Top. Web Syst. Technol.*, 2015, pp. 73–78.

[17] D. Willis, A. Dasgupta, and S. Banerjee, "Paradrop: A multi-tenant platform to dynamically install third party services on wireless gateways," in *Proc. ACM Int. Workshop Mobility Evolving Internet Archit.*, 2014, pp. 43–48.

[18] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 171–172.

[19] Y. Zhai, Y. Wang, I. You, J. Yuan, Y. Ren, and X. Shan, "A DHT and MDP-based mobility management scheme for large-scale mobile internet," in *Proc. IEEE Conf. Comput. Commun. Workshops.*, 2011, pp. 379–384.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, vol. 1, no. 1. Cambridge, MA, USA: MIT Press, 1998.

[21] C. J. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3/4, pp. 279–292, 1992.

[22] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 2094–2100.

[23] R. Buyya, A. Beloglazov, and J. H. Abawajy, "Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges," in *Proc. Int. Conf. Parallel and Distrib. Process. Tech. and Appl.*, Las Vegas, Nevada, USA, July 12–15, 2010, vol. 2, pp. 6–20, 2010.

[24] C. Action, "Digital mobile radio towards future generation systems," European Communities, Tech. Rep. EUR 18957,1999.

[25] V. Abhayawardhana, I. Wassell, D. Crosby, M. Sellars, and M. Brown, "Comparison of empirical propagation path loss models for fixed wireless access systems," in *Proc. IEEE Veh. Technol. Conf.*, 2005, pp. 73–77.

[26] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," in *Proc. IFIP Int. Conf. Netw.*, 2015, pp. 1–9.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learn. Workshop*, 2013.

[28] M. Tokic and G. Palm, "Value-difference based exploration: Adaptive control between epsilon-greedy and softmax," in *Proc. 34th Annu. German Conf. Adv. Artificial Intelligence*, 2011, pp. 335–346.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[30] H. V. Hasselt, "Double q-learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2010, pp. 2613–2621.

[31] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *Int. Conf. Learning Representations*, Puerto Rico, 2016.

[32] T. Jaakkola, M. I. Jordan, and S. P. Singh, "Convergence of stochastic iterative dynamic programming algorithms," *Advances in Neural Info. Process. Syst.*, pp. 703–710, 1994.

[33] TimeComplexity, Time complexity - python wiki, 2017. [Online]. Available: https://wiki.python.org/moin/TimeComplexity

[34] CRIU, Criu, 2017. [Online]. Available: https://criu.org/Docker

[35] CRIU, Memory dumping and restoring, 2017. [Online]. Available: https://criu.org/Memory_dumping_and_restoring

[36] LZ4, Lz4 - extremely fast compression, 2018. [Online]. Available: http://lz4.github.io/lz4/

[37] Z. jazyk, Web bench 1.5, 2004. [Online]. Available: http://home.tiscali.cz/ cz210552/webbench.html

[38] M. A. Brown, Traffic control howto, 2006. [Online]. Available: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO

[39] S. Godard, Sysstat, 2018. [Online]. Available: http://sebastien.godard.pagesperso-orange.fr

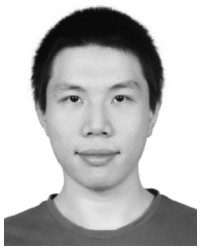
[40] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, "CRAWDAD dataset epfl/mobility (v. 2009-02-24)," Feb. 2009. [Online]. Available: http://crawdad.org/epfl/mobility/20090224

[41] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning," in *Proc. IEEE 37th Int Conf. Distrib. Comput. Syst.*, 2017, pp. 372–382.

[42] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, 2011, pp. 315–323.

[43] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency Comput. Practice Exp.*, vol. 24, no. 13, pp. 1397–1420, 2012.
[44] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, 2014.
[45] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proc. 2nd ACM SIGCOMM Workshop Mobile Cloud Comput.*, 2013, pp. 15–20.
[46] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2013.
[47] P. Garcia Lopez , A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, 2015.
[48] W. Li, Y. Zhao, S. Lu, and D. Chen, "Mechanisms and challenges on mobility-augmented service provisioning for mobile cloud computing," *IEEE Commun. Mag.*, vol. 53, no. 3, pp. 89–97, Mar. 2015.
[49] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
[50] S. Soo, "Towards proactive mobility-aware fog computing," M.S. thesis, Institute of Computer Science, University of Tartu, Tartu, Estonia, 2017.
[51] T. Taleb and A. Ksentini, "An analytical model for follow me cloud," in *Proc. Global Commun. Conf.*, 2013, pp. 1291–1296.
[52] A. Ksentini, T. Taleb, and M. Chen, "A markov decision process-based service migration procedure for follow me cloud," in *Proc. IEEE Int. Conf. Commun.*, 2014, pp. 1350–1354.
[53] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, "You can teach elephants to dance: Agile vm handoff for edge computing," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, 2017, Art. no. 12.
[54] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder, "Docker containers across multiple clouds and data centers," in *Proc. IEEE/ACM 8th Int Conf Utility Cloud Comput.*, 2015, pp. 368–371.
[55] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Migrating running applications across mobile edge clouds: poster," in *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, 2016, pp. 435–436.
[56] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, 2017, Art. no. 11.
[57] H. A. Al-Rawi, K.-L. A. Yau, H. Mohamad, N. Ramli, and W. Hashim, "A reinforcement learning-based routing scheme for cognitive radio ad hoc networks," in *Proc. IFIP Wireless Mobile Netw. Conf.*, 2014, pp. 1–8.
[58] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Top. Netw.*, 2016, pp. 50–56.
[59] N. B. Truong, G. M. Lee, and Y. Ghamri-Doudane, "Software defined networking-based vehicular adhoc network with fog computing," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage.*, 2015, pp. 1202–1207.

Zhiqing Tang received the BS degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015. He is currently working towards the PhD degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include fog computing, resource allocation, and reinforcement learning.



Xiaojie Zhou received the BS degree from the School of Data and Computer Science, Sun Yat-sen University, China, in 2016. He is currently working toward the master's degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include fog computing, resource scheduling, and reinforcement learning.

Fuming Zhang is currently working toward the bachelor's degree in the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University. He is currently a member of CyberSpace Intelligence Computing Lab. He was a software development engineer intern with Microsoft Asia-Pacific Technology Co., Ltd, Shanghai, China. His research interests include fog computing, resource scheduling, distributed computing, and machine learning.

Weijia Jia (SM'08) received the BSc and MSc degrees from Center South University, China, in 1982 and 1984, respectively, and the PhD degree from Polytechnic Faculty of Mons, Belgium, in 1993. He is currently a full-time Zhiyuan chair professor with Shanghai Jiaotong University. He is leading currently several large projects on next-generation Internet of Things, environmental sensing, smart cities and cyberspace sensing and associations etc. He worked with German National Research Center for Information Science (GMD) from 1993 to 1995 as a research fellow. From 1995 to 2013, he has worked with City University of Hong Kong as a full professor. He has published more than 400 papers in various IEEE Transactions and prestige international conference proceedings. He is a senior member of the IEEE.

Wei Zhao (F'01) is currently the rector of the University of Macau, China. Before joining the University of Macau, he served as the dean of the School of Science, Rensselaer Polytechnic Institute. Between 2005 and 2007, he served as the director of the Division of Computer and Network Systems in the US National Science Foundation when he was on leave from Texas A and M University, where he served as senior associate vice president for research and professor of computer science. He has made significant contributions in distributed computing, real-time systems, computer networks, cyber security, and cyber-physical systems. He is a fellow of the IEEE.