Doctoral Dissertation

Dynamic Service Placement in Edge Computing Environment using Reinforcement Learning-based Live Migration

Se Yeon Jeong (정 세 연)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2022

강화학습 기반의 Live Migration 정책을 이용한 엣지 컴퓨팅 환경에서의 동적 서비스 배치

Dynamic Service Placement in Edge Computing Environment using Reinforcement Learning-based Live Migration

Dynamic Service Placement in Edge Computing Environment using Reinforcement Learning-based Live Migration

by

Se Yeon Jeong
Department of Computer Science and Engineering
Pohang University of Science and Technology

A dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Computer Science and Engineering

Pohang, Korea
06. 15. 2022
Approved by
James Won-Ki Hong (Signature)
Academic advisor

Dynamic Service Placement in Edge Computing Environment using Reinforcement Learning-based Live Migration

Se Yeon Jeong

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH

06. 15. 2022

Committee Chair	James Won-Ki Hong	(Seal)
Member	Jae-Hyeong Yoo	(Seal)
Member	Young-Joo Suh	(Seal)
Member	Inseok Hwang	(Seal)
Member	Jong Kim	(Seal)

DCSE 정 세 연 Se Yeon Jeong
20182714
Dynamic Service Placement in Edge Computing Environment using Reinforcement Learning-based Live Migration.
강화학습 기반의 Live Migration 정책을 이용한 엣지 컴퓨팅 환경에서의 동적 서비스 배치.
Department of Computer Science and Engineering , 2022, 89p
Advisor : James Won-Ki Hong.
Text in English.

ABSTRACT

To meet various service requirements in the 5G era, edge computing has become prevalent to provide not only operators with the efficiency in resource management but also users with the improved Quality of Experience (QoE). In accordance with the current Multi-access Edge Computing (MEC) which allows deployment of mission-critical services in the proximity of users, operators need to make proper service placement decisions across the network core or edges. Dynamic service placement via live migration, which enables any service instance to dynamically move with minimum downtime, is a persistent decision making process to optimize the service arrangement while responding the changing environment. The evolution of underlying cloud orchestration tools allows to use live migration in production data centers, with the purpose of load balancing and high availability. However, the diversity in edge

computing environment complicates the application of dynamic service placement in practice, with the various requirements of 5G services and the hierarchical structure of data center networks. In this thesis, a methodology for dynamic service placement is proposed using deep reinforcement learning (DRL) algorithms based on the interaction between agent (i.e., policy generator) and edge computing environment. The proposed multi-agent DRL algorithms not only generalize the dynamics of edge computing to the RL context of environment, agent, state, action and reward, but also enable agents to optimize placement policy with trials of migrating service instances across the multiple data centers and their estimated worth. The proposed algorithms also consider the migration cost of service downtime, and service availability depending on server failure in validating live migration decisions. Comparing to the existing single-agent DRL algorithm for dynamic service placement, the proposed multi-agent DRL algorithms decrease the average service latency by 9% and increases average service availability by 67% with 72% less number of migrations, in the simulation of edge computing environment where 1,000 service requests are deployed among 120 servers across 16 data centers.

Contents

List of Tables

List of Figures

I. Introduction

1.1 Motivation

To meet various service requirements in the 5G era, edge computing has become a key enabler to provide not only operators with the efficiency in resource management but also users with the improved Quality of Experience (QoE). Nowadays, the evolution of edge computing infrastructure (e.g., high-spec COTS server) and virtualization technology (e.g., containerized microservices) extends to Multi-access Edge Computing (MEC) which allows deployment of mission-critical services in the proximity of users [2]. Besides the importance of rapid adoption of latest technique in a particular domain of edge computing, the importance of optimization in service orchestration is still persistent, considering that increasing demand of numerous services in massive machine type communication (mMTC) [3] and their variety in requirements of ultra-reliable and low latency communication (uRLLC) [4].

Service placement in edge computing is a decision to retrieve the optimal location of a running service instance, either central cloud data center (DC) or edge DC [5]. The necessity of placement decision is originated from the heterogeneity of edge computing that includes the followings: geographically remote DCs, differences in their resource capacity and various service requirements from users at distributed network edges. Therefore, operators or service providers should pay close attention so that the placement decision can be beneficial to resource efficiency and service quality. One step forward, dynamic service placement is a decision-making process to achieve the optimality in the location of services during their whole life cycle (i.e., from initial deployment to end). Since services are requested over different time and their termination is non-deterministic, a dynamic service placement problem in edge computing has been considered as a representative optimization problem [6][7][8].

After a placement decision is made, the relocation of a running service instance is realized by auto-scaling and/or live-migration in practice. Auto-scaling is an operation to automatically adjust computing resources assigned to a service instance. In horizontal scaling, the number of a service instance can be increased (scale-out) with the placement decision on the additional instances based on the concerning goals (e.g., placement in the most underutilized machine) [9]. Since sessions are newly established between the additional service instances and users, a mechanism for session management should be considered in the service architecture if an operator needs to preserve the existing session among instances. Whereas, live migration does placement in a more direct way by migrating the target instance to a new location (migration destination) with minimum interruption to the service availability, which means existing sessions between the instance and users are preserved. While live migration for Virtual Machine (VM) has been mature for the last decade with practical usage in production DCs [10][11] and implementation variants (e.g., reducing total migration time or service downtime), live migration for container has come to the fore currently with the emerge of microservices and cloud-native network function (CNF) [12][13]. Apart from the implementation details, live migration in its original concept is suitable to realize dynamic service placement in edge computing for various goals (e.g, load balancing or energy reduction).

Dynamic service placement via live migration is also expected to have an important role in private 5G networks [14]. Since CAPEX/OPEX is a main hurdle for small and medium-sized enterprises in composing their private 5G networks, a hybrid network that only equips minimized on-premise 5G cores (e.g., UPF) and can bring other functions on demand to its MEC host (i.e., cached from the public cloud of operator) can be a reasonable use case where live migration policy offers flexibility and high availability [15][16]. The successful integration of private 5G with MEC is expected to realize various 5G use cases [17].

1.2 Problem Statement and Research Goals

Considering the importance of orchestrating different types of services in edge computing and the complexity of 5G networking use cases, in this thesis, a methodology of dynamic service placement in edge computing is proposed using Deep Reinforcement Learning (DRL)-based live migration policy. The main problem statement this research targets to address is summarized by the followings:

- **Edge computing environment:** a topology, with host machines across multiple DCs in network core and edges and different link costs, is considered as the target environment with system dynamics.
- **Edge computing services:** multiple service requests, with different specifications of Service Level Agreements (SLAs), user locations, request arrival times and amount of demanding resources, are dynamically instantiated among host machines for their service duration.
- **Live migration controller:** a migration controller periodically generates decisions of migrating any service instance to new placement location, with the aim of maximizing the objective function (Equation 1.1) between two terms in trade-off (i.e., is it worth deploying a service in the network edge with the risk of lower availability?).

$$Obj. = (-average\_service\_latency) + average\_service\_availability \quad (1.1)$$

Since the complexity in 5G networking is highly expected to be increased with the heterogeneity of edge computing, MEC services and private 5G specifics, the difficulty of optimization in management and orchestration policy is also increased. To overcome related challenges, in this thesis, DRL is used to solve the optimization problem of dynamic service placement in edge computing. Accordingly, the presented

problem statement is modeled as a DRL context of environment, agent, state, action and reward, so that the agent (i.e., migration controller) can learn from interactions with the environment to reach the optimality in its action behavior by evaluating the resulted rewards which are related to the objective function. Major contributions of this research are as follows.

- The problem of dynamic service placement in edge computing is formulated as the corresponding DRL model of environment, agent, state, action and reward.
- Multi-agent DRL algorithms for dynamic service placement are proposed to optimize live migration policy so that average service latency and service availability can be improved. With the arbitration of the central (global) migration controller, each of distributed agents learns own placement policy only for services hosted in the dedicated DC to reduce the problem scale and improve the learning performance.
- Live migration service downtime is modeled to relate with migration cost so that the proposed algorithms can minimize the cost in migration decision making. The resulted policy revokes any migration decision that is expected to violate SLA availability of the target service. Failure scenario in any edge computing server is also considered to be related with service availability. Those considerations for the improved practicality are modeled in the proposed DRL algorithms since both operators and users favor less number of migrations as possible.
- A simulator for edge computing environment is implemented so that different service orchestration algorithms can be developed and verified in a scalable manner. The event-driven simulator with various Python library is published on an open repository for interested researchers.
- In the edge computing simulation of deploying 1,000 service requests among 120 machines across 16 DCs, the proposed algorithms for dynamic service

placement decrease the average service latency by 9% and increase the average service availability by 67%, under 72% less number of total migrations than a single-agent algorithm based on the existing work. The proposed algorithms also decrease average service latency by 80% than a baseline service deployment algorithm that does not support dynamic placement via live migration.

1.3 Organization

The remainder of this thesis is organized as follows. In Chapter 2, background information and the existing literature related to this research are presented. In Chapter 3, the detailed design architecture of the dynamic service placement algorithms with the multi-agent DRL model and the migration cost model is provided. In Chapter 4, implementation details on the edge computing simulator and considerations on various edge computing specifics are descried. In Chapter 5, experiment results are provided regarding the performance benefits of the proposed dynamic service placement algorithms, comparing to the existing work. Finally, discussion on remarkable observations and conclusion of this thesis is described with future work.

II. Background and Related Work

In this chapter, first the underlying technique for design and implementation of the proposed methodology is introduced: VM live migration, service placement in edge computing and DRL. Then, related literature is surveyed to provide existing limitations and how they are addressed in this research.

2.1 Background

2.1.1 VM Live Migration

VM live migration is to move a running VM from the current host to another server while preserving the availability (e.g., network connectivity) of the service application inside the VM. Many companies that lead the market of server virtualization have commercialized VM live migration in their products, and VM live migration has became an essential functionality in DC management for cloud computing. Once requested, a VM live migration controller instantiates a separate VM on the server destined for migration, using the same VM image/profile of the target VM. For the next step of synchronizing their states, there are several implementation variants but two most popular approaches are introduced: pre-copy and post-copy (Fig. 2.1). States that need to be synchronized between the source VM and the destination VM are CPU registers, memory and disk. Disk is easily shared in most modern DCs that provide network-attached storage (NAS) to host servers, and the volume of CPU registers is trivial compared to that of memory.

In pre-copy live migration [18], the source VM transmits its memory to replace the cold state of the destination VM with its hot state. The challenge is a decision when to stop the source VM and switch the operational authority to the destination VM, while minimizing the amount of service unavailability; this research defines it as

service downtime. Pre-copy divides the state transmission process into several iterative steps where memory pages that are dirtied since the previous step is transmitted. If a certain condition (e.g., 50 or fewer dirty pages to transfer) is met in an iteration, pre-copy pauses the source VM while sending the remaining (small) pages to the destination VM. Note that none of the VMs is available to outside during this period of freezing time. Once the destination VM has been prepared to resume the service application, the source VM hands over the operational authority and is permanently terminated. Pre-copy provides overall stability but also involves increase in total migration time and downtime for write-intensive applications who have high page dirty rate.

In contrast, post-copy [19] allows the destination VM to take over the operational authority after a minimum downtime to synchronize CPU registers only. Dirty pages are incrementally copied from the source VM when the corresponding memory region is accessed by the operational service application on the destination VM. Demand paging, active pushing or prepaging can be used to implementation. Post-copy provides shorter total migration time and downtime but runtime performance degradation is expected especially for read-intensive applications who have high page fault rate.

Despite the downtime during the process above is tens or hundreds of millisecond [20], cloud operating systems (e.g., OpenStack) usually involve additional networking operations in DC-level management. Such post-migration overheads include detaching/re-attaching virtual network interfaces (vNIC), packet redirection and IP address reconfiguration (i.e., DHCP involved), and may increase service downtime up to thousands of millisecond [21]. In Section 3.4.1, the service downtime occurred in a whole live migration process of a service is modeled as migration cost and related to the service's SLA availability.

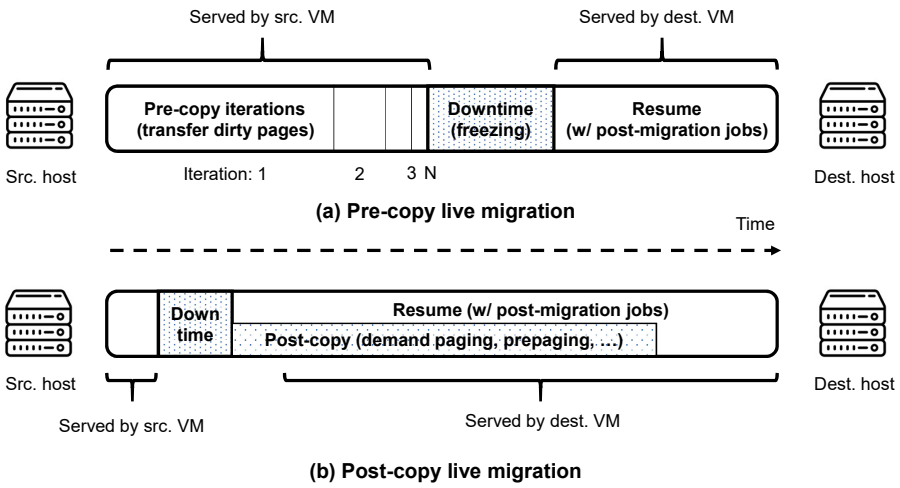


Figure 2.1: VM live migration (pre-copy vs. post-copy)

2.1.2 Service Placement in Edge Computing

Multi-level DCs planning and deployment are the most important solution to meet the low latency and large bandwidth service requirements in the 5G era [15]. Depending on providers and their operational regions, a general multi-level DC architecture for edge computing consists of 1 or 2 sites of center (cloud) DCs, a few of region (fog) DCs and many (on-demand) edge DCs in which a service can be deployed. In general, a cloud DC with the main resource pools (e.g., computing, networking and storage) provides the least placement cost but possibly worse QoS (e.g., bandwidth and latency). On the other hand, an edge DC that is likely composed of tens or hundreds of COTS servers promises improved service experience due to the proximity to users but higher placement cost or risk of availability. Therefore, operators need to have optimized policies that consider both cost and performance factors involved in service placement. In addition, different types of MEC services with various requirements increase the complexity of the service placement problem (Table 2.1).

Nowadays, with the advent of Industry 4.0, many enterprises try to lead their market by composing own 5G infrastructure to increase productivity (e.g., factory au-

Table 2.1: Use cases of URLLC services and their requirements [1]

Service types	**E2E latency**	**E2E availability**	**User experienced throughput**
AR/VR	10ms	99.999%	40-700Mbps
Cloud gaming	<7ms (uplink)	99.999%	1Gbps
Factory AGV control	5ms	99.9999%	100kbps (downlink) 3-8Mbps (uplink)
UAV (drone)	<100ms	99.999%	<128bps
Fault mgmt. in power generation	<30ms	99.9999%	1Mbps

tomation) and benefit from customized/dedicated services (e.g., network slicing). Such private 5G networks can be categorized into the following composition, depending on the location of 5G core functions [15][22] (Fig. 2.2):

- **5G radio access network (RAN) sharing:** the operator provides enterprises with the 5G core functionality through end-to-end slicing with the shared RAN. This mode is suitable to small and medium-sized enterprises who have limited CAPEX/OPEX for infrastructure (e.g., edge DCs).

- **Hybrid 5G:** in this mode, an enterprise deploys mission-critical services and a dedicated data plane function (i.e., UPF) inside own MEC infrastructure with a private 5G radio. The enterprise can also utilize live migration on demand (i.e., dynamically) to cache additional network functions/services from the operator's cloud. This mode that enables confidentiality and offload on enterprise data is suitable to medium and large-sized enterprises, when considering the expenditure.

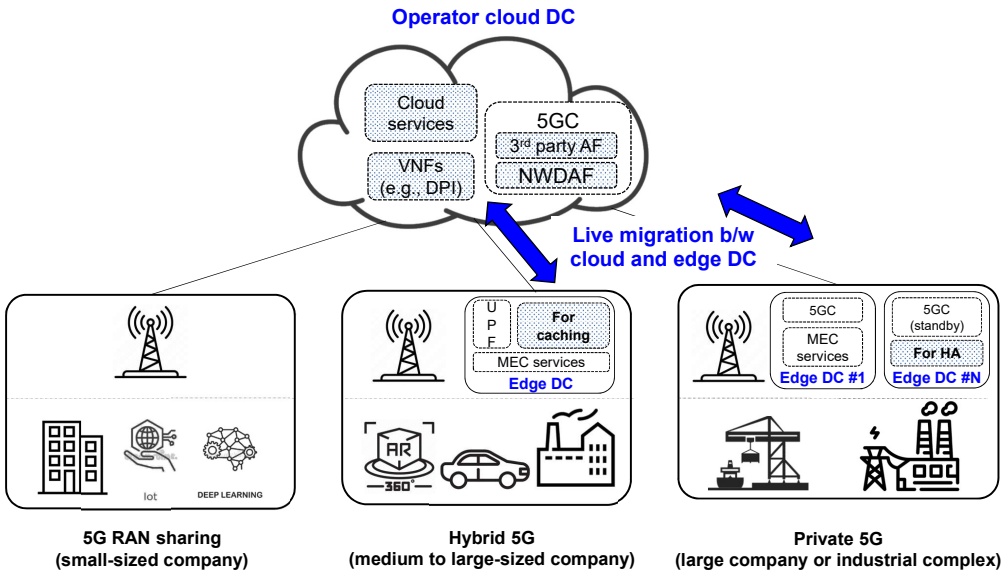


Figure 2.2: Private 5G network composition and use cases

- **Private (dedicated) 5G:** this type of private 5G is applicable to a super-large enterprise or industrial complex where the massive number of services are requested and served in the vicinity for ultra low latency. This mode can deploy multiple base stations, MEC hosts and scalable 5G cores (5GCs) with transit networks in site. Live migration of auxiliary functions can be utilized to achieve high availability (HA) in case of failure in standby servers.

VM live migration can be used for service placement in edge computing. For example, the performance of an edge computing service can be improved by dynamically migrating the service instance across multiple DCs over time so that its SLAs are preserved (e.g., evacuation from abnormal edge server). In this thesis, the optimization problem of dynamic service placement in edge computing is modeled to DRL, as a scalable methodology to approximate an optimal live migration policy for improvement in service latency and service availability.

2.1.3 Deep Reinforcement Learning

Briefly, DRL combines an RL algorithm with a deep neural network (DNN) that approximates an optimal policy to control the corresponding RL agent. The policy can become more robust to dynamics of the target environment as learning proceeds with various experiences of the agent. In Fig. 2.3, given a state s_t of Environment, Agent takes an action a_t according to Policy $\pi(s_t, a_t) = P(a_t|s_t)$ which is a probability distribution to take the action in the given state. Policy is represented as the corresponding DNN in the agent. Then, Environment generates a reward r_t as an estimated worth of the action and sends the reward to Agent with the next state s_{t+1}.

There are two general types of RL algorithms depending on learning approaches of agents. First one is value-based agent to which Q-learning [23] and DQN [24] belong, and they define Q-function (Equation 2.1). In short, a Q-function is the expected value of a cumulative reward R_t from the given state and action to the end of a trial (e.g., reaching to the final state in a maze game). Since Agent behaves in a non-deterministic way according to Policy in the given state and action, a cumulative reward is different for each trial.

$$Q(s_t, a_t) = E[R_t|s_t, a_t] = E[r_{t+1} + r_{t+2} + r_{t+3} + ...] = E[r_{t+1} + R_{t+1}] \quad (2.1)$$

Since the maximum cumulative reward in a trial is pursued, it is required to take a series of actions that can maximize Q-function (Equation 2.2). In practical implementation of a value-based algorithm, the probability of taking either the most valuable action (i.e, exploitation) or a random action (i.e., exploration) is controlled throughout the whole learning process. Exploration is needed for an agent to avoid being trapped on a local optimum particularly in the early stage of learning when Q-function is not sufficiently tuned [25]. γ is a discount factor for future rewards.

$$Q(s, a) \leftarrow r + \gamma \underset{a' \in A}{max} Q(s', a') \quad (2.2)$$

In each step of learning, a DQN agent updates its DNN (i.e., refining the policy) by blending the current estimate of the Q-function and a point-estimate of what the

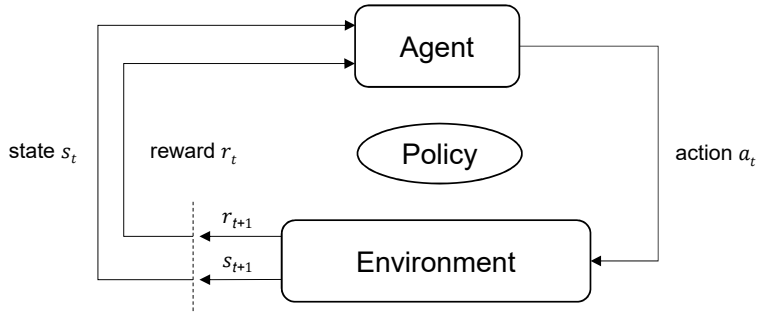


Figure 2.3: Framework of Reinforcement Learning

Q-function should be (Equation 2.3).

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \underset{a' \in A}{max} Q(s',a')) \tag{2.3}$$

Different with a value-based agent, a policy-based agent samples an action depending on Policy $P(a_t|s_t)$ itself. So, the policy-based agent learns how to adjust the probability function of taking actions according to the resulted cumulative reward R_t. In DRL, the probability function is represented as a DNN that is updated based on the following loss function.

$$L = -R_t \log P(a_t|s_t) \tag{2.4}$$

In Equation 2.4, R_t can be replaced with the Q-function (Equation 2.1), and this leads to an Actor-Critic algorithm that is a hybrid of value-based agent and policy-based agent. Therefore, an Actor-Critic algorithm uses two DNNs; a critic network to update the Q-function (i.e., expected R_t) and another actor network whose loss function is Equation 2.4 to update the probability function (i.e., Policy).

This research proposes two dynamic service placement algorithms that utilize DQN and Temporal Difference Actor-Critic (TD A-C) [26] respectively. The main reason of using DRL is to construct a mechanism that estimates fitness of a service instance to a host server in a robust manner to environment dynamics. Computed fitness values are used for criteria to decision making in service placement. Different learning

approaches of DQN and TD A-C lead to differences in selecting actions (Section 3.2.2) and learning performance of agents (Section 5.3).

2.2 Related Work

2.2.1 Optimizing Live Migration Decision

Yi et al. [27] proposed a load balancing method that migrates Virtualized Network Functions (VNFs) that are involved in overloaded servers or bottleneck links to underloaded servers, considering the cost of bandwidth consumption throughout hops involved in the migration. The authors simulated VNFs in mesh networks with tens of nodes, and observed that the number Service Function Chains (SFCs), nodes and links in overloaded state are reduced by the proposed migration-based load balancing. Similarly, Rui et al. [28] proposed a VNF live migration method that aims at balancing loads between servers. Using the Petri net-based model for SFC availability in their algorithm, the authors argued that service availability must be considered in migrating service instances.

Regarding power management, Basu et al. [29] proposed a VM migration algorithm that minimizes the number of active servers whose electricity consumption is proportional to the CPU utilization, while preserving SLAs of a service running in VM. The authors modeled a cost function that combines two trade-off factors of energy consumption and SLA violation (i.e., the overall performance of services degrades with a fewer number of host machines), and solved the optimization problem using RL with a reward function that evaluates possible migration actions in terms of the cost function. The authors simulated a single DC environment with different patterns of workloads, and showed that the proposed RL-based method can converge on an optimal migration policy that incurs less cost than heuristic-based algorithms, providing better scalability with a fewer number of migrations.

VM live migration is also one traditional fault mitigation method in network management, and currently draws attention in combination with the capability of fu-

ture anomaly prediction using machine learning. Ibrahimpasic et al. [30] proposed a decision-making method to migrate an (network) outage-risky VNF in an edge DC to a cloud DC. The migration is executed only if the reconfiguration cost of the new service path to the cloud DC is lower than the operator's loss from the QoS degradation due to retaining the operation in the original risky edge DC. In this work, deep learning is used to predict the pattern of user request arrival, based on the current location and mobility of users, in order to estimate the outage probability of the edge DC in near future. Regarding mitigation of server failure via VM live migration, Lin et al. [11] used a deep learning model that can predict the probability of server failure by learning both temporal data (e.g., memory utilization) and spatial data (e.g., server location inside rack) that are collected at a real production cloud DC for one month. The model then ranks top-k servers in order of predicted failure scores, and the result shows a tendency that servers who had previous failure history are more likely to be failed again in future. The authors reported that the overall VM failure rate can be reduced by 30% using the deep learning-based failure prediction model (i.e, failure scores) in allocating and migrating VMs to servers.

2.2.2 DRL-based Dynamic Service Placement

As one of branches that dynamic service placement addresses, service migration is the concept of moving a service instance to the MEC server closest to the user while considering migration cost; it is also called Follow-Me Cloud (FMC) [31]. The importance of service migration is increased with the advent of MEC and V2X (Vehicle-to-Everything), but it is still challenging to predict user (vehicle) mobility which allows to reduce the migration cost (e.g., total migration time) by identifying possible next MEC servers and migrating in advance. Zhang et al. [32] argued that service migration decision can be optimized through DRL approach even if there is no prior knowledge on user's mobility pattern. In the proposed DRL model, a state is defined as the distance between user and host server, and an action is either to migrate

the service instance to the MEC host closest to the user or not to migrate (i.e., keep the service location). The selected migration action is evaluated in terms of the resulted QoS benefit and migration cost which is bandwidth consumption throughout link hops involved in transferring the instance. The authors simulated a random walk user in a grid topology with tens of MEC servers and showed that their DQN-based service migration algorithm outperforms existing dynamic programming-based algorithms with larger total reward and better scalability. Park et al. [33] proposed a similar DQN algorithm but additional energy consumption of servers involved in a migration action is modeled as migration cost. In the simulation where a random walk user is placed in a grid topology with hundreds of servers, and the FMC controller (i.e., agent) periodically identifies an MEC server whose migration benefit results in the maximum total reward, the proposed DQN-based migration policy is reported to outperform policies based on Q-learning and heuristics.

Regarding VM consolidation which mainly packs (places) VMs together into a subset of all servers for performance improvement (e.g., reduced power consumption) while preserving their minimum requirements (i.e., SLAs), Zeng et al. [34] proposed a DRL-based VM live migration policy adaptive to the energy efficiency in a cloud DC. In the proposed DRL model, workloads of all host servers in a cloud DC are predicted by a recurrent neural network and are considered as state space. Then VMs either in overloaded or underloaded servers are migrated to normal servers, according to placement decisions by DQN that learns how to evaluate the suitability of a VM to a host server, in terms of the cost of energy consumption and SLA violation. The simulation results on a single cloud DC with 200 host servers showed that the proposed scalable VM consolidation algorithm based on DRL-based migration decisions can achieve 50% decrease in total energy consumption and SLA violations, compared to heuristic-based algorithms.

In a point that VM live migration can be used in dynamic service placement in edge computing, two related studies based on DRL are reviewed. Dalgkitsis et al. [35] proposed a dynamic resource-aware VNF placement algorithm for MEC services (e.g.,

uRLLC). The placement decision on a target VNF is made based on the state space: a specification of the VNF (e.g., required CPU, memory and SLA latency), resource usage of all MEC servers and available bandwidth of all links. The worth of an action to place the VNF on the selected MEC server is evaluated by the resulted service latency (i.e., reward). The authors showed in a Mininet-emulated topology with a single cloud DC and 5 edge DCs (servers) that the proposed Deep Deterministic Policy Gradient (DDPG) algorithm can achieve 23% and 31% reduction in the number of SLA violations and VNF rejections respectively, compared to heuristic algorithms that places VNFs in non-overloaded MEC servers or in the cloud DC only. Different with single agent DRL algorithms used in the previously reviewed studies, Dai et al. [36] proposed a multi-agent DRL model where each agent creates migration decisions on VMs running in the assigned MEC server only, with less monitoring overhead and shorter learning time by allowing each agent to treat a smaller state space. In the proposed Actor-Critic algorithm design, the state space of each agent includes resources and a sequence of profiles of VMs that are hosted in its dedicated server. The action space of each agent is a set of MEC servers to which the corresponding VM is migrated while preserving its SLA latency. The worth of a migration action is evaluated by the sum of per-hop bandwidths each required in sending a VM image file from source to destination MEC server (i.e., fewer hops are preferred). The evaluation of a simulation topology with 20 MEC servers showed the proposed multi-agent DRL algorithm can produce a larger cumulative (total) reward than heuristics and the single agent DRL, with lower number of migrations (e.g., less bandwidth consumption) and less learning time.

Finally, limitations of the existing work in DRL-based dynamic service placement and how this research is different and improved are summarized (Table 2.2).

- The existing work [32][33] aims at optimizing a migration policy for a single target service and its user. **In this research, concurrent migration of multiple services and related orchestration aspects are considered (Section 3.2.2).**

- The existing work [34] aims at optimizing a migration policy used in a single DC. **In this research, migration policy in a multi-level DC network (i.e., multiple DCs and their internal servers) is considered to support the edge computing environment of 5G services (Section 4.1).**

- The existing work [35][36] does not consider SLA availability of each service and failure of edge (MEC) servers [37]. **In this research, the cost of live migration service downtime is modeled (Section 3.4.1) and related to SLA availability of any service (Section 3.4.2) so that total number of migrations is reduced; i.e., a service instance can be migrated within a limited number for its duration. Server failure scenario is also considered so that the proposed migration algorithms can evacuate risky services in advance or perform fault mitigation by load balancing [11][38].**

Table 2.2: Summary of related work in DRL-based dynamic service placement via live migration

Reference	DRL Algorithm	State	Action	Objective (reward)	Service	SLA	Topology
[32]	DQN	Distance b/w user and service	Migrate service to closest server	Min. BW consump.	Single	X	Distributed MEC servers
[33]	DQN	Distance b/w user and service	Migrate service to closest server	Min. BW consump., Min. energy consump.	Single	X	Distributed MEC servers
[34]	DQN	Predicted server workload	Migrate service in abnormal server to normal server	Min. energy consump. Min. SLA violation fee	Multiple	Availability	Single DC
[35]	DDPG	Service specification, Server resource load, Link BW	Migrate service to server with highest fitness	Min. service latency	Multiple	Latency, Throughput	Multi-level DCs (cloud, edges)
[36]	Multi-agent Actor-Critic	Service specification, Server resource load	Migrate service to server with highest fitness	Min. BW consump.	Multiple	Latency	Distributed MEC servers
Proposed	Multi-agent DQN/AC	Service specification, DC resource load	Migrate service to DC with highest fitness	Min. service latency, Max. service availability	Multiple	Latency, Availability	Multi-level DCs (cloud, edges)

III. Design

In this chapter, edge computing scenario is introduced where dynamic service placement via live migration is effectively used for optimization in service performance and/or resource efficiency. Afterwards, a DRL model that represents the dynamic service placement problem in terms of RL environment, agent, state, action and reward is explained, and multi-agent DRL algorithms that generate migration policy optimized for service latency and service availability are proposed based on the DRL model. Finally, a reward model that considers the availability of edge machines (i.e., MEC host) and a cost model that relates live migration service downtime with SLA availability of any service are proposed.

3.1 Dynamic Service Placement in Edge Computing

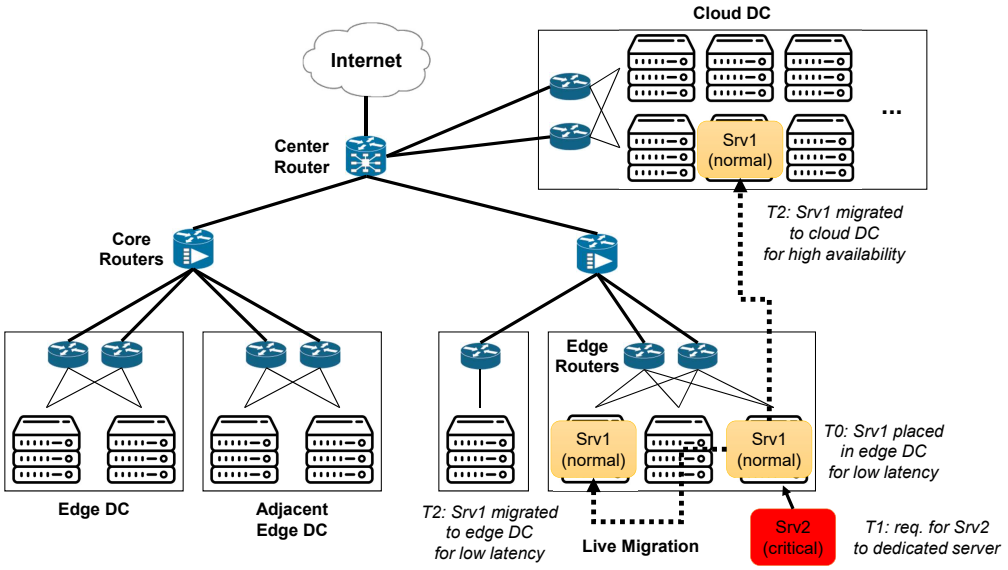
In edge computing of multiple DC networks, overall service performance and resource efficiency can be improved through techniques that supports dynamic service placement: auto-scaling and live migration. These two operations are essential for network administrators in managing DC infrastructure and orchestrating multiple service instances. Their implementation is mature and already included in many cloud computing platforms (e.g., OpenStack) but their application as a persistent strategy (not one-time usage) is challenging due to the complexity of the management domain in edge computing; deployment cost of (MEC) services with different requirements (e.g., user location, resource demands and SLAs) is highly different across multiple DCs where resources are hierarchically distributed. This research focuses on using live migration to realize dynamic service placement, and uses DRL to address the management complexity by enabling the target migration strategy to be getting aware of the edge computing dynamics. A DRL agent that learns the given environment can

generate optimized migration decisions; **when** is the best to migrate **which** service instances to **where**. Some service placement scenarios are introduced to analyze how migration policy can behave more efficiently.

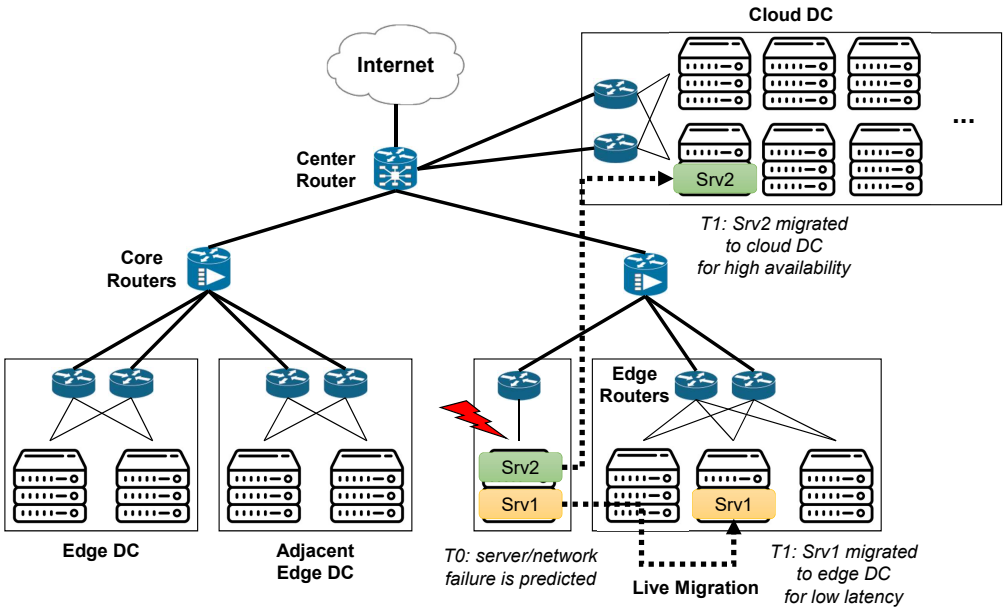
In Fig. 3.1(a), by time 0 (T0), Service1 (Srv1) has been initially deployed and run in the edge DC. Let us assume that Service1 is a normal cloud service (e.g., Software as a Service) with non-critical SLA or none. So obviously it would be not SLA violation if Service1 was placed in the cloud DC, but the operator had decided to place it in the edge DC for user satisfaction (e.g., lower latency) or maintenance reason (e.g., load balancing). Then, a request for Service2 (Srv2) is expected at time 1 (T1) with the requirement of critical SLAs (e.g., ultra-low latency or hardware dependency). Before the actual request to Service2 arrives, the operator should reserve the assigned server (e.g., closest MEC host), otherwise there will be delay in deployment/provisioning at least or, in worst, SLA violation of Service 2 (i.e., service rejection). To achieve effective host/resource overbooking [39][40] that satisfies both Service 1 and Service 2, the operator can perform live migration of Service1 to another place without interrupting service availability, but the problem still requires optimization in when and where to migrate:

- One policy is to migrate Srv1 to an adjacent edge DC just before T1, considering intra-DC memory migration.
- Another policy is to migrate Srv1 to the cloud DC at T0, considering inter-DC block (disk) migration.

Similarly, there are migration policy options for fault mitigation in a proactive migration scenario (Fig. 3.1(b)). One operator prefers to evacuate outage-risky service instances to a high availability zone at the cost of increasing latency, or another one prefers to migrate them according to service types at the cost of losing availability. To this end, early detection/prediction of failures in DC is recently studied using Artificial Intelligence (AI) technique [11][41][42], but it is still challenging (to reproduce) and out-of-scope in this research. However, in DRL via iterative learning



(a) Scenario 1: migration for service latency vs. service availability



(b) Scenario 2: mgiration for fault mitigation

Figure 3.1: Dynamic Service Placement via Live Migration

process on the given service placement scenario, a DRL agent can experience some anomaly symptoms (e.g., overloaded edge servers) and possibly encounter the failure cases throughout its exploratory actions. If the penalty of encountering failure states is indicated in the reward model [43], the DRL agent can learn how to behave in a way to avoid taking actions that lead to the penalty state. In the early stage of DRL with less confidence on the worth of possible actions, the behavior (policy) would be conservative or radical but finally be aware of the optimal actions that lead to the maximum cumulative reward (Section 2.1.3). In the simulation-based evaluation, this research conditionally injects server failure events that incur the termination of related running services, and proves that the proposed DRL algorithms can generate placement policy to suppress service failures by proactively migrating risky services.

3.2 DRL Model for Live Migration

3.2.1 Overall Design

In general, a migration decision can be optimized depending on that **which** service instances should be migrated to **where** and **when**. As to consider concurrent migration of multiple services without conflict, a migration decision should be made on each service every decision time. Note that a service can stay in the current host[1] or be migrated according to a mechanism that evaluates the fitness of the service to a host based on a desirable criteria. The interval of each decision time also can effect the granularity of the decision-making. In abstract, the proposed DRL-based dynamic service placement algorithms address the following migration aspects:

- **Which**: a policy generates migration decisions on individual services which are currently running on hosts. A service can stay in the existing host or migrate to another machine.

- **Where**: for each service, DNN of a DRL agent computes the score of fitness

[1]Host, server and machine have the same meaning and are used interchangeably in this thesis.

to each machine based on the current states of the machine, the service specification (both explicit) and the rewards of previous migration actions which are implicitly reflected in the current DNN parameters. As having a higher fitness score, a machine is likely to be selected as the new host of the service (i.e., fitness score of a server that is not compliant with the service's SLA is 0).

- **When**: migration decisions on running services are made by the DRL agent every decision interval. A shorter interval can improve the performance by increasing the optimization granularity but also increase monitoring and learning burden of the agent. This issue is observed in the implementation of the first single-agent DRL model (Section 3.2.1) and is solved in the next multi-agent DRL model that reduces the state space from server-level to DC-level (Section 3.2.2).

To intuitively introduce a basic DRL model for dynamic service placement and elaborate possible design choices, a single-agent DRL model is designed first (Fig. 3.2). The basic components of an RL model are defined: state, action, reward, agent and environment. First, in the single-agent DRL model, a state is represented as a service-machine placement map with *Mapping_Features* of the corresponding service-machine pairs. *Mapping_Features* consists of i) current remaining resources in the machine, ii) demanding resources of the service to place, iii) the service's SLA latency and SLA availability and iv) *Path_Cost* between the locations of the service user (i.e., an edge DC in proximity) and the machine (i.e., at a cloud DC or edge DC). As a subset of all possible observations from the environment, the defined state space is a design choice to represent essential information that is needed for service deployment in DCs in practice [44][10]. For each running service and each machine, a *Mapping_Features* is generated and concatenated to construct a mini-batch (i.e., state) as input for the DNN in Central Migration Agent.

In Central Migration Agent, the mini-batch is fed into the DNN where the input dimension is same with the length of *Mapping_Features*. Parameters of the DNN

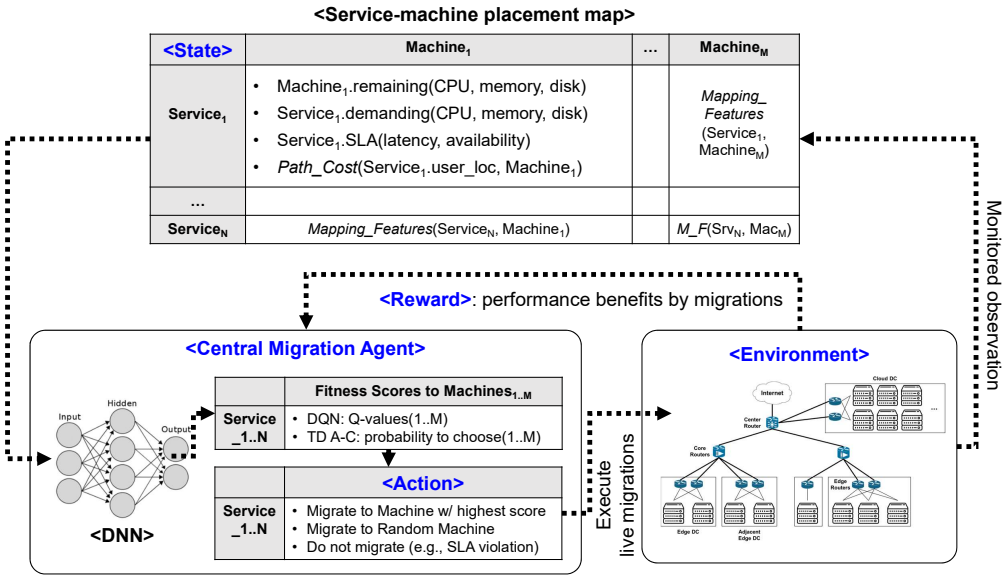


Figure 3.2: Architecture of the single-agent DRL model for dynamic service placement

model are tuned so that its output can represent a fitness score of the service-machine pair according to the worth of placing the service on the machine (i.e., reward). In other words, as learning proceeds, a higher fitness score is assigned to a service-machine pair that can maximize the total reward accumulated for each migration. In this research, for the evaluation of fitness scores and the training of DNN, two existing DRL algorithms of DQN and Temporal Difference Actor-Critic (TD A-C) are utilized (Section 2.1.3). In the proposed DRL model, these two algorithms have different mechanisms in converting fitness scores (i.e., DNN outputs) into the corresponding migration actions. In the given state, DQN outputs the fitness score of a service-machine pair as a Q-value which estimates the worth of placing (migrating) the service in the machine. Whereas, TD A-C outputs a probability that indicates the likelihood of the agent (i.e., policy) to select the placement action in the given state. Both of their agents interact with the environment via selected migration actions and guide (reinforce) their DNN

models to output more accurate fitness scores from the resulted rewards as learning proceed. Note that a sub-action[2] in this intermediate step is to select a machine (ID) to which a service instance migrate.

Then for each service, the final sub-action can be of i) migrating to the machine with the highest fitness score (exploitation in DQN), ii) migrating to the randomly selected machine (exploration in DQN) or iii) staying in the current host (i.e., do not migrate). The exploration behavior is needed for the DQN agent to avoid converging on a local optimum by searching unexperienced state space that might result in higher cumulative rewards than previously searched state space [45]. The matter of selecting a proportion between exploitation and exploration is an implementation choice. In this research, decaying epsilon [46] is used to choose exploitation with 90% and exploration with 10% at the beginning and reduce the probability of exploration by 1% as learning proceeds. Note that exploration in the TD A-C agent depends on the current probability distribution of taking actions (i.e., DNN). The third action type arises when there are no available machines to accommodate the service instance, or migration to the selected machine is expected to violate SLA latency and/or SLA availability of the service (Section 3.4.2). Considering that a service instance which is in the middle of "live" migration due to previous migration decisions keeps available to its user and still seen as a running service to Central Migration Agent, a migration decision on such services should be the third action type to avoid conflict.

Lastly, the final action set (i.e., finalized migration actions) is applied to the edge computing environment as each of migration requests to the migration subsystem (e.g., OpenStack Neutron [47]). A migration request specifies the system-level identifiers of the target service instance (VM or container), current host (source) and destination host machine. The environment sends a reward as a feedback signal to the agent so that it can update fitness scores and better estimate the worth of the applied migration actions in future decision-making. According to the objective function defined in the

[2]Since concurrent migration of multiple services is considered in this research, a set of sub-actions corresponds to an action in a single service migration model [32][33].

problem statement (Equation 1.1), the reward is modeled as the sum of the average service latency benefit and the average service availability benefit that can be derived by comparing before/after applying the migration actions to the environment. Details on the proposed reward model is provided in Section 3.3. The environment also generates monitored observation (i.e., next state) for the next loop.

3.2.2 Multi-agent DRL Model

During the verification of the described single agent model in simulation using thousands of real user service requests in the data set (Section 4.2), some performance issues are experienced in the learning process; gradient exploding [48][49] that leads to nondeterministic termination of the used deep learning engine (PyTorch) happens when a high learning rate is used to train the DNN model. Empirical analysis turns out that the state size[3] in the single agent model is too large for the DNN model to tune parameters accordingly when a short migration decision interval (e.g., 10s) is used. Since it was preferred to address this issue without changing the structure of the DNN architecture (e.g., number of layers) and spec of the machine to run the algorithm, reducing the state size was aimed first for the scalability to more complex dynamic service placement scenario. As a result, the design choice was changed to the multi-agent DRL model in Fig. 3.3.

The multi-agent DRL model for dynamic service placement is motivated by the related work [36] which optimizes migration policy in the MEC environment using a multi-agent DRL model. In short, each MEC host server acts as an individual DRL agent to generate migration decisions only for local services running in the host. The authors argue that their multi-agent model can alleviate the communication bottleneck occurred in a centralized single agent model and is suitable for each MEC server (i.e., agent) to control its local domain, considering the inherent distributed computing environment of MEC [50]. The differences in this thesis are as follows:

[3]The number of all services (N) * the number of all machines (M) * the length of *Mapping_Features*

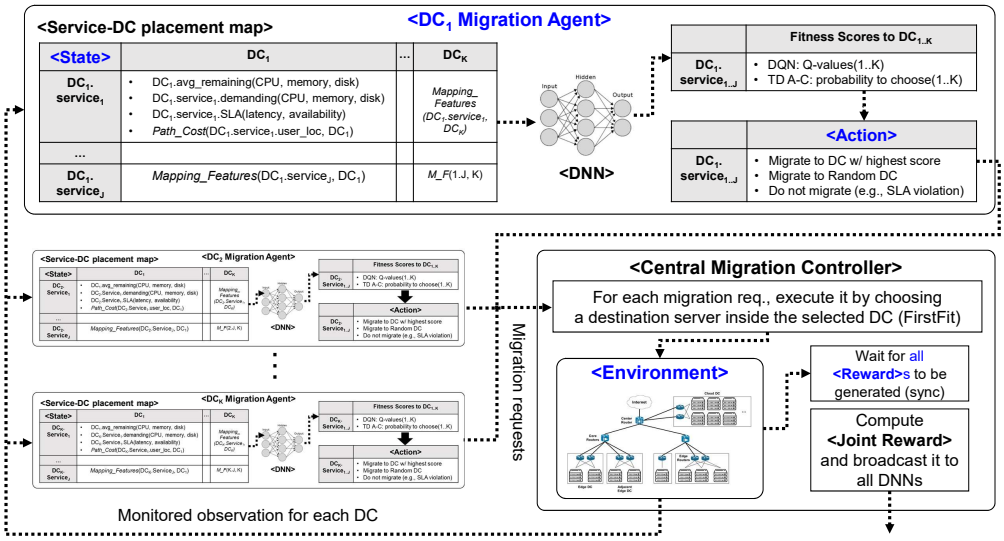


Figure 3.3: Architecture of the proposed multi-agent DRL model for dynamic service placement

- Each agent is assigned to an MEC server in the related work, while each agent is assigned to one DC in this research (i.e., one-to-one mapping), to achieve dynamic service placement across multiple edge computing DCs and the reduction in state space.

- The main objective of the related work is to ensure SLA latency of each service while minimizing migration cost of bandwidth consumption. Whereas, the main objective of this research is to improve overall service latency and service availability without SLA violation (i.e., only migration that does not violate service SLAs is permitted).

Regarding the new multi-agent model, its components are explained along with the comparison to those of the previous single-agent model. Considering that edge computing topology consists of multiple DCs across network core and edges (Fig. 3.1), the role of Central Migration Agent in the single-agent model is delegated to each of

DC Migration Agents whose control domain is its assigned DC. The main purpose of the one-to-one mapping between DC and agent is to impose smaller state and action space on each agent than Central Migration Agent (i.e., single agent), so the number of agents and their assigned DCs is a design choice. For each agent dedicated to a DC (cloud or edge), a state is changed to a service-DC placement map only for local services. Since the placement unit by the agent's decision is changed to DC (i.e., selecting the best DC to place, not server/machine), *Mapping_Features* is almost same with the previous model but machine-level features are replaced with DC-level features (e.g., resource utilization averaged over machines within the DC). As a result, the state size[4] is reduced as well as action space to allow each agent to handle a smaller problem of inter-DC service placement (i.e., divide and conquer). The downside of the reduction is the loss of granularity in selecting destination servers, which can lead to performance degradation after migration, since DC-level average values are not fully representative to the status of each machine. Verification shows that the proposed multi-agent DRL algorithms compensate the downside with learning more experiences in the given time and can outperform the single agent algorithm (Section 5.3).

For each of local services and DCs, *Mapping_Features* is generated and concatenated to construct a mini-batch for the corresponding DNN model in the agent (Fig. 3.4). Note that individual DNNs in the proposed multi-agent DRL model are independently trained according to the given configuration of local services (i.e., state) in the corresponding DC. The configurations can be different during the early stage of learning but should converge on a certain state that provides the joint optimality of all agents after sufficiently learned. In the multi-agent DRL model, an output of the DNN model represents a local service's fitness scores to all DCs. DQN and TD A-C algorithms with different learning approaches are still used to estimate the worth of migrating the service to the selected DC in terms of reward. Sub-action for each local service is i) to migrate the service to the DC with the highest fitness score (exploitation), ii) to randomly selected DC (exploration) or iii) not to migrate. The third

[4]The number of DC-local services (J) * the number of all DCs (K) * the length of *Mapping_Features*

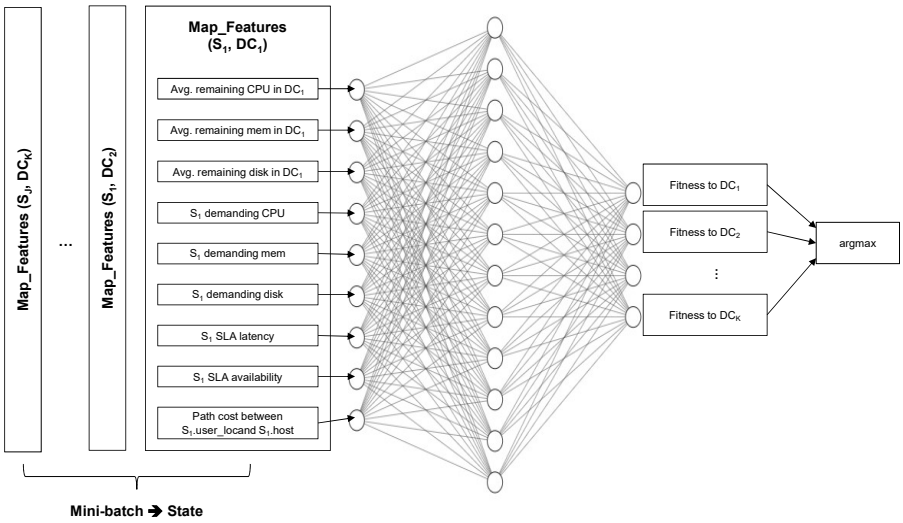


Figure 3.4: DNN model of the multi-agent DQN algorithm that computes fitness scores between service and DC

action type occurs when migrating to the selected DC leads to SLA violation of service latency or service availability. The action space of exploration in the multi-agent model is also reduced from random selection of any machine to any DC. Since each agent can control its local services only and check their migration status (e.g., ongoing live migration), there are no conflicting decisions on the same service among different agents. The final action set of all local services is transformed to a set of migration requests and sent to Central Migration Controller where the migration subsystem is operated.

Before applying an migration request to the environment, Central Migration Controller has a role of choosing the final destination host to newly place the service instance among servers in "the selected DC". Since the agent who requests the migration has already confirmed that migrating the service to the DC is free from SLA violation of both service latency and service availability, Central Migration Controller uses a simple first-fit algorithm to find out an internal host machine available to accommo-

date the target service instance. Similar to the single-agent model, the worth of a migration action is evaluated by the sum of the service latency benefit and the service availability benefit before/after migrating the service. However, for each decision tick, Central Migration Controller generates the joint reward that averages rewards of all the agents, so that the joint reward can represent the whole system benefit from the applied migration actions, not individual agent's benefit (Section 3.3.2).

3.3 Reward Model

Reward in RL is a critical criterion to guide the agent (its DNN model) to behave in the environment world. In this section, a detailed explanation on the reward model of the proposed dynamic service placement algorithms is provided. In the proposed multi-agent DRL model, the reward of $Agent_k$ is the sum of service latency benefit B_L and service availability benefit B_A where the benefits are the differences between before and after the migration action in the given state of $Agent_k$ (Equation 3.1). Service latency benefit can be negative if the migration places a service from its user location farther than before (e.g., from edge to cloud DC). Service availability also can be negative if migrating a service to a server with lower availability (e.g., from cloud to edge DC). In the proposed reward model, availability of any service depends on the probability of failure in its host server. Despite each of the two benefits is normalized within -1 to 1 and then the result is calibrated to a non-linear function (Section 3.3.1), their probability distributions are inherently different so coefficients of w_L and w_A are used to balance their importance.

$$Reward_k = w_L B_L + w_A B_A \tag{3.1}$$

For migration requests by $Agent_k$, the service latency benefit B_L and the service availability benefit B_A are averaged over J services from the set of all local services S_k running in the corresponding DC, where each service s_i migrates from machine m_i^{old} to machine m_i^{new} (Equation 3.2 and Equation 3.3 respectively). The service $Latency$

is round-trip time (RTT) between user and host machine of the service. Whereas, the service $Availability$ depends on the availability of the host machine. A service instance in cloud computing can be in a failure state due to software errors [51] or hardware errors [52]. In this thesis, the latter is treated only but measuring availability or failure probability of server/machine is still a challenging issue, with a need for field data about failure history which is usually confidential and unavailable to public. Therefore, referring to articles mentioning that hard disk (HDD) is one major root cause of server failure in cloud DC [53][52], a simplified server failure model, $P_{failure}$, is defined to relate the probability of server failure with abnormal high disk utilization of the server during the last N monitoring intervals in a row (Equation 3.4). The rationale of considering disk (i.e., storage device) in a server failure model also closely relates to the fact that VM live migration requires disk transmission through WAN between DCs in different NAS domains [54].

$$B_L(S_k) = \frac{1}{J}\sum_{i=0}^{J}\frac{Latency(s_i, m_i^{old}) - Latency(s_i, m_i^{new})}{Latency(s_i, m_i^{old})} \tag{3.2}$$

$$\text{where } s_i \in S_k \text{ and } Latency(s, m) = RTT(s.user\_loc, m.loc)$$

$$B_A(S_k) = \frac{1}{J}\sum_{i=0}^{J}\frac{Availability(s_i, m_i^{new}) - Availability(s_i, m_i^{old})}{Availability(s_i, m_i^{old})} \tag{3.3}$$

$$\text{where } s_i \in S_k \text{ and } Availability(s, m) = 1 - P_{failure}(s, m)$$

$$P_{failure}(s, m) = \begin{cases} 0, \begin{cases} \text{if } m \text{ does not host } s \\ \text{or } m.disk\_util_{t-i} < 0.95, \text{ for } 0 \leq i < N \end{cases} \\ \frac{1}{N}\sum_{i=0}^{N} m.disk\_util_{t-i}, \text{otherwise.} \end{cases} \tag{3.4}$$

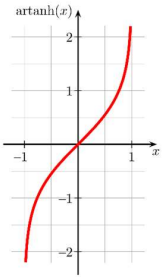
$$\text{where } N \text{ is the length of monitoring windows}$$

In the implementation of the proposed DRL simulator (Section 4.1), a monitoring component periodically updates all the states of machines according to the current service placement, while injecting server failure events when the disk overutilization condition is met and preventing failed servers from being considered in service placement until the end of the one simulation cycle (i.e, episode). All services running in failed machines are immediately interrupted and queued to be re-deployed in other SLA-compliant machines to finish the remaining service duration. Consequently, occurrences of server failure would likely deteriorate the following service placement decisions due to increased provisioning delays of future service requests with fewer available machines (i.e, high resource contention). Therefore, a service placement algorithm should prevent servers from being failed by balancing the usage of resources (e.g, disk utilization) among servers. To this end, the proposed DRL agents should learn how to balance between QoS and load distribution. Despite the importance of considering server availability in deploying highly reliable services (e.g., URLLC) in MEC environment, only the related work [37] considers the service availability model with different probability of server failure in cloud or edge DC.

3.3.1 Reward Calibration

In the proposed reward model, the units of two different terms of service latency and service availability are millisecond and probability (0 1) respectively, while leaving the normalization issue in calculating the reward by their sum. In spite of their evaluation as benefit before/after migration so as to treat them in the common unit of percentage (%), their statistics (e.g., min/max or average) is still different, which encourages to use empirically tuned coefficients of w_L and w_A in the reward computation to compensate their numerical differences. This reward coordination by coefficient is commonly used in the existing work with more than two objectives of different scale (e.g., cost vs. performance) [9][33].

To accelerate the learning performance of RL model by reward calibration [55][56],



$$\operatorname{artanh} x = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$$

Figure 3.5: Reward calibration by projection to the inverse hyperbolic tangent function

each of service latency benefit and service availability benefit is clipped to the range of $[-1, 1]$ and the normalized values are projected into the inverse hyperbolic tangent function [57] (Fig. 3.5). The projection to the non-linear function exponentially scales the benefit values as being closer to the both extremes in the x-axis; 1 when the migration leads to the most service latency/availability benefit or -1 when the least benefit. In other words, the purpose of this calibration is to highlight what is the best action or the worst action in the given state, so that the DRL agent can effectively learn such obvious cases than the original linear reward function. The calibrated reward $Reward_k^{'}$ of $Agent_k$ is given by Equation 3.5.

$$Reward_k^{'} = \frac{1}{2} w_L \ln\left(\frac{1 + B_L^{'}(S_k)}{1 - B_L^{'}(S_k)}\right) + \frac{1}{2} w_A \ln\left(\frac{1 + B_A^{'}(S_k)}{1 - B_A^{'}(S_k)}\right) \tag{3.5}$$

where both $B_L^{'}$ and $B_A^{'}$ are clamped to $[-1, 1]$

3.3.2 Joint Reward

In a general multi-agent DRL model, each of agents tries to train its own policy to generate the maximum cumulative reward by updating its DNN with rewards from iterative experiences. Therefore, If there is no mechanism to orchestrate them in a global view, each agent is ignorant of other agents (and their policies) and behaves in a greedy

manner. This means that DC_k Migration Agent (Fig. 3.3) may generate myopic decisions which can deteriorate the entire system performance; for instance, evacuating all services the agent controls to an edge DC that is currently underutilized but occupied with rapid demand soon, since the agent only know that the corresponding migration decisions will increase its own service availability benefit. In RL, this myopic behavior of an agent can be corrected by experiencing cases where the total reward is increased with different actions as learning proceeds (i.e., increase in learning time). However, in multi-agent DRL, the greedy of individual agents can impede chances to reach the optimality within the given learning time or lead to local optimum.

To this end, a mechanism is added to Central Migration Controller that waits for individual rewards of agents to join and computes the single $Reward_{joint}$ for all agents in common. To enable $Reward_{joint}$ to be a representative reward for the whole system benefit during the current migration decision tick, it is defined as the average over reward values of all agents. Then Central Migration Agent broadcasts $Reward_{joint}$ to all agents, allowing each agent to update its DNN with the joint reward and correct its policy in accordance with the global benefit. The final reward of $Action_k$ of $Agent_k$ in the given $State_k$ is computed by Equation 3.6.

$$Reward_{joint} = \frac{1}{K} \sum_{k=1}^{K} Reward_k^{'} \tag{3.6}$$

where K is the number of agents and their assigned DCs

3.4 Migration Cost Model

VM live migration offers the capability of dynamic service placement in edge computing, but it is an operation with cost. Migration cost is considered in general as bandwidth consumption [32][36] and/or energy consumption [33][34]. For the former, an amount of network bandwidth is proportionally consumed by the size of VM image file in case of block (disk) migration or by the page dirty rate in case of memory

migration, since the implementation of VM live migration must involve transmission of VM's states from source to destination host server. Regarding energy consumption involved in live migration, Hu et al. [58] presents an energy consumption model that combines bandwidth, content size, noise power and transmission rate in dynamic service migration scenario. Therefore, defining a refined migration cost model is vital in optimizing live migration policy for dynamic service placement in real environment; if migration cost is inadequate or not considered, the occurrence of a tremendous number of migration can offset the resulting profit and deteriorate QoS implicitly.

3.4.1 VM Live Migration Service Downtime

Service downtime due to VM live migration has not been considered as a migration cost in existing work. To the extension to the discussion about service downtime in Section 2.1.1, VM live migration service downtime is composed of two different downtime periods during the whole live migration process. First type of downtime occurs during the switchover from the source VM to the destination VM to ensure their complete synchronization by transmitting the remaining dirtied memory pages of the source VM (Fig. 2.1). Since this type of downtime is dependent both on the page dirty rate of the VM, i.e., how fast the service application modifies the VM's memory pages, and available network bandwidth for the transmission, the length of downtime can be exponential if the page dirty rate (pages/s or KB/s) exceeds the available bandwidth (Mbps) [21]. Post-copy live migration emerges to avoid prolonged switchover downtime possible in pre-copy, but the loss of network connectivity after the fast switchover in post-copy requires rebooting the VM to restart the demand paging process. Therefore, in modern VM live solution, a hybrid approach is preferred that uses pre-copy as default and post-copy as fallback [47].

The second type of VM live migration downtime is post-migration overheads that include (un)binding a virtual network interface, reallocation of IP address (i.e., DHCP) and other networking-related configurations to redirect incoming packets to

the destination VM [21]. To model the expected downtime of the current live migration decision as migration cost, Equation 3.7 is defined for service s (i.e., running in VM) and migration destination machine m_{dst}.

$$downtime(s, m_{dst}) = \frac{s.memory}{s.duration} * \frac{1}{path(s.host, m_{dst}).bw} + const\_postmig \tag{3.7}$$

Since measuring page dirty rate of each VM requires additional monitoring overheads in practice and the reference data set of real DC user requests (Section 4.2) for evaluation does not provide such granularity, the page dirty rate is estimated as the memory usage intensity of the target VM for its service duration, referring to the argument that memory-intensive workloads are main optimization targets in pre-copy live migration [59][18]. Since the cold state of the destination VM is synchronized by iterative transmission of dirty pages from the operational source VM in pre-copy live migration (Section 2.1.1), available bandwidth of the path between current host machine $s.host$ and destination machine m_{dst} is also considered. In previous work[5], downtime during post-migration operations is repeatedly measured in the OpenStack testbed using memory live migration (i.e., NFS-backed) of different VM workloads, and an average downtime of 2,500ms was measured. There is a large gap between our measurement and other research articles arguing that total service downtime including post-migration overheads in OpenStack live migration is under 1,000ms [60][61]. It was found that the main networking component in our lab OpenStack is Neutron (Open vSwitch [62]-based) but the authors' is nova-network (Linux bridge-based) which is officially deprecated since OpenStack Newton [63]. Detailed implementation differences are out of scope in this research and need future work, but it is indicated that service downtime can vary in live migration implementations of different hypervisors, releases/versions and purposes. Therefore, the post-migration overhead is represented as a constant value $constant\_postmig$ that depends on the underlying implementation.

[5] https://github.com/dpnm-ni/ni-migration-modeling-public

3.4.2 Migration Downtime and SLA Availability

As migration cost, live migration service downtime is related to SLA availability of each service. The basic idea is based on the fact that a service's downtime (i.e., unavailability) is accumulated during its migrations and the service provider must prevent the current migration decision from violating the service's SLA availability. For each service s, the current decision of migrating the service to a new host (e.g., one with the highest fitness score) can be either allowed or revoked (Equation 3.8).

$$migration(s \rightarrow m_{dst}) = \begin{cases} 0, \text{if } s.sla\_availability > 1 - unavailability(s, m_{dst}) \\ 1, \text{otherwise.} \end{cases} \tag{3.8}$$

$$unavailability(s, m_{dst}) = \frac{s.cumulative\_downtime + downtime(s, m_{dst})}{s.duration} \tag{3.9}$$

Service unavailability in Equation 3.9 represents the ratio of the expected total downtime over the requested duration for the service s. For each service s, the numerator is sum of the accumulated downtime during previous migrations and the expected $downtime$ (Equation 3.7) due to the current migration decision. Cumulative downtime of a service is counted when it i) is terminated due to the host server failure (Equation 3.4) or ii) previous migration decisions are allowed. In summary, the current decision on migrating a service to a new host is permitted and applied to the edge computing environment through Central Migration Controller only if the service availability after the migration is equal and greater than the SLA availability. If a migration decision is not allowed, Central Migration Controller ignores the corresponding migration request and instructs the agent who submits the request to revoke the corresponding action (i.e., do not migrate in Fig. 3.3). For instance of a service s whose SLA availability is 0.999 (99.9%) and the duration is 100s, if $s.cumulative\_downtime$ is 0.1s from the previously executed migrations and the expected $downtime$ from the current

migration decision is 0.1s, $unavailability$ of 0.002 results in the service availability of 0.998 (99.8%) which is less than the SLA availability, leading to the cancellation of the current migration request. Note that the proposed model can be considered as more practical by limiting the number of migrations for each service within SLA availability. It is also considered to assign different SLA availability depending on service types (Section 4.2).

The overall design architecture of the proposed multi-agent DRL model with the considerations above is described in Fig. 3.6. Finally, Algorithm 1 presents the main thread of the proposed algorithms, and Algorithm 2 presents the role of Central Migration Controller who mediates the operation of multiple agents and their responses. Detailed logic of each DC Migration Agent who interacts with the environment via migration to generate state, action and reward in dynamic service placement is provided in Algorithm 3.

Algorithm 1 Main Function

Input: srv_reqs: request records of services to be deployed (Section 4.2)

while $epi\_no < epi\_max\_no$ **do**
    $dep\_alg \leftarrow$ Deployment alg. of Random/CloudFirst/LeastLatency
    Run Scheduler with srv_reqs, dep_alg to execute service deployment
    $mig\_alg \leftarrow$ Placement alg. of MA_DQN/MA_TDAC
    // Call Algorithm 2 to execute dynamic service placement
    Run Central Migration Controller with mig_alg
    // One episode ends. Train agent with samples
    for Each DC Migration Agent **do**
        Update main Q-network
        if $epi\_no\%10 == 0$ **then**
            Synchronize target Q-network with main Q-network
        end if
    end for
end while

Algorithm 2 Central Migration Controller

Input: $mig\_alg, mig\_interval$: migration decision interval

while Any service is not finished **do**
  for Each DC Migration Agent **do**
    // Call Algorithm 3
    $transactions \leftarrow$ Decisions by DC Migration Agent with mig_alg
  end for
  $Reward_{joint} \leftarrow$ Average over all $Reward'$ in $transactions$
  for Each $transaction$ **do**
    $state, action, next\_state \leftarrow transaction$
    // Memory: replay buffer (DQN) or volatile buffer (TD-AC)
    Store $state, action, Reward_{joint}, next\_state$ in memory
  end for
  Sleep for $mig\_interval$
end while

Algorithm 3 DC Migration Agent

Input: mig_alg

Output: $state, action, Reward', next\_state$

$srvs \leftarrow$ Local services running in machines in dedicated DC
$letency \leftarrow$ Average service latency
$avail \leftarrow$ Average service availability
$dc\_profile \leftarrow$ Resource profiles averaged over machines in DC
$state \leftarrow$ Mini-batch of mapping features between $srvs$ and $dc\_profile$
$action \leftarrow$ Dest. DCs to which each srv migrates depending on mig_alg
for Each service srv **do**
    if srv is in the middle of previous migration **or** migrating srv to dest. DC is expected to violate $srv.SLA\_availability$ (Equation 3.8) **then**
        Revoke the migration action on srv
    else
        Execute live migration of srv to dest. DC
    end if
end for
$dc\_profile\_after \leftarrow$ DC profile after migrations in action
$next\_state \leftarrow$ Mapping features after migrations
$letency\_after \leftarrow$ Average service latency after migrations
$avail\_after \leftarrow$ Average service availability after migrations
$B_L \leftarrow$ Service latency benefit from $\Delta(latency, latency\_after)$
$B_A \leftarrow$ Service availability benefit from $\Delta(avail, avail\_after)$
$Reward' \leftarrow$ Reward calibration to $w_L B_L + w_A B_A$ (Equation 3.5)

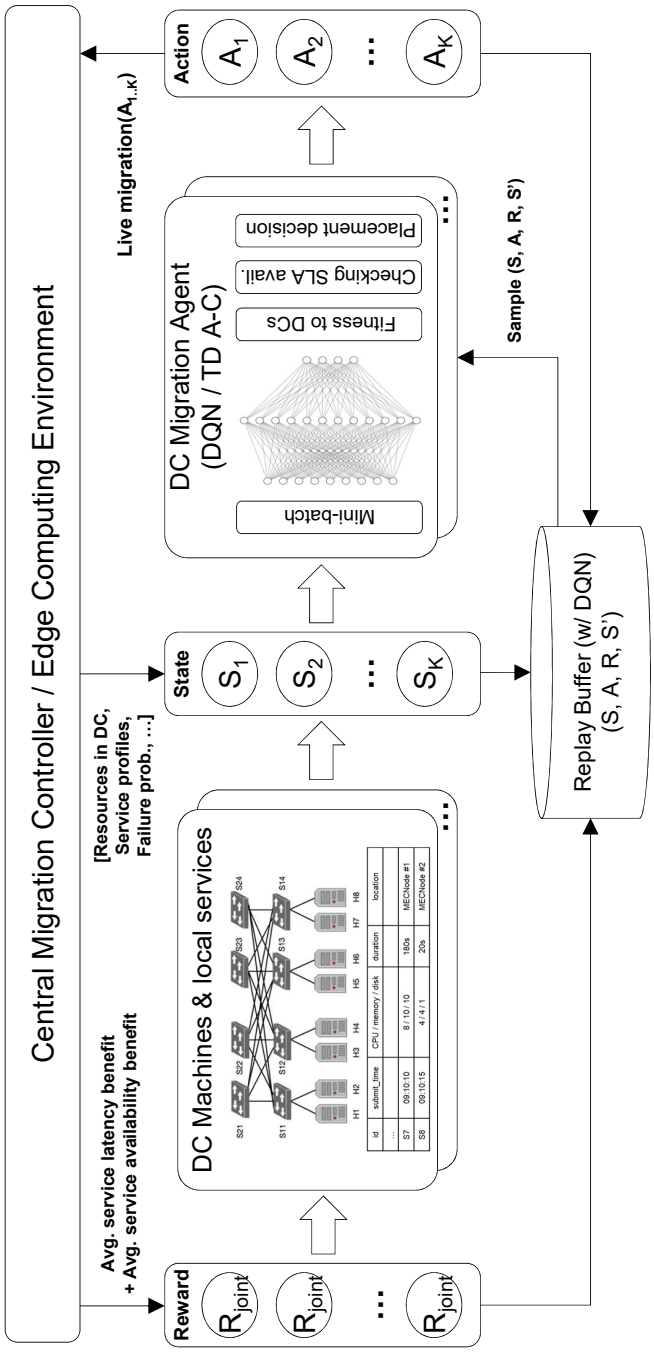


Figure 3.6: Dynamic service placement in edge computing by the proposed DRL algorithms

IV. Implementation

In this chapter, details on the implementation of a simulation environment on which the proposed DRL algorithms for dynamic service placement run are described first. Although there are existing simulators for edge computing [64][65][66], they are not originated from the purpose of service placement in multi-level DCs, so it is difficult to extend related edge computing components and develop service placement algorithms including the support on RL. So a new edge computing simulator is developed in this research using SimPy [67] which provides a Python library for discrete event simulation. In the implementation, all simulation components are individually organized in an object-oriented manner for extensibility and cooperate based on the event-driven processing under the main SimPy engine. Next, the reference edge computing topology is described where reference MEC services are dynamically placed by the proposed DRL algorithms.

4.1 Edge Computing Service Placement Simulator

The simulator for dynamic service placement in edge computing is highly motivated by the existing open source simulator, named CloudSimPy[1], which supports a testing framework for job scheduling algorithms in a single cloud data center [68]. Starting from the SimPy-based architecture of CloudSimPy, the proposed simulator differs in that multi-level DCs are geographically distributed and service requests are generated by users with their vicinity (edge adjacency). In addition, any network topology imported as a graph format is easily converted into the corresponding edge computing topology with attributes of link (e.g., delay) and node, including support on graph functions (e.g., computing shortest path cost). Fault injection is also available

[1] https://github.com/FengcunLi/CloudSimPy

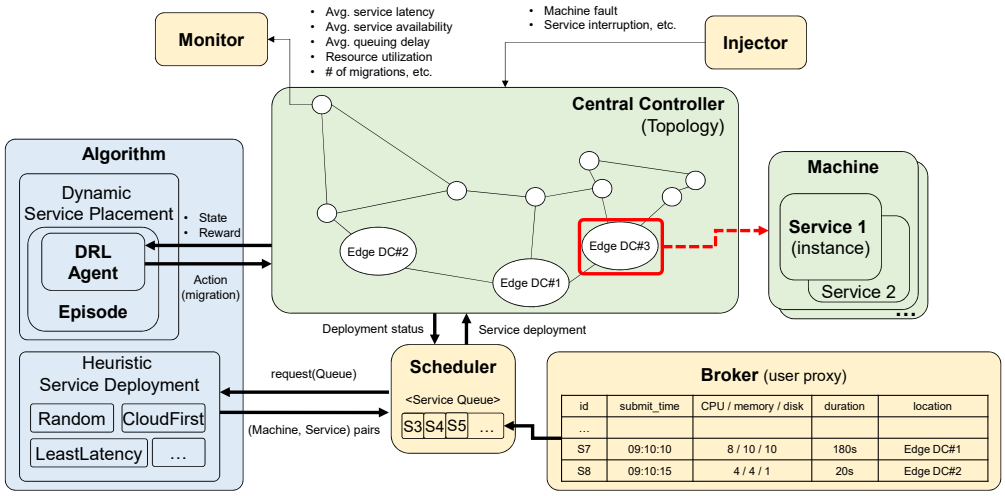


Figure 4.1: Implementation of the simulator for service placement in edge computing

to simulate failures of servers and network links in edge computing. In the following subsections, details of modular components in the proposed simulator are presented along with Fig. 4.1.

4.1.1 Simulation Components

Broker, Scheduler, Monitor, and Injector are all SimPy processes which fetch their own discrete events and interrupt related event handlers. The basic function of Broker is a proxy to submit each service request along its submission time into Scheduler. One may easily add more functions that cloud brokers usually do, for instance, filtering target cloud sites.

When a service request event is triggered, Scheduler puts it in a scheduling queue while notifying the current status of the queue to the service deployment algorithm. After a deployment decision is made on any service-to-machine (server) pair, Scheduler instantiates the service on the machine and propagates the state update to related simulation components. Scheduler repeats this process every scheduling interval (e.g., 1s) until the successful termination of all service instances is ensured, since this re-

search has designed and modeled service placement scenario where running services can be interrupted by fault (e.g., server failure) and they should be rescheduled by Scheduler to maintain the service availability for the remaining service duration.

The Monitor component periodically (e.g., 5s) collects information on the target edge computing environment including resource utilization of each machine and measurement values that indicate the performance of running services. On the other hand, Injector can trigger various events when certain conditions of the target environment are satisfied from monitoring. A generic Injector is extended to FaultInjector that simulates server failure scenario whose model is described in Section 3.3 and notifies the environment of failed servers by triggering server failure events. Running services in the failed hosts should be interrupted and re-deployed in other normal servers to complete their remaining service time. Note that these simulation components or SimPy processes have isolated Python logic for extensibility, but they are aligned on the same simulation time axis of the main SimPy engine. Simulation is a wrapper to execute those simulation components at once and attach them to the main Simpy engine.

4.1.2 Edge Computing Components

Central Controller, Machine, and Service are Python objects that compose respective properties of the target edge computing environment. Central Controller maintains lists of submitted service requests and their placement mappings to machines which are decided by the Algorithm component. Central Controller also stores the target edge computing topology as a Python NetworkX [69] object to provide convenient graph functions. From Internet Topology Zoo [70], one can find various network topology in graph formats (e.g., GML or GraphML). The simulator converts nodes and edges in the target graph representation into edge computing machines and network links respectively, and each link cost is assigned based on the geographical distance between two machines whose direct paths exist. However, one can preserve the original topology information on link delay or bandwidth if provided and use them in developing

own algorithms.

Central Controller creates a Machine object to abstract a physical server in DC. In the simulator, one can also generate a leaf-spine topology with several server racks to simulate an interior of edge DC and adjacent edge DCs, and test different service placement approaches across them. A Machine object has CPU, memory and disk capacity, as well as their current utilization, and a list of Service objects running on it.

A Service object is created by Broker when the corresponding service request is submitted. As a base unit of a placement decision with Machine, Service has a resource profile of demanding CPU, memory, and disk for its instance (e.g., VM or container), SLA latency and SLA availability, a counter for the remaining service duration and various timestamps to record the placement states, as well as user location[2]. Depending on the current placement location of a service, Monitor records the service path cost (e.g., round-trip time) between the user at a network edge and the corresponding host machine for the service duration. Note that in practical dynamic service placement scenario, path cost of a running service can be variable by user movement or service migration [31][32].

Service is also a SimPy process that is aligned with the shared simulation time axis from its deployment to the expiry of the service duration. Thus, the FaultInjector process can terminate a running service by interrupting the corresponding Service process handler [71] that forces to start a Python logic for rescheduling of the service. The rescheduling decision may be delayed depending on the current states of the service queue in Scheduler and available resources in machines, while imposing an increased placement cost on the service (e.g., service deployment delay). When the service is successfully finished, it returns the occupied resources to the host machine, and Central Controller updates the states of related simulation components so that other pending service requests can be processed by Scheduler. An example of interaction between simulation components in a deployment decision is elaborated in Fig. 4.2.

[2]The user location is simplified to be specified by ID of the closest edge DC.

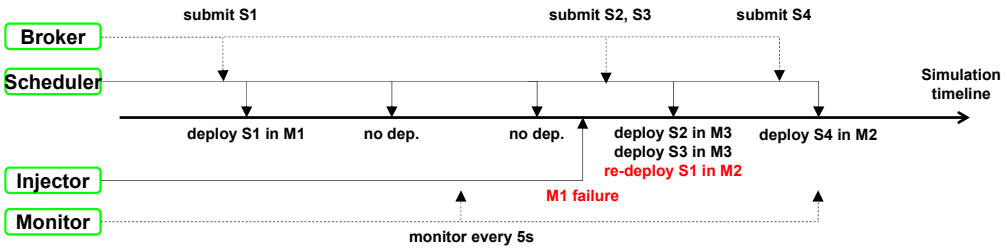


Figure 4.2: Event-driven interaction between simulation components

4.1.3 DRL Components

This simulator also modularizes DRL-related simulation components to support the extensibility in developing various (DRL) algorithms for service orchestration including dynamic service placement. An Algorithm component has a logic for service placement policy via live migration and is triggered by an Agent component which is a SimPy process aligned with the shared simulation timeline. For each DC, an individual Agent component and a shared Algorithm component compose the proposed multi-agent DRL models together as an individual DC Migration Agent (Fig. 3.3). Each DC Migration Agent identifies possible service-DC mappings and inputs their feature vectors (i.e., the current RL state in mini-batch) into the DNN in the corresponding Agent component. PyTorch [72] is used to implement DNNs with different hyper parameters depending on the use of specific DRL algorithms of DQN or TD-AC in the evaluation (Section 5.1). The outputs of the DNN with the TD-AC algorithm form a probabilistic distribution with the same dimension of the batch size, while representing fitness scores of all the service-DC input pairs. The larger fitness score of a pair, the higher probability of choosing the pair for the final service placement decision. In a DC Migration Agent, a service placement policy which is approximated by the DNN tends to behave in exploratory and generates irrational decisions at the early stage of learning. However, as learning from experiences in different migration decisions through repeating a simulation cycle (i.e., episode), the Agent component

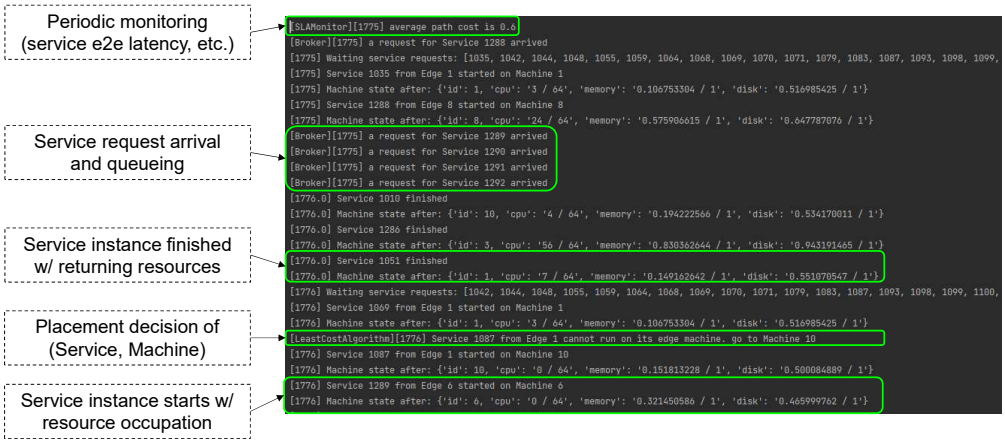


Figure 4.3: Implementation of event logging in the simulator

is reinforced (trained) to produce a larger fitness score to a service-DC mapping that can lead to a higher cumulative reward than other mappings. Scikit-learn [73] and NumPy [74] are also used in processing feature data (e.g., normalization) and calibrating reward functions.

Episode is a wrapper to attach Agent components to Simulation and execute them in the main SimPy engine at once. The cycle of one episode starts at the arrival of the first service request and ends at the finish of the last-standing service instance, during which each of multiple Agent components experiences different service placement decisions via live migration. The simulator is maintained in the open repository[3] for researchers who are interested in developing service placement algorithms in edge computing environment. Basic event logging is also supported in the simulator (Fig. 4.3).

4.2 Edge Computing Services and Topology

To secure the feasibility of the proposed method of dynamic service placement in edge computing, an open data set [44] of more than 10,000 requests for container-

[3]https://github.com/dpnm-ni/ni-migration-simulation-public

based online services traced in a real DC for 24 hours is used. Since service request records in the data set were traced so as for service deployment research (e.g., load balancing) in a single cluster of host servers, they were customized into edge computing service requests in this research. The new user location field is indeed the old destination machine ID field that represents the edge DC ID closest to the user after modulo operation with the number of total edge DCs. In addition, the service requests are categorized into multiple types, depending on SLA latency and SLA availability to represent the variety of service requirements in edge computing (Table 4.1). According to the service type, the instance of a service request can be dynamically migrated across the placement options in the target topology. Attributes of the modified data set (records) for edge computing service requests are shown in Fig. 4.4. The $plan\_cpu, plan\_mem$ and $plan\_disk$ fields respectively represent resources of virtual cores, memory and disk (both normalized over all records) demanded by the service instance. Arrival time of service requests depends on the $submit\_time$ field, and each service request stays as a service instance occupying the demanding resources until the $duration$ field expires.

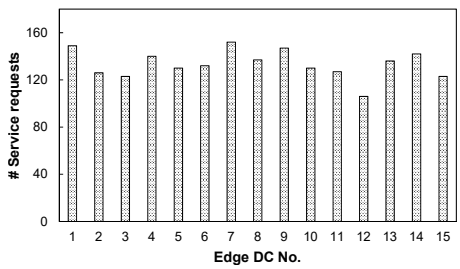
To simulate and validate practical orchestration scenarios of edge computing service requests above, a reference edge computing topology is built in the simulator with 1 cloud DC and 15 edge DCs (Fig. 4.5). A geography markup language (GML)) [75] file is created to draw the graph layout of nodes, edges and their configurations (e.g., IDs and link costs), and NetworkX library is used to project the graph into the corresponding topology in the simulator. Thanks to the use of GML, it is easy to change the configurations of nodes and edges, allowing developed service orchestration algorithms to be evaluated in various topology configurations of different numbers of machines, resource capacity or link costs. Inside each DC, an internal spine-and-leaf topology is provisioned as a sub-graph object of NetworkX to the super-graph to simulate a modern DC architecture and distinguish intra-DC management from inter-DC. For simplicity of edge DC, It is assumed that only one machine exists in a server rack (e.g., H1) and link costs between switches (e.g., S11) are 0.

Table 4.1: Types of service requests with different requirements

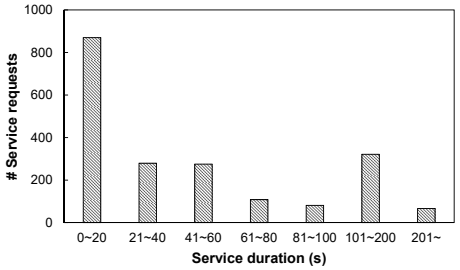
Service	Proportion	SLA latency	SLA availability	Placement options
Type 1	20%	<5ms	For each service record, randomly selected from [None, 99%, 99.9%, 99.99%]	Local edge DC (i.e., MEC host)
Type 2	20%	<10ms		Local edge DC, Adjacent edge DCs
Type 3	20%	<50ms		Local edge DC, Adjacent edge DCs, Cloud DC
Type 4	40%	None		Local edge DC, Edge DCs, Cloud DC

Based on the previous discussion on edge computing services, link delays are configured to intend under-10ms latency between adjacent edge DCs (directly connected via a core router), allowing services of type 2 with 10ms SLA latency to be capable of being deployed in an adjacent edge DC. Total link delays between an edge DC and the cloud DC is also intended to allow services of type 3 with 50ms SLA latency to be deployed in the cloud DC. In the same context, a service of type 1 (i.e., latency-critical service) should be deployed in the closest server to its user. There is no restriction in placement of services of type 4 which have no SLA latency, leaving room for optimization in placement policy with respect to various performance aspects. Link delays are configurable parameters in the simulator and assigned to intend different placement options (locations) depending on service types.

In the reference edge computing topology, the cloud DC features a huge amount of computing power, and high propagation delay from network edges. Whereas, each of 15 edge DCs features limited computing power, low propagation delay from net-



(a) Number of service requests over edge DCs



(b) Distribution of service duration

service_id	user_loc	plan_cpu	plan_mem	plan_disk	submit_time	duration	e2e_latency	e2e_availability
2	609	8	0.084818678	0.056808536	0	131.964467	1000	0
3	762	4	0.042409339	0.034085122	0	58.14285714	1000	99.9
4	1299	8	0.084818678	0.085212805	0	109.4526749	5	99.9
5	1135	8	0.084818678	0.056808536	54	38.06896552	10	99.99
6	1211	4	0.042409339	0.034085122	54	30.61538462	10	99
7	931	4	0.042409339	0.034085122	54	36.87671233	50	0
8	658	8	0.084818678	0.056808536	54	25.48717949	1000	99
9	383	4	0.042409339	0.034085122	77	11	10	99
10	409	8	0.084818678	0.056808536	77	3	50	99.9
11	717	4	0.042409339	0.034085122	95	154.5287356	1000	99.9
12	1260	8	0.084818678	0.056808536	95	91.0867052	1000	0
14	990	8	0.084818678	0.056808536	95	46.76190476	1000	99
15	1193	4	0.042409339	0.034085122	95	65.31111111	50	0
16	373	4	0.042409339	0.034085122	95	93.73684211	1000	99.9

(c) Examples of service request specification

Figure 4.4: Configuration of service requests for dynamic service placement in edge computing

work edges, and relatively high possibility of server failure (Section 3.3), considering that CAPEX and OPEX are restrictive in edge DCs [37]. In the next section, experiment results are explained on how baseline service deployment algorithms (without migration) differently treat such features in deploying services among machines in the cloud DC or edge DCs, which is followed by the evaluation of the proposed dynamic service placement algorithms that improve the baseline service deployment performance via DRL-based optimized live migration policy.

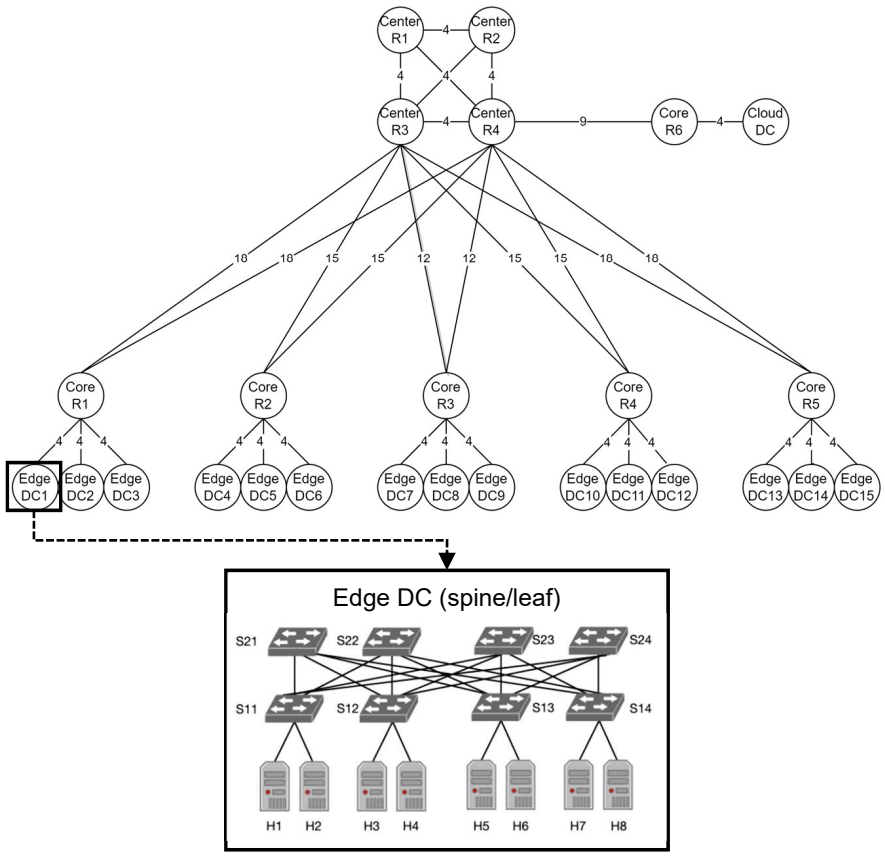


Figure 4.5: Reference edge computing topology of 1 cloud DC and 15 edge DCs with different link costs

V. Evaluation

5.1 Experiment Setup

Based on the proposed simulation environment, in this section, evaluation of the proposed dynamic service placement algorithms is conducted. If not explicitly mentioned, the following experiment results were verified with the first 1,000 service requests in the reference data set of edge computing services (Fig. 4.4(c)) and their placement in the reference edge computing topology (Fig. 4.5) with host machines of different configurations in Table 5.1. For simplicity, it is assumed that there is a single super server in the cloud DC that has a huge amount of resources and almost no possibility to the disk failure. The scales of memory and disk are set based on their normalized values specified in the default service request records [44].

Table 5.2 describes hyper parameters of each DRL agent (i.e., migration controller) and two different DRL algorithms in the proposed multi-agent DRL model for dynamic service placement via live migration (Section 3.2.2). Following experiments are conducted on the edge computing simulator running on a DELL PowerEdge R740 server with 36 CPU cores from two Intel Xeon Gold 5220 sockets (2.20GHz), 64GB RAM and Ubuntu 18.04 installed. For the reproducibility of experiments, all the implementation of the proposed algorithms and simulator is openly available on GitHub[1] with a guide for recommended execution environment.

5.2 Baseline Service Deployment Algorithms

In this thesis, (static) service deployment and dynamic service placement are distinguished. The former is algorithms to start the provisioning of a service from user's

[1] https://github.com/dpnm-ni/ni-migration-simulation-public

Table 5.1: Configurations of simulated machines

Machine	# CPU cores	Memory (norm.)	Disk (norm.)	Failure condition	Total number
In cloud DC	1000	10	10	None	1
In edge DC (i.e., MEC server)	16	0.2	0.2	>95% disk util. for 10 monitoring intervals in a row	15 * 8

Table 5.2: Hyper parameters of the proposed DRL model for dynamic service placement

	MA-DQN	MA-TDAC
wL (weight on service latency benefit)	1	
wA (weight on service avail. benefit)	0.5	
Learning rate	0.0001	0.001
Reward discount factor	0.98	
Migration decision interval (by agent)	10s	
# DNN neurons in input layer	10 $\rightarrow$ ReLu	
# DNN neurons in hidden layer 1	32 $\rightarrow$ ReLu	
# DNN neurons in hidden layer 2	32 $\rightarrow$ ReLu	
# DNN neurons in output layer	16[1]	1[2]$\rightarrow$ Softmax
Replay batch size	10	N/A
Probability of exploration[3]	10% $\rightarrow$ 1%	N/A

[1] 1 cloud DC + 15 edge DCs.

[2] Prob. distribution over 16 DCs.

[3] Decaying epsilon (Section 3.2.1)

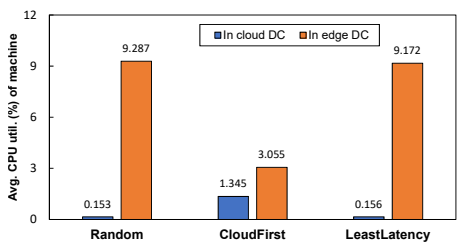
request by deploying the corresponding service instance in the initial location once (i.e., cloud DC or edge DC). The latter is the proposed algorithms to dynamically live-migrate a running service instance in certain locations possibly multiple times so that the placement can improve target performance indicators. Thus, service deployment is needed before dynamic service placement, and different service deployment results (i.e., initial service arrangement) can impact on the following dynamic service placement results. In this thesis, the following three baseline service deployment algorithms are used:

- **Random**: at a serivce request arrival, the Random algorithm finds a list of candidate machines that are eligible for the service regarding requested resources and SLA latency, and deploys the corresponding service instance in a randomly selected one.

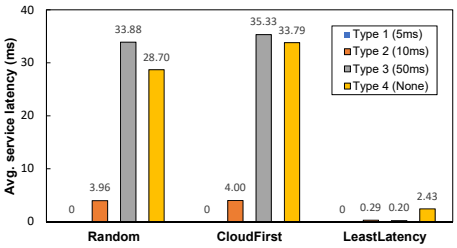
- **CloudFirst**: at a serivce request arrival, the CloudFirst algorithm deploys the service instance in a machine of the cloud DC first if not violating the SLA latency. Otherwise, the CloudFirst algorithm finds machines in edge DCs and deploys the service instance in the first available machine. In short, CloudFirst reserves resources in edge DCs for latency-critical services.

- **LeastLatency**: at a serivce request arrival, the LeastLatency algorithm sorts out candidate machines that are eligible for the service regarding requested resources and SLA latency in the ascending order of distance (i.e., end-to-end latency) from the service user. Then the algorithm deploys the service instance in the first one from the sorted machines.

To show the different initial service-machine mappings according to service deployment algorithms, average CPU utilization of machines in the cloud DC and edge DCs (averaged over the whole 15 edge DCs) and average service latency are compared in Fig 5.1. As shown in Fig. 5.1(a), CloudFirst uses the cloud DC's resources more than Random and LeastLatency, since CloudFirst uses edge DCs' resources only for



(a) Average CPU utilization of machines in cloud DC and edge DCs

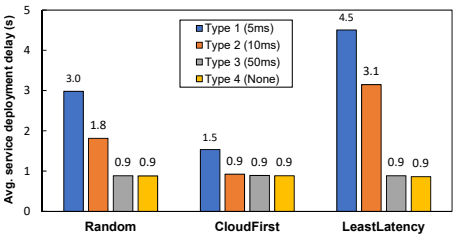


(b) Average service latency with different service types

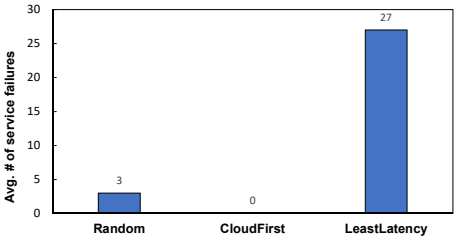
Figure 5.1: Performance of different service deployment algorithms in resource utilization and service latency

services with low latency requirement. Whereas, LeastLatency deploys any service in the closest edge machine to its user and finds the next closest one only if the demanding resources are not met in the first machine. As a result (Fig. 5.1(b)), average service latency of LeastLatency is extremely lower than that of CloudFirst for all the service types (Table 4.1). Note that the average service latency of Type 1 is 0 since the requirement of 5ms-latency requires deploying services in the closest edge DCs to their users, and the simulator assumes that link delay in access network is 0 (Section 4.2). CloudFirst and Random are not able to pursue the least latency, but are able to satisfy different services types whose requirements of SLA latency are 5ms, 10ms and 50ms respectively in service deployment.

The greedy behavior of LeastLatency leads to fast saturation of occupied resources in edge DCs and the following downsides are shown in Fig. 5.2. Service deployment delay in Fig. 5.2(a) represents interval from a service request arrival to initialization of the service instance for actual service provisioning (i.e., the start time of service duration). Obviously, low or zero service deployment delay is favored by users and can be related with SLA availability (e.g., service deployment rejection). For services with type 1 and 2, LeastLatency respectively shows 3 and 3.5 times higher service deployment delay than CloudFirst, since its likelihood of deploying any service in



(a) Average service deployment delay with different service types



(b) Total number of service failures

Figure 5.2: Performance of different service deployment algorithms in deployment queuing delay and service failures

edge DCs is higher, which results in more resource conflict between services in edge DCs. In addition, the more services run in resource-constraint edge DCs, the more edge servers (machines) are failed due to the server failure model that is based on disk over-utilization. Total number of services interrupted by their host server failure is largest in LeastLatency (Fig. 5.2(b)). CloudFirst does not generate service failures in the experiment setup since it deploys non-critical services (almost half of all service requests) in the cloud DC so that edge DCs can maintain relatively low resource utilization. The behavior of Random that lists candidate machines and randomly chooses one results in a sort of load balancing that leads to a lower probability of server failures than LeastLatency. However, its average service latency is not comparable to LeastLatency.

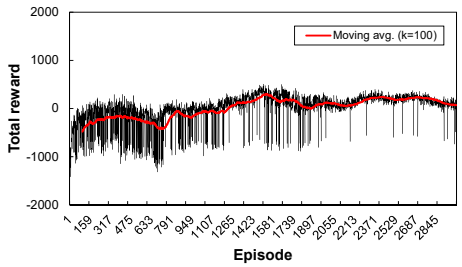
Those service deployment algorithms are too simple to be used in real scenario but the performance trade-off between cloud-orientation and edge-orientation in service deployment can be observed. In the following sections, CloudFirst is used as a baseline service deployment algorithm to evaluate the proposed DRL-based dynamic service placement algorithms since it is more practical in that SLA latency of all services are satisfied, while leaving room for improvement in service latency.

5.3 DRL Algorithms for Dynamic Service Placement

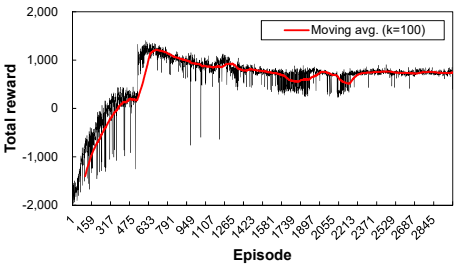
5.3.1 MA-DQN and MA-TDAC: proposed multi-agent DRL algorithms

In this thesis, two different DRL algorithms are proposed to generate optimal live migration policy for dynamic service placement in edge computing (Section 3.2.2). MA-DQN is in the family of value-based agent and MA-TDAC is in the family of policy-based agent. The differences leads to different generation methods for a fitness score of the given service-machine mapping. In the given state, MA-DQN estimates a fitness score by evaluating the worth of selecting the action (e.g., migrating service_1 to machine_A), while MA-TDAC computes a fitness score as a probability which is proportional to the worth of selecting the action. Both of them evaluate the worth of selecting an action from the resulted instant reward and update parameters of DNNs to induce their own policy to maximize the cumulative reward throughout iterative learning. If not explicitly mentioned, the corresponding hyper parameters in Table 5.2 are used in the following experiments.

The learning process of MA-DQN and MA-TDAC throughout 3,000 episodes is shown in Fig. 5.3. It can be observed with the moving averages of every 100 points that agents of both algorithms behave in a way that incrementally increase their total rewards as learning proceeds. Total reward (or cumulative reward) is sum of instant rewards from all migration actions during the episode. The increasing total reward can be interpreted as a positive signal to optimization of the objective function (Equation 1.1). Note that total reward converges on a certain level, since an instant reward (Equation 3.1) is sum of service latency benefit and service availability benefit which are values in trade-off relationship; the more migrations to network edges for low latency, the more server failures. To compare two algorithms using their mean total rewards, MA-DQN is better in the early stage of learning, but the mean total reward of MA-TDAC over the last 100 episodes, 739, indicates that overall service performance benefits from its live migration policy is larger than that of MA-DQN. The following experiments explore where these differences come from by analyzing their



(a) MA-DQN

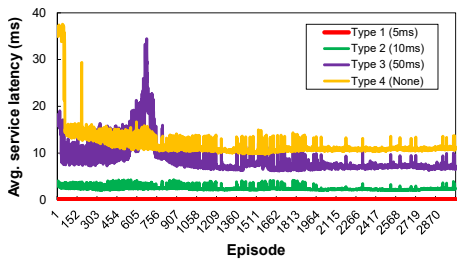


(b) MA-TDAC

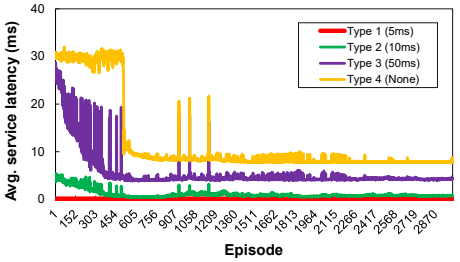
Figure 5.3: The changing trend in total rewards according to live migration policies of the proposed DRL algorithms

policies and resulted system performance. It is also observed that due to the explicit mechanism for policy's exploration (i.e., decaying epsilon) with high occurrences in the early stage of learning, more oscillation is observed in MA-DQN and diminishes as episodes iterates.

Average service latency of MA-DQN and MA-TDAC throughout 3,000 episodes is shown in Fig. 5.4. Latency for different service types is one objective that both algorithms try to minimize by evaluating service latency benefit from their migration decisions in the reward model. Considering that the average service latency results from the CloudFirst-based service deployment are 0, 4, 35.33 and 33.79ms for the corresponding service types respectively (Fig. 5.1(b)), they can be reduced to 0, 2.39, 7.13 and 11.12ms with dynamic service placement (i.e., live migration) by MA-DQN and 0, 0.79, 4.26 and 7.86ms by MA-TDAC. Average service latency is decreased by 80%, 88% and 76% for service type 2, 3 and 4 respectively with dynamic service placement by MA-TDAC, with 3,000 times of learning an episode during which each DRL agent periodically generates migration decisions on currently running services. In the given experiment setup, MA-TDAC achieves lower average service latency for all service types than MA-DQN, which results in larger values in service latency benefit, leading to the larger total reward in MA-TDAC (Fig. 5.3(b)). MA-DQN shows lower average service latency during the early stage of learning with the intermediate policy that



(a) MA-DQN

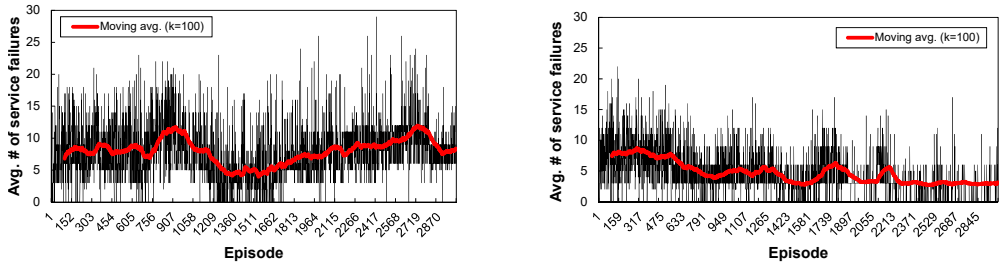


(b) MA-TDAC

Figure 5.4: Average service latency according to live migration policies of two proposed DRL algorithms

allows the rapid latency decrease for service type 3 and 4 from its initial highly exploratory migration decisions. However, MA-DQN behaves in reverse by generating migration decisions that rapidly increase latency of type 3 services at around episode 600 (note that in Fig. 5.3(a), total rewards also decrease there) and becomes converged on a certain level of service latency as learning proceeds. The conclusion is that MA-TDAC outperforms MA-DQN in average service latency, but different approaches on learning of two algorithms (i.e., policy-based vs. value-based) can lead to nondeterministic results in different experiment setups (e.g., number of service requests and machines) from which their agents need to re-learn.

Next, average numbers of service failures of MA-DQN and MA-TDAC throughout 3,000 episodes are shown in Fig. 5.5. In the considered dynamic service placement model, services running in a server whose disk utilization is overloaded for consecutive monitoring intervals are interrupted by the injection of a server failure event in the edge computing simulator, while re-deployment is needed for those failed services (Section 4.1.1). Thus, occurrences of service failures relate with service availability which is one objective that both algorithms try to maximize by evaluating service availability benefit from their migration decisions in the reward model. The average number of service failures resulted from the CloudFirst-based service deployment is 0 due to the conservative resource allocation (i.e, service deployment) in edge DCs



(a) MA-DQN (b) MA-TDAC

Figure 5.5: Average number of service failures according to live migration policies of two proposed DRL algorithms

(Fig. 5.2(b)). With the cost of increased number of service failures, MA-DQN and MA-TDAC can decrease average service latency by dynamically placing services from the cloud DC to an edge DC or vice versa via live migration (Fig. 5.4). Note that the common reward function is used with a larger weight on service latency ($w_L = 1$) and a smaller weight on service availability ($w_A = 0.5$) in both algorithms (Table 5.2), which means that migration policies pursue not only for lower service latency but also for higher availability.

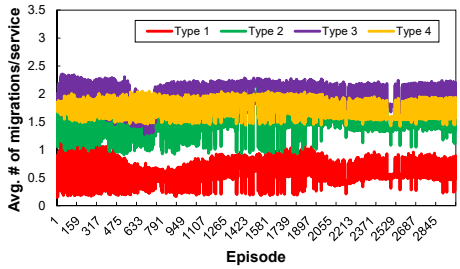
In Fig. 5.5(a), MA-DQN ends up with 8.33 service failures in average at the last 100 episodes despite the intermediate policy at around episode 1,500 with less service failures, since a DRL agent is expected to behave in a way that maximizes the total reward that involves service latency benefit as well. Whereas, MA-TDAC shows 3.13 service failures in average at the last 100 episodes and gradually decreasing trend of average service failures as learning proceeds (Fig. 5.5(b)). Thus, this research can conclude that MA-TDAC successfully find an optimal live migration policy that minimizes average service latency and maximizes service availability in the given experiment setup.

To disclose internal behavior of migration policy, average numbers of migrations per service of MA-DQN and MA-TDAC throughout 3,000 episodes are shown in Fig. 5.6. Both algorithms limit the number of migrations of a service in its duration by

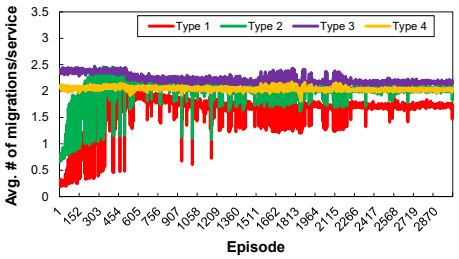
ensuring expected downtime due to the current migration decision must not violate the service's SLA availability (Section 3.4.2). Note that total number of migrations of a service in its duration is highly variable with regard to the migration decision interval of the controller/agent (e.g., 10s), since service duration is different for each service in the used data set (Section 4.2).

In Fig. 5.6(a), MA-DQN maintains the number of migrations per service constantly for all service types. Interestingly, average number of migrations per service of type 1 (blue) is 0.5, which means MA-DQN tends to be conservative in migrating critical services from their initial deployment locations (i.e., MEC host server that is closest to the user), in order to find different service-machine placement configurations which could provide performance improvement (e.g., average service latency). This fact also implies that occupation of edge resources by a critical service remains as static as possible for the whole service duration, which leads to tendency of higher resource utilization and more service failures than MA-TDAC (Fig. 5.6). This conservatism corresponds to the disadvantage of MA-DQN in the given experiment setups, but one administrator can favor in less number of live migrations and less service downtime for critical services in different dynamic placement scenario. On the other hand, MA-TDAC raises average number of migrations per service of type 1 and 2 in Fig. 5.6(b). It can be expected that this behavior of MA-TDAC encourages its agents to search different service-machine placement for performance improvement by migrating critical services more. Note that, as migration destination, candidate machines that can satisfy the strict SLA latency of critical services are confined in the same edge DC (i.e., local central office). So, one administrator may not favor in shuffling critical services (i.e., local migration within edge DC), at the expense of increased service downtime.

Fig. 5.7 supports the previous analysis on differences of both algorithms in generating migration decisions to cloud DC or edge DC. MA-TDAC shows the total number of migrations to edge DCs of 1990 and to the cloud DC of 14 while MA-DQN shows 21% less migrations to edge DCs and 13% less to the cloud DC. The more reduction of migrations to edge DCs in MA-DQN is originated from its conservative decision

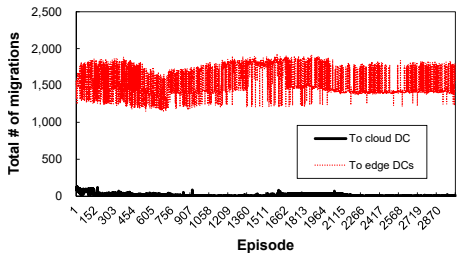


(a) MA-DQN

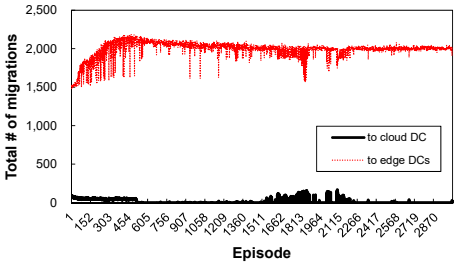


(b) MA-TDAC

Figure 5.6: Average number of migrations per service according to live migration policies of two proposed DRL algorithms



(a) MA-DQN



(b) MA-TDAC

Figure 5.7: Total number of migrations to the single cloud DC or 15 edge DCs according to live migration policies of two proposed DRL algorithms

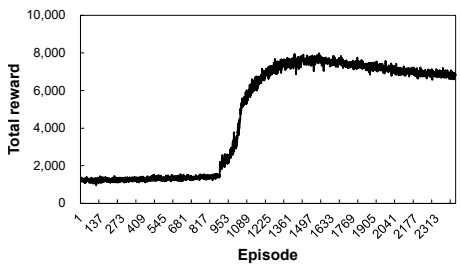
making on critical servicies, which allows them to stay in the initial MEC host servers longer and blocking migration of non-critical services to edge DCs as observed in Fig. 5.6(a).

5.3.2 SA-DDPG: single-agent DRL algorithm to compare

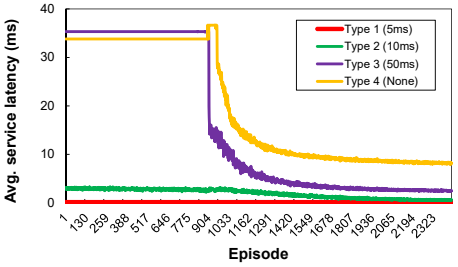
Before evaluation on the performance benefit using the proposed DRL algorithms, I implemented a comparable single-agent DRL algorithm for dynamic service placement. Referring to the related work [35] which uses DDPG for dynamic resource aware VNF placement, the single-agent algorithm is denoted as SA-DDPG (single-agent DDPG) below. SA-DDPG generates migration decisions based on fitness scores

of the given service-machine mappings (Table 2.2). The differences with the proposed algorithms (MA-DQN and MA-TDAC) lie in that SA-DDPG i) uses a single agent DDPG algorithm, ii) considers available link throughput, iii) does not consider SLA availability in its reward and iv) does not limit the number of migrations per service to ensure not to violate the service's SLA availability. In short, DDPG is a hybrid version of DQN and TD-AC algorithms in that it uses replay memory and doubling networks of DQN over the Actor-Critic architecture [76]. Since the implementation details are not fully described in the original paper [35], the single-agent DRL model introduced in Section 3.2.1 is used to implement SA-DDPG. However, link throughput is not considered in migration decision since this research assumes that the amount of bandwidth consumption due to migration is mere to the link bandwidth of transport networks used in modern edge computing environment [77].

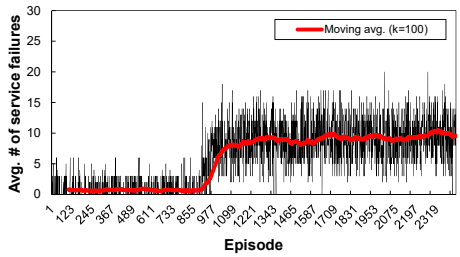
The key performance factors of SA-DDPG is summarized in Fig. 5.8. For equality regarding different learning time required for one episode, SA-DDPG was trained for 2,500 episodes with a smaller learning rate of 0.00001 which is 100 to 1,000 times less than MA-DQN and MA-TDAC respectively. The reason is that a gradient exploding problem that terminates the learning process is observed when using a high learning rate for the large state size which is inevitable in the single agent model. With the low learning rate, the single agent of SA-DDPG who uses the global information about service-machine mappings across all the DCs to generate migration decisions shows more stable learning curve than MA-DQN or MA-TDAC (Fig. 5.8(a)). However, intermediate policy of SA-DDPG dose not provide any performance benefit in average service latency before it reaches to the tipping point at around episode 900 (Fig. 5.8(b)). At the tipping point,the policy, for the first time, comes to know that migrating services of type 3 and 4 which are initially deployed in the cloud DC to edge DCs are highly beneficial to its reward (Fig. 5.8(d)). Since SA-DDPG in its reward evaluation is ignorant of service availability but aware of service latency only, average number of service failures shows an increasing curve as learning proceeds (Fig. 5.8(c)). Average number of migrations per service in SA-DDPG is larger than



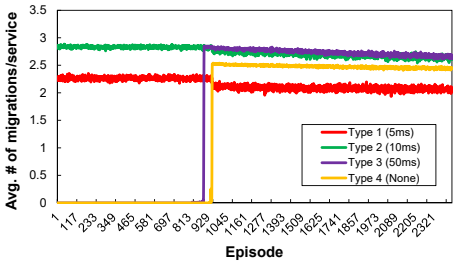
(a) Total reward



(b) Average service latency



(c) Average number of service failures



(d) Average number of migrations per service

Figure 5.8: Performance of SA-DDPG after learning 3,000 episodes

those of MA-DQN and MA-TDAC regardless of service types, since the modeled relationship between migration downtime and SLA availability (Section 3.4.2) is not considered in SA-DDPG.

5.3.3 Performance Comparison

In this section, different DRL algorithms, SA-DDPG, MA-DQN (proposed) and MA-TDAC (proposed), that generate optimized live migration policy for dynamic service placement are compared in various performance aspects. The baseline service deployment algorithm, CloudFirst which only cares about the initial placement of services, is also compared to provide pros and cons of introducing dynamic service placement in edge computing. Note that the presented performance results are the average value over the last 100 episode of total 3,000 episodes for each DRL algorithm.

In Fig. 5.9, the results of average service latency according to different dynamic

placement algorithms are provided. First, since CloudFirst does not conduct any live migration but only initial deployment for a service, it shows the baseline average service latency of different service types. Considering that propagation delay from an network edge (i.e., user) to the cloud DC is about 35ms in the reference edge computing topology in the simulator, non-critical services of type 3 and 4 are initially deployed in the cloud DC while critical services of type 1 and 2 are initially deployed in the local edge DC or adjacent edge DCs, where the propagation delay between them is set to 8ms (Fig. 4.5). For services of type 2, MA-TDAC shows 80%, 4% and 67% decrease, and for services of type 4, MA-TDAC also shows 77%, 9% and 29% decrease in average service latency of CloudFirst, SA-DDPG and MA-DQN respectively. For services of type 3, MA-TDAC shows 88% and 40% decrease in average service latency of CloudFirst and MA-DQN while 54% increase in that of SA-DDPG. In summary, MA-TDAC outperforms in average service latency except for type 3 services in SA-DDPG (Table 5.3).

In Fig. 5.10, the results of average service deployment delay according to different dynamic placement algorithms are provided. As reminder, service deployment delay represents how long a service request has to wait from its arrival to the first deployment in a host server. Since a critical service of type 1 with 5ms SLA latency can be placed only in the closest server (MEC host), its request at arrival suffers more service deployment delay as candidate servers' resource is being depleted, i.e., more service instances are already deployed in edge DCs. For services of type 1, SA-DDPG shows 11%, 33% and 42% decrease in average service deployment delay of CloudFirst, MA-DQN and MA-TDAC respectively, and there are no significant differences among algorithms for other service types. In summary, SA-DDPG outperforms in average service deployment delay (Table 5.4).

In Fig. 5.11, the results of average number of service failures according to different dynamic placement algorithms are provided. A service (instance) is failed due to its host server failure which is incurred by overload in disk utilization for consecutive monitoring intervals (Equation 3.4 in Section 3.2.2). A failed services immediately

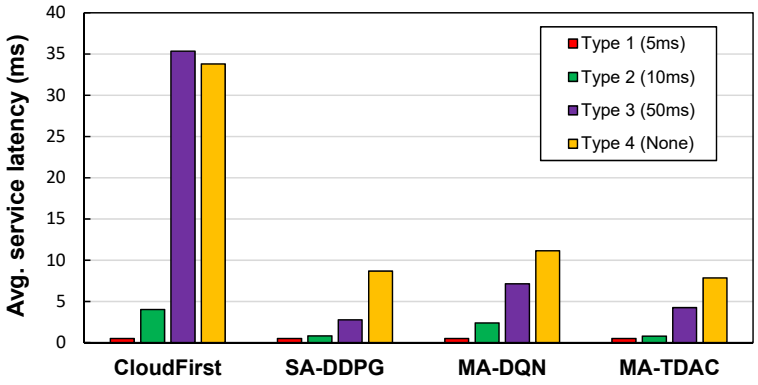


Figure 5.9: Average service latency with different algorithms

Table 5.3: Average service latency

	Average service latency (ms)			
	CloudFirst	SA-DDPG	MA-DQN	MA-TDAC
Type 1 (5ms)	0	0	0	0
Type 2 (10ms)	4.005	0.823	2.393	0.789
Type 3 (50ms)	35.328	2.771	7.132	4.258
Type 4 (None)	33.794	8.673	11.123	7.859

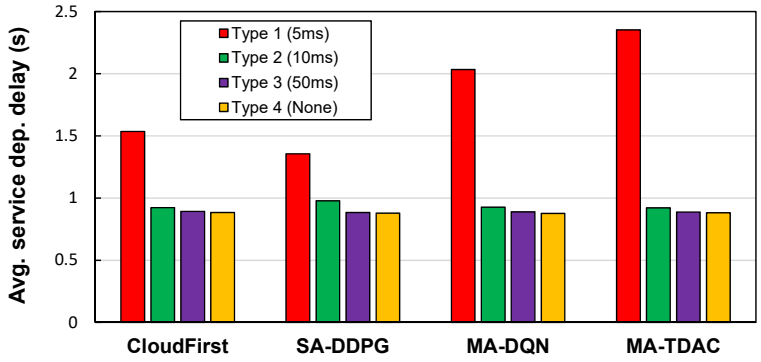


Figure 5.10: Average service deployment delay with different algorithms

Table 5.4: Average service deployment delay

	Average service deployment delay (s)			
	CloudFirst	SA-DDPG	MA-DQN	MA-TDAC
Type 1 (5ms)	1.536	1.356	2.033	2.353
Type 2 (10ms)	0.923	0.979	0.927	0.922
Type 3 (50ms)	0.893	0.884	0.889	0.888
Type 4 (None)	0.993	0.878	0.8771	0.882

terminates, so it should be re-deployed in a different machine to complete its remaining service duration with increased service deployment delay in the scheduling queue (Section 4.1). Thus, average number of service failures during one episode is a meaningful factor to indicate average service availability, and MA-TDAC shows 67% and 62% decrease in the measurement of SA-DDPG and MA-DQN. In summary, MA-TDAC outperforms in average number of service failures (Table 5.5). In that service availability and serivce latency is in trade-off, CloudFirst without dynamic service placement shows no service failures (i.e, no server failures) at the expense of high

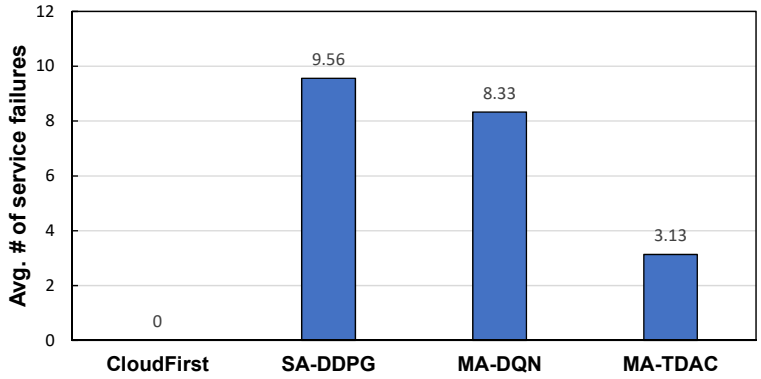


Figure 5.11: Average number of service failures with different algorithms

Table 5.5: Average number of service failures

	CloudFirst	SA-DDPG	MA-DQN	MA-TDAC
Avg. # of service failures	0	9.56	8.33	3.13

service latency.

In Fig. 5.12, the results of average number of migrations per service according to different dynamic placement algorithms are provided. Live migrations is a key enabler to realize dynamic service placement, but it also is costly in that downtime is implied (Section 3.4.1). Thus, it is important for a live migration policy to maintain the number of migrations as low as possible. Due to the migration cost model that revokes a current migration decision on a service if expected downtime incurs the violation of the service's SLA availability, both MA-DQN and MA-TDAC limit average number of migrations per service. Since there is no mention of migration downtime in the reference manuscript [35], any downtime cost model is not included in SA-DDPG. Since CloudFirst is not related with any migration performance, it is skipped in the experiment. MA-DQN shows 72% and 66% decrease for services of type 1, and 42% and 24% decrease for services of type2, compared to SA-DDPG and MA-TDAC re-

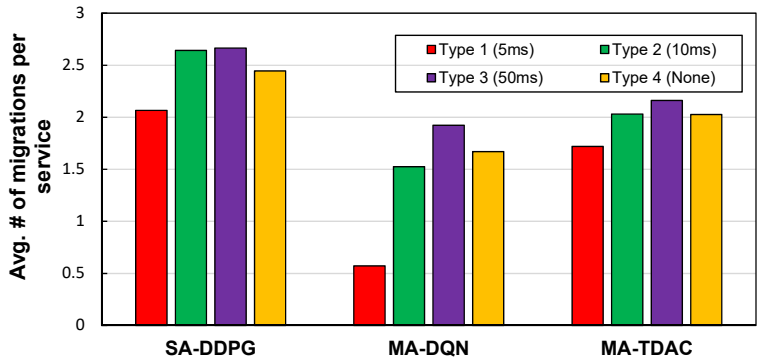


Figure 5.12: Average number of migrations per service with different algorithms

Table 5.6: Average number of migrations per service

	Average # of migrations per service		
	SA-DDPG	MA-DQN	MA-TDAC
Type 1 (5ms)	2.067	0.571	1.719
Type 2 (10ms)	2.642	1.525	2.031
Type 3 (50ms)	2.665	1.923	2.162
Type 4 (None)	2.445	1.669	2.0256

spectively. MA-DQN also shows 27% and 11% decrease for services of type 3, and 31% and 17% decrease for services of type 4, compared to SA-DDPG and MA-TDAC respectively. In summary, MA-DQN first outperforms in average number of migrations per service, which is preferred to be minimized especially for ultra reliable MEC services (Table 5.6).

In Fig. 5.13, the results of average learning time per episode according to different dynamic placement algorithms are provided. For SA-DDPG with a single agent, both MA-DQN and MA-TDAC with multiple agents, all agents in common generate

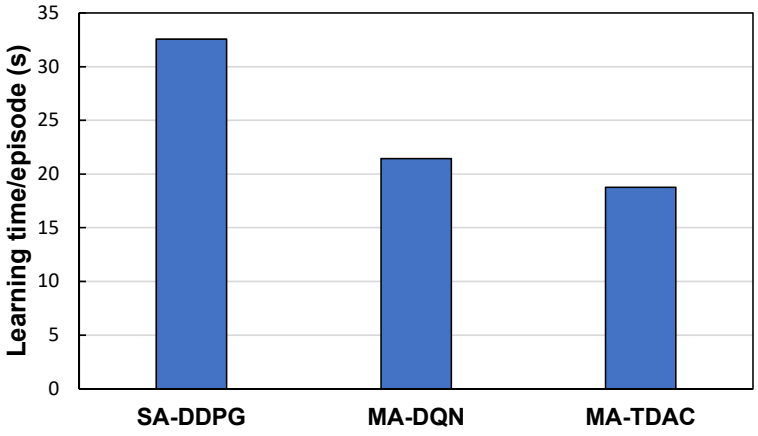


Figure 5.13: Average learning time per episode with different algorithms

migration decisions every 10s, and all hyper parameters of their DNNs are same in the experiment above[2] (Section 5.1). Thus, one major root cause of different learning time per episode is difference in state space size that each agent and its DNN should process. It was described that the state configuration in the single agent DRL model where the central controller (agent) uses global information on all services and machines to generate their mapping decisions (Section 3.2.1). This research also stated the state configuration in the multi-agent DRL model where each of multiple controllers considers mapping between only services hosted in the corresponding DC and other DCs, using DC-level information which is generalized from inside machines (Section 3.2.2). MA-TDAC outperforms SA-DDPG and MA-DQN with respectively 42% and 12% decrease in average learning time per episode (Table 5.7).

5.3.4 Cross-validation of Training Models

To validate how robust the proposed multi-agent DNN models trained on a certain set of service requests performs dynamic service placement to unseen service requests

[2]Since learning rate is a constant for multiplication in every gradient computation, it may effect on the number of learning steps to reach the target performance in mind but not a learning step itself.

Table 5.7: Average learning time per service

	SA-DDPG	MA-DQN	MA-TDAC
Avg. learning time per epi. (s)	32.576	21.432	18.774

(i.e., new environment dynamics), cross-validation was performed. First, the reference service request records (Fig. 4.4) are split into three different training data sets of 1000 records each. Note that not only traffic demand (Fig. 5.14) is different but also requested resources, duration and SLAs vary among service requests in each data set. MA-TDAC is used to train dynamic service placement policy for each data set during 1,000 and 4,000 episodes respectively. Then, trained models are tested on each test data set to generate the results of dynamic service placement on the corresponding service requests. Performance improvement (%) in service latency (Type 1 3) and total number of service failures (i.e., service availability) is presented in Fig. 5.15.

It can be observed that most of models trained with 1,000 episodes perform well on untrained data sets; the model trained with dataset1 is robust (i.e., generalizable) to dynamic service placement on unseen service requests in dataset2 and dataset3. Among 1,000 episode-trained models, the model trained with dataset2 shows performance decrease by 6 to 25% in service latency and by 24 to 33% in service availability, comparing to the static service deplyoment without any live migration (CloudFirst). The reason is that the training of 1,000 episodes is not sufficient to the dynamics of dataset2, with inconsistent total rewards in the model analysis. The fact is supported by the poor performance of testing on dataset2 itself (6.031% increase in service latency of Type 2) which is supposed to be optimized during the training stage. However, after training dataset2 for 4,000 episode long, the model finally shows improvement in service latency for untrained data sets. The conclusion is that there are different effective numbers to train models depending on data sets, or different environments.

The models trained with dataset1 and dataset3 for 4,000 episodes result in per-

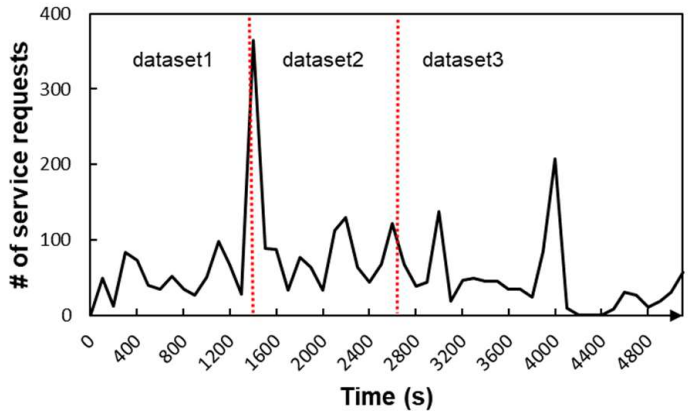


Figure 5.14: Data sets for cross-validation of training models; each data set has 1,000 service requests with different specifications

formance decrease in both service latency and service availability, comparing to the models trained for 1,000 episodes; in the analysis, total rewards start decreasing at a point as learning proceeds. One suspected root cause of this over-fitting behavior is the occurrence of catastrophic forgetting [78] where an agent may forget what it has previously learned via its DNN. This issue is one major downside followed by the use of neural network in general, and some measures are currently being proposed in the aspect of DRL; scheduling the proportion between exploitation and exploration [79]. In the current state of the proposed algorithms, it is required for operators to empirically find the best number of training according to the given environment.

		MA-TDAC trained on					
		dataset1		dataset2		dataset3	
		1000 epi.	4000 epi.	1000 epi.	4000 epi.	1000 epi.	4000 epi.
Tested on	dataset1	86.632	79.813	-6.031	70.035	85.087	58.272
	dataset2	85.992	83.866	-24.881	72.002	85.992	50.831
	dataset3	87.709	85.957	-21.691	70.488	88.181	60.772

(a) Service latency (Type 2)

		MA-TDAC trained on					
		dataset1		dataset2		dataset3	
		1000 epi.	4000 epi.	1000 epi.	4000 epi.	1000 epi.	4000 epi.
Tested on	dataset1	88.221	84.161	39.060	84.832	83.910	84.020
	dataset2	91.137	86.889	27.758	89.802	91.137	88.816
	dataset3	95.845	94.326	65.165	95.466	95.845	94.285

(b) Service latency (Type 3)

		MA-TDAC trained on					
		dataset1		dataset2		dataset3	
		1000 epi.	4000 epi.	1000 epi.	4000 epi.	1000 epi.	4000 epi.
Tested on	dataset1	76.303	72.330	10.890	68.957	71.908	72.152
	dataset2	77.825	75.510	14.016	70.965	77.939	76.269
	dataset3	69.530	68.349	3.160	65.002	70.033	68.690

(c) Service latency (Type 4)

		MA-TDAC trained on					
		dataset1		dataset2		dataset3	
		1000 epi.	4000 epi.	1000 epi.	4000 epi.	1000 epi.	4000 epi.
Tested on	dataset1	29.730	-21.622	-24.324	8.108	27.027	-16.216
	dataset2	100.000	100.000	-33.333	100.000	100.000	100.000
	dataset3	100.000	-37.500	62.500	62.500	100.000	100.000

(d) Total number of service failures

Figure 5.15: Cross-validation on performance improvement (%) of dynamic service placement by MA-TDAC comparing to static service deployment (CloudFirst)

VI. Conclusion

6.1 Summary

In this thesis, multi-agent DRL-based dynamic service placement algorithms are proposed with the corresponding DRL model for migration agents and an edge computing environment, of which a multi-level DC topology and different types of service requests are composed. In the proposed algorithms, each of migration controllers (i.e., agents) periodically generates placement decisions on which local services hosted in the dedicated DC should be migrated to where of DCs such that the cumulative reward on average service latency and average service availability is maximized. During the decision making, any migration action that is evaluated to violate the SLA availability due to the expected migration downtime, depending on the serivce's migration history and memory intensity, is revoked and treated as a "do not migrate" action.

Since an instant reward of any migration action is fed back to update the DNN in the context of each agent (e.g., its current state), decisions of individual agent do not always maximize the global benefit. In other words, a migration controller may generate greedy decisions that impede behavior of other migration controllers; for instance, a controller migrates all service instances to another DC since it learns that hosting none of service instance is beneficial to its reward, without recognition to the increased burden on the victim DC. To overcome this irrationality in the multi-agent DRL model, this research places the central migration controller who arbitrates behavior of different migration controllers by disseminating both edge-level information (e.g., average resource utilization of each DC) and the joint reward normalized over rewards of individual agents.

According to the verification results on dynamic placement of total 1,000 service requests of four different requirement types in 120 machines scattered around both the

cloud DC and 15 edge DCs, the proposed DRL algorithms show an increasing trend in total reward over episodes and the convergence where the accumulation of instant rewards on two trade-off values of service latency benefit and service availability benefit is maximized. In comparison to a single-agent DRL-based dynamic service placement algorithm in the existing work, the proposed algorithms decreases the average service latency by 9% and increases average service availability by 67% under 72% reduced number of migrations. The results of cross-validation of training models with 3,000 reference edge computing service requests indicate that the proposed dynamic service placement algorithms are robust to various edge computing environments if DNN models (i.e., agents) are sufficiently trained on the base environment. In summary, the proposed algorithms can improve the average service latency by dynamically relocating non-critical services from network core to edges or vice versa within the degree of not interrupting critical services, and the average service availability by balancing the demand from services instances over edge DCs, after recognition to the root cause of server failure (e.g., disk over-utilization) from iterative live migration experiences.

6.2 Future Work

6.2.1 Improvement in Multi-agent Model

Although the idea of 1-to-1 mapping between a DC and migration agent brings a smaller size of state space and shorter learning time than the single agent model, some experiment results indicate (possible) inferiority of the multi-agent model in service latency and service deployment delay, in the case of insufficient learning iterations. In particular, it is difficult for multiple agents to consent what are the best migration decisions and how each agent behaves to reach the optimality at the early stage of learning, where the misbehavior may lead to an local optimum. A complement mechanism can be added to the reward model to bootstrap [80] the behavior of multiple agents at the early stage of learning or apply the federated learning concept for robustness [81].

6.2.2 Verification in Real Testbed

As mentioned in Section 2.1.1, VM live migration in OpenStack involves a few seconds of service downtime mainly due to the post-migration overheads of attaching/detaching virtual network interfaces and reconfiguration of the IP address. Despite an estimation for live migration downtime is modeled (Section 3.4.1), the performance of VM live migration in a real cloud environment such as OpenStack is highly dependent on the implementation specifics. To verify the feasibility of the proposed dynamic service placement algorithms, the future work aims at porting the methodology in a laboratory-level OpenStack testbed for edge computing. In the previous work [9], a DRL-based auto-scaling methodology was successfully verified on the same real testbed . In addition to, future work also considers reducing the live migration service downtime in OpenStack within hundreds of ms unit by streamlining the overall process in the existing code in order to secure the feasibility of dynamic service placement via live migration.

요 약 문

5G 시대의 다양한 서비스 요구사항을 충족시키기 위해 널리 보급되어온 엣지 컴퓨팅(edge computing)은 운영자에게 자원 관리의 효율성을 제공할 뿐만 아니라 사용자에게 향상된 서비스 품질을 제공한다. 사용자에 가장 인접한 위치에 mission critical 서비스의 배포를 가능케하는 MEC 기술의 등장에 따라, 운영자들은 네트워크의 코어(core)와 엣지에 걸친 적절한 위치에 서비스를 최적 배치할 필요성을 느끼고 있다. 임의의 서비스에 대해 최소한의 중단 시간 내 물리적인 이전을 지원하는 live migration 기반 동적 서비스 배치(dynamic service placement)는 변화하는 환경에 대응하여 서비스 배치를 최적화하기 위한 지속적인 의사 결정 프로세스를 의미한다. 클라우드 환경을 구성하는 플랫폼의 발전으로 단일 데이터 센터 수준에서 부하분산(load balancing) 및 고가용성(high availability) 목적으로 live migration 기술이 활용되고 있다. 하지만 서로 다른 5G 서비스 요구사항 및 데이터센터의 계층적 구조와 같은 복잡성으로 인해 엣지 컴퓨팅 환경에서 동적 서비스 배치를 실제로 적용하는데는 어려움이 따르고 있다. 따라서 본 논문에서는 정책의 생성 주체와 엣지 컴퓨팅 환경 간의 상호작용을 모델링하는 심층강화학습(Deep Reinforcement Learning, DRL) 기반의 동적 서비스 배치 알고리즘을 제안한다. 제안하는 다중 에이전트(multi-agent) DRL 알고리즘은 엣지 컴퓨팅 환경에서의 동적 서비스 배치 문제를 강화학습의 환경(environment), 에이전트(agent), 상태(state), 액션(action) 및 보상(reward)으로 추상화하여, 에이전트가 여러 데이터 센터에 걸쳐 서비스 인스턴스를 migration하고 그 가치를 평가함으로써 최선의 배치 정책을 학습할 수 있게한다. 제안하는 알고리즘은 또한 live migration 결정을 검증할 때 서비스 중단시간(downtime)을 migration 비용(cost)으로 평가하고, 서버 장애에 따른 서비스 가용성(availability)을 고려한다. 16개 데이터 센터에 있는 120개 서버에 1,000개 서비스 요청을 처리(배포)하는 엣지 컴퓨팅 환경의 시뮬레이션에서, 기존의 동적 서비스 배치를 위한 단일 에이전트(single-agent) DRL 알고리즘과 비교하여, 제안된 다중 에이전트 DRL 알고리즘은 서비스 당 평균 migration 횟수가 72% 감소했음에도 불

구하고 평균 서비스 지연 시간을 9% 감소시키고 평균 서비스 가용성은 67% 증가시킨다.

References

[1] Xinli Hou and Liang Xia. Verticals URLLC Use Cases and Requirements. `https://www.ngmn.org/publications/verticals-urllc-use-cases-and-requirements.html`, 2020 [Online]. Accessed: 2022-04-22.

[2] Peter Corcoran and Soumya Kanti Datta. Mobile-edge computing and the internet of things for consumers: Extending cloud computing and services to the edge of the network. *IEEE Consumer Electronics Magazine*, 5(4):73–74, 2016.

[3] Adel Nadjaran Toosi, Redowan Mahmud, Qinghua Chi, and Rajkumar Buyya. Management and orchestration of network slices in 5g, fog, edge, and clouds. *Fog and Edge Computing: Principles and Paradigms*, 8:79–96, 2019.

[4] Petar Popovski, Kasper Fløe Trillingsgaard, Osvaldo Simeone, and Giuseppe Durisi. 5g wireless network slicing for embb, urllc, and mmtc: A communication-theoretic view. *Ieee Access*, 6:55765–55779, 2018.

[5] Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. An overview of service placement problem in fog and edge computing. *ACM Computing Surveys (CSUR)*, 53(3):1–35, 2020.

[6] Tao Ouyang, Zhi Zhou, and Xu Chen. Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. *IEEE Journal on Selected Areas in Communications*, 36(10):2333–2345, 2018.

[7] Zhaolong Ning, Peiran Dong, Xiaojie Wang, Shupeng Wang, Xiping Hu, Song Guo, Tie Qiu, Bin Hu, and Ricky YK Kwok. Distributed and dynamic service

placement in pervasive edge computing networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(6):1277–1292, 2020.

[8] Adyson Magalhäes Maia, Yacine Ghamri-Doudane, Dario Vieira, and Miguel Franklin de Castro. Dynamic service placement and load distribution in edge computing. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2020.

[9] Do-Young Lee, Se-Yeon Jeong, Kyung-Chan Ko, Jae-Hyoung Yoo, and James Won-Ki Hong. Deep q-network-based auto scaling for service in a multi-access edge computing environment. *International Journal of Network Management*, 31(6):e2176, 2021.

[10] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. Vm live migration at scale. *ACM SIGPLAN Notices*, 53(3):45–56, 2018.

[11] Qingwei Lin, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, Youjiang Wu, Randolph Yao, et al. Predicting node failure in cloud service systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 480–490, 2018.

[12] Shunmugapriya Ramanathan, Koteswararao Kondepu, Miguel Razo, Marco Tacca, Luca Valcarenghi, and Andrea Fumagalli. Live migration of virtual machine and container based mobile core network components: A comprehensive study. *IEEE Access*, 9:105082–105100, 2021.

[13] Keerthana Govindaraj and Alexander Artemenko. Container live migration for latency critical industrial applications on edge computing. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 83–90. IEEE, 2018.

[14] Qi Wang, Jose Alcaraz-Calero, Ruben Ricart-Sanchez, Maria Barros Weiss, Anastasius Gavras, Navid Nikaein, Xenofon Vasilakos, Bernini Giacomo, Giardina Pietro, Mark Roddy, et al. Enable advanced qos-aware network slicing in 5g networks for slice-based media use cases. *IEEE transactions on broadcasting*, 65(2):444–453, 2019.

[15] ZTE Corporation. 5G Medium and Long Term Planning for Operators. `https://res-www.zte.com.cn/mediares/zte/Files/PDF/white_book/202011171751.pdf`, 2020 [Online]. Accessed: 2022-04-22.

[16] Ioannis Sarrigiannis, Elli Kartsakli, Kostas Ramantas, Angelos Antonopoulos, and Christos Verikoukis. Application and network vnf migration in a mec-enabled 5g architecture. In *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–6. IEEE, 2018.

[17] Verizon. Public vs. private MEC explained. `https://www.verizon.com/about/news/public-private-mec`, 2021 [Online]. Accessed: 2022-06-08.

[18] Khaled Z Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[19] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[20] Changyeon Jo, Youngsu Cho, and Bernhard Egger. A machine learning approach to live migration modeling. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 351–364, 2017.

[21] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W Moore, and Andy Hopper. Predicting the performance of virtual machine migration. In *2010 IEEE international symposium on modeling, analysis and simulation of computer and telecommunication systems*, pages 37–46. IEEE, 2010.

[22] Marc Sauter and Chris White. Creating the 5G Factory of the Future. `https://www.gsma.com/iot/wp-content/uploads/2020/10/2020-10-13-IoT-WebTalk-5G-Private-Networks.pdfl`, 2020 [Online]. Accessed: 2022-04-22.

[23] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[25] Arryon D Tijsma, Madalina M Drugan, and Marco A Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2016.

[26] Dotan Castro, Dmitry Volkinshtein, and Ron Meir. Temporal difference based actor critic learning-convergence and neural implementation. *Advances in neural information processing systems*, 21, 2008.

[27] Bo Yi, Xingwei Wang, Min Huang, and Kexin Li. Design and implementation of network-aware vnf migration mechanism. *IEEE Access*, 8:44346–44358, 2020.

[28] Lanlan Rui, Xushan Chen, Zhipeng Gao, Wenjing Li, Xuesong Qiu, and Luoming Meng. Petri net-based reliability assessment and migration optimization strategy of sfc. *IEEE Transactions on Network and Service Management*, 18(1):167–181, 2020.

[29] Debabrota Basu, Xiayang Wang, Yang Hong, Haibo Chen, and Stéphane Bressan. Learn-as-you-go with megh: Efficient live migration of virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1786–1801, 2019.

[30] Amina Lejla Ibrahimpašić, Bin Han, and Hans D Schotten. Ai-empowered vnf migration as a cost-loss-effective solution for network resilience. In *2021 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 1–6. IEEE, 2021.

[31] Tarik Taleb and Adlen Ksentini. Follow me cloud: interworking federated clouds and distributed mobile networks. *IEEE Network*, 27(5):12–19, 2013.

[32] Cheng Zhang and Zixuan Zheng. Task migration for mobile edge computing using deep reinforcement learning. *Future Generation Computer Systems*, 96:111–118, 2019.

[33] Sung Woon Park, Azzedine Boukerche, and Shichao Guan. A novel deep reinforcement learning based service migration model for mobile edge computing. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–8. IEEE, 2020.

[34] Jing Zeng, Ding Ding, Xuan Kai Kang, Huamao Xie, and Qian Yin. Adaptive drl-based virtual machine consolidation in energy-efficient cloud data center. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[35] Anestis Dalgkitsis, Prodromos-Vasileios Mekikis, Angelos Antonopoulos, Georgios Kormentzas, and Christos Verikoukis. Dynamic resource aware vnf placement with deep reinforcement learning for 5g networks. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[36] Yu Dai, Qiuhong Zhang, and Lei Yang. Virtual machine migration strategy based on multi-agent deep reinforcement learning. *Applied Sciences*, 11(17):7993, 2021.

[37] Louiza Yala, Pantelis A Frangoudis, and Adlen Ksentini. Latency and availability driven vnf placement in a mec-nfv environment. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[38] Seyeon Jeong, Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Proactive live migration for virtual network functions using machine learning. In *2021 17th International Conference on Network and Service Management (CNSM)*, pages 335–339. IEEE, 2021.

[39] Jungmin Son, Amir Vahid Dastjerdi, Rodrigo N Calheiros, and Rajkumar Buyya. Sla-aware and energy-efficient dynamic overbooking in sdn-based cloud data centers. *IEEE Transactions on Sustainable Computing*, 2(2):76–89, 2017.

[40] Sultan Alanazi and Bechir Hamdaoui. Energy-aware resource management framework for overbooked cloud data centers with sla assurance. In *2018 IEEE global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

[41] Weiliang Ji, Shihui Duan, Renai Chen, Song Wang, and Qiang Ling. A cnn-based network failure prediction method with logs. In *2018 Chinese Control And Decision Conference (CCDC)*, pages 4087–4090. IEEE, 2018.

[42] Zhijing Li, Zihui Ge, Ajay Mahimkar, Jia Wang, Ben Y Zhao, Haitao Zheng, Joanne Emmons, and Laura Ogden. Predictive analysis in network function virtualization. In *Proceedings of the Internet Measurement Conference 2018*, pages 161–167, 2018.

[43] Hongman Wang, Yingxue Li, Ao Zhou, Yan Guo, and Shangguang Wang. Service migration in mobile edge computing: A deep reinforcement learning approach. *International Journal of Communication Systems*, page e4413, 2020.

[44] Alibaba. Alibaba Cluster Trace Program. `https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2017`, 2022 [Online]. Accessed: 2022-05-26.

[45] Tianpei Yang, Hongyao Tang, Chenjia Bai, Jinyi Liu, Jianye Hao, Zhaopeng Meng, and Peng Liu. Exploration in deep reinforcement learning: a comprehensive survey. *arXiv preprint arXiv:2109.06668*, 2021.

[46] Aakash Maroti. Rbed: Reward based epsilon decay. *arXiv preprint arXiv:1910.13701*, 2019.

[47] OpenStack foundation. Live-migrate instances. `https://docs.openstack.org/nova/latest/admin/live-migration-usage.html`, 2019 [Online]. Accessed: 2022-05-31.

[48] George Philipp, Dawn Song, and Jaime G Carbonell. The exploding gradient problem demystified-definition, prevalence, impact, origin, tradeoffs, and solutions. *arXiv preprint arXiv:1712.05577*, 2017.

[49] Sekitoshi Kanai, Yasuhiro Fujiwara, and Sotetsu Iwamura. Preventing gradient explosions in gated recurrent units. *Advances in neural information processing systems*, 30, 2017.

[50] Xiaofei Wang, Yiwen Han, Chenyang Wang, Qiyang Zhao, Xu Chen, and Min Chen. In-edge ai: Intelligentizing mobile edge computing, caching and communication by federated learning. *IEEE Network*, 33(5):156–165, 2019.

[51] Alessandro Pellegrini, Pierangelo Di Sanzo, and Dimiter R Avresky. A machine learning-based framework for building application failure prediction models. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 1072–1081. IEEE, 2015.

[52] Xiaoyi Sun, Krishnendu Chakrabarty, Ruirui Huang, Yiquan Chen, Bing Zhao, Hai Cao, Yinhe Han, Xiaoyao Liang, and Li Jiang. System-level hardware failure prediction using deep learning. In *2019 56th ACM/IEEE design automation conference (DAC)*, pages 1–6. IEEE, 2019.

[53] Guosai Wang, Lifei Zhang, and Wei Xu. What can we learn from four years of data center hardware failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2017.

[54] Mayank Mishra, Anwesha Das, Purushottam Kulkarni, and Anirudha Sahoo. Dynamic resource management using virtual machine migrations. *IEEE Communications Magazine*, 50(9):34–40, 2012.

[55] Ali Malik, Volodymyr Kuleshov, Jiaming Song, Danny Nemer, Harlan Seymour, and Stefano Ermon. Calibrated model-based deep reinforcement learning. In *International Conference on Machine Learning*, pages 4314–4323. PMLR, 2019.

[56] Yixuan Liu, Hu Wang, Xiaowei Wang, Xiaoyue Sun, Liuyue Jiang, and Minhui Xue. Delayed rewards calibration via reward empirical sufficiency. *arXiv preprint arXiv:2102.10527*, 2021.

[57] Wikipedia. Inverse hyperbolic functions. `https://en.wikipedia.org/wiki/Inverse_hyperbolic_functions#Inverse_hyperbolic_tangent`, 2022 [Online]. Accessed: 2022-05-16.

[58] Jintian Hu, Gaocai Wang, Xiaotong Xu, and Yuting Lu. Study on dynamic service migration strategy with energy optimization in mobile edge computing. *Mobile Information Systems*, 2019, 2019.

[59] Aidan Shribman and Benoit Hudzia. Pre-copy and post-copy vm live migration for memory intensive applications. In *European Conference on Parallel Processing*, pages 539–547. Springer, 2012.

[60] A Blog of the ZHAW Zurich University of Applied Sciences. An analysis of the performance of live migration in Openstack. `https://blog.zhaw.ch/icclab/`

an-analysis-of-the-performance-of-live-migration-in-openstack/, 2014 [Online]. Accessed: 2022-05-16.

[61] Md Israfil Biswas, Gerard Parr, Sally McClean, Philip Morrow, and Bryan Scotney. A practical evaluation in openstack live migration of vms using 10gb/s interfaces. In *2016 IEEE symposium on service-oriented system engineering (SOSE)*, pages 346–351. IEEE, 2016.

[62] OpenStack. Open vSwitch: Self-service networks. https://docs.openstack.org/neutron/latest/admin/deploy-ovs-selfservice.html, 2022 [Online]. Accessed: 2022-06-08.

[63] OpenStack. Networking with nova-network. https://docs.openstack.org/nova/pike/admin/networking-nova.html, 2021 [Online]. Accessed: 2022-06-08.

[64] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3493, 2018.

[65] Jungmin Son, TianZhang He, and Rajkumar Buyya. Cloudsimsdn-nfv: Modeling and simulation of network function virtualization and service function chaining in edge computing environments. *Software: Practice and Experience*, 49(12):1748–1764, 2019.

[66] S Massari, N Mirizzi, G Piro, and G Boggia. An open-source tool modeling the etsi-mec architecture in the industry 4.0 context. In *2021 29th Mediterranean Conference on Control and Automation (MED)*, pages 226–231. IEEE, 2021.

[67] Team Simpy. Discrete event simulation for Python. https://simpy.readthedocs.io/en/latest/, 2022 [Online]. Accessed: 2022-05-21.

[68] Fengcun Li and Bo Hu. Deepjs: Job scheduling based on deep reinforcement learning in cloud data center. In *Proceedings of the 2019 4th international conference on big data and computing*, pages 48–53, 2019.

[69] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[70] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[71] Team Simpy. Process Interaction. `https://simpy.readthedocs.io/en/latest/simpy_intro/process_interaction.html`, 2022 [Online]. Accessed: 2022-05-21.

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[73] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[74] Numpy project and community. The fundamental package for scientific computing with Python. `https://numpy.org/`, 2022 [Online]. Accessed: 2022-06-01.

[75] David S Burggraf. Geography markup language. *Data Science Journal*, 5:178–204, 2006.

[76] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[77] Wenfeng Xia, Peng Zhao, Yonggang Wen, and Haiyong Xie. A survey on data center networking (dcn): Infrastructure and operations. *IEEE communications surveys & tutorials*, 19(1):640–656, 2016.

[78] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[79] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.

[80] Carles Gelada and Marc G Bellemare. Off-policy deep reinforcement learning by bootstrapping the covariate shift. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3647–3655, 2019.

[81] Yufeng Zhan, Peng Li, and Song Guo. Experience-driven computational resource allocation of federated learning by deep reinforcement learning. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 234–243. IEEE, 2020.

Acknowledgements

먼저 의욕만 앞서고 연구 자체는 서툴렀던 저를 계속해서 좋은 방향으로 이끌어 주시고 격려해주신 지도교수 홍원기 교수님께 감사드립니다. 교수님께서 언제나 새로운 연구 분야에 관심을 가지시고 선도하시기 때문에 저도 같은 태도를 가지려고 노력하게 되었으며, 연구실에도 좋은 자극이 되는 것 같습니다. 또한 내색은 안하시지만 언제나 꼼꼼하게 연구 지도해주시는 유재형 교수님께도 감사 인사를 드립니다. 들려주시는 여러 이야기를 통해 연구자로서 뿐만 아니라 사회인으로서 필요한 덕목을 배울 수 있었습니다.

DPNM 연구실 입학전에 선배들이 너무 엄격하면 어쩌지 걱정했던게 엊그제 같은데 이제 제가 연구실 최고참이 되었습니다. 과연 제가 그 당시에 우러러 보던 선배들의 모습에 이르렀을까하는 부분에는 개인적인 아쉬움이 남지만, 연구실 선, 후배 동료들과 함께한 좋은 기억들이 더 많은 것 같습니다. 학교든 사회에서 만나든, 앞으로도 계속해서 좋은 인연을 이어갈 수 있기를 바랍니다.

저를 위한 것이라면 언제나 부족함 없게 물심양면 지원해주시는 부모님께 감사 인사를 전합니다. 존경하며, 앞으로 부끄럽지 않게 살아가는 것으로 보답드리고 싶습니다. 끝으로 힘든 순간마다 응원해주시고 용기를 주신 고모, 고모부께도 감사 드립니다.

Curriculum Vitae

Personal Information

Name: **Seyeon Jeong**

Position: Ph.D. student

Laboratory: Distributed Processing & Network Management (DPNM) Lab.

Department: Computer Science and Engineering

Education

2018. 03. – 2022. 08. Department of Computer Science and Engineering, Pohang University of Science and Technology (Ph.D.)

2015. 03. – 2018. 02. Department of Computer Science and Engineering, Pohang University of Science and Technology (M.S.)

2009. 03. – 2015. 02. School of Computer Science and Engineering, Kyungpook National University (B.S.)

Research Areas of Interest

Software-Defined Networking (SDN), Network Function Virtualization (NFV), Cloud Computing, Artificial Intelligence-based Network Management

Research / Project Experiences

1. **Development of virtual network management technology based on artificial intelligence**

   Funded by Institute for Information & Communications Technology Promotion (IITP) (2018 – 2023)

   This research project aims to study a virtual network management using artificial intelligence (AI), including development of AI-based algorithms which are essential for VNF life-cycle management to monitor and control resources of NFV environment in real time. My contribution to this project is to participate in building an OpenStack-based NFV testbed and implementing a PoC system for machine learning based NFV management. The monitoring part of the system uses collectd, OpenFlow (flow statistics and ONOS-based Virtual TAP for packet mirroring) and Kafka for efficient measuring the capacity or the states of NFV physical/virtual resources which can be used for learning by machine learning algorithms on such VNF deployment and SFC routing path decisions. The models are working on frameworks such as AutoML and TensorFlow, and their outputs of classification or prediction for optimal decisions are reflected through OpenStack Tacker (NFVO) or ONOS. Related research topics cover reinforcement learning and live migration.

2. **Development of Traffic Engineering based on Artificial Intelligence**

   Funded by Samsung (2020 – 2022)

   This research project aims to develop the artificial intelligence (AI)-based fast routing algorithm using network status information monitored in the segment routing environment. This algorithm not only satisfies service requirements (e.g., throughput, latency and packet loss) but also handles network failures.

My contribution to this project is development of a network topology simulator and survey on reinforcement learning-based routing.

3. **Development of Core Technologies for Programmable Switch in Multi-Service Networks**

   Funded by Institute for Information & Communications Technology Promotion (IITP) (2017 – 2020)

   This project aims to support multi-services network based on programmable switches. Research goal focus on extending P4 languages and defining new switch machine model, compiler, and multi-services network architecture. My contribution to this project is to survey knowledge-defined networking and design the overall knowledge plane to interwork with the INT-based network monitoring/control system. I am also studying P4 and intent-based networking since this project.

4. **Korea-US Collaborative Research on SDN/NFV Security/Network Management and Testbed Build**

   Funded by Institute for Information & Communications Technology Promotion (IITP) (2015 – 2017)

   This research project aims to study SDN/NFV based WAN network stability/service management, and construct SDN/NFV based WAN testbed between Korea and US. My contribution to this project is to develop an application-aware traffic engineering system using ONOS. The traffic engineering system identifies application traffic by DPI. To forward traffics to the DPI node and receive DPI result, I implemented REST APIs in ONOS. I also developed a traffic engineering application running on top of ONOS to allocate different bandwidth for each identified application traffic. During the project, I studied the overall architecture of ONOS and the application framework.

5. **Development of Smart Mediator for Mashup Service and Information Sharing among ICBMS Platform**

Funded by Institute for Information & Communications Technology Promotion (IITP) (2015 – 2018)

This R&D project aims to develop a smart mediator to support development of mashup services by interconnecting ICBMS (IoT, Cloud, Big data, Mobile, Security) platforms. My contribution to this project is to design and implement the core modules including message router, IoT adapter, big data adapter, and web dashboard for mashup service use cases (e.g., building management system). The message router supports the development of mashup services by providing useful functions such as service chaining, and routing required by developers in cooperation with various platforms according to the request messages transmitted through OpenAPIs. The IoT adapter supports interoperability with IoT platforms. The big data adapter interoperates with data analysis platforms. I worked as a lead developer for the implementation of the smart mediator including JavaScript-based web development using Node.js and AngularJS, and mashup service development.

Publications: International Journal Papers

1. Lee, D. Y., **Jeong, S. Y.**, Ko, K. C., Yoo, J. H., & Hong, J. W. K. "Deep Q-network-based auto scaling for service in a multi-access edge computing environment", International Journal of Network Management (SCIE), e2176.

2. Stanislav Lange, Nguyen Van Tu, **Se-Yeon Jeong**, Do-Young Lee, Hee-Gon Kim, Jibum Hong, Jae-Hyoung Yoo, James Won-Ki Hong, "A Network Intelligence Architecture for Efficient VNF Lifecycle Management", IEEE Transactions on Network and Service Management (TNSM) (SCIE), August 2020.

Publications: International Conference Papers

1. **Seyeon Jeong**, Tu Van Nguyen, Jae-Hyoung Yoo, James Won-Ki Hong, "Proactive Virtual Network Function Live Migration using Machine Learning", 17th International Conference on Network and Service Management (CNSM 2021), Izmir, Turkey, Oct. 25-29, 2021.

2. Stanislav Lange, Heegon Kim, **Seyeon Jeong**, Heeyoul Choi, Jae-Hyoung Yoo, James W. Hong, "Predicting VNF Deployment Decisions under Dynamically Changing Network Conditions", 15th International Conference on Network and Service Management (CNSM 2019), Halifax, Canada, Oct. 21-25, 2019.

3. **Seyeon Jeong**, Heegon Kim, Jae-Hyoung Yoo, James W. Hong, "Machine Learning Based Link State Aware Service Function Chaining", The 20th Asia-Pacific Network Operations and Management Symposium (APNOMS 2019), Matsue, Japan, Sep. 18-20, 2019.

4. Stanislav Lange, Heegon Kim, **Seyeon Jeong**, Heeyoul Choi, Jae-Hyoung Yoo, James W. Hong, "Machine Learning-based Prediction of VNF Deployment Decisions in Dynamic Networks", The 20th Asia-Pacific Network Operations and Management Symposium (APNOMS 2019), Matsue, Japan, Sep. 18-20, 2019.

5. Heegon Kim, **Seyeon Jeong**, Doyoung Lee, Heeyoul Choi, Jae-Hyoung Yoo, James W. Hong, "A Deep Learning Approach to VNF Resource Prediction using Correlation between VNFs", 2nd International Workshop on Emerging Trends in Softwarized Networks (ETSN 2019), Paris, France, June 28, 2019.

6. Heegon Kim, Doyoung Lee, **Seyeon Jeong**, Heeyoul Choi, Jae-Hyoung Yoo, James W. Hong, "A Machine Learning-based Method for Virtual Network Function Resource Demand Prediction", 5th IEEE Conference on Network Softwarization (Netsoft 2019), Paris, France, June 24-28, 2019.

7. **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “Design and Implementation of Virtual TAP for SDN-based OpenStack networking”, 16th IFIP/IEEE International Symposium on Integrated Network Management (IM 2019), Washington DC, USA, Apr. 8-12, 2019, pp. 233-241.

8. Jibum Hong, **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “Design and Implementation of eBPF-based Virtual TAP for Inter-VM Traffic Monitoring”, 1st International Workshop on High-Precision Networks Operations and Control (HiPNet 2018), Rome, Italy, Nov. 9, 2018, pp. 402-407.

9. **Seyeon Jeong**, Doyoung Lee, James Won-Ki Hong, “OpenFlow-based Virtual TAP using Open vSwitch and DPDK”, 16th IEEE/IFIP Network Operations and Management Symposium (NOMS 2018), Taipei, Taiwan, April 23-27, 2018.

10. Doyoung Lee, **Seyeon Jeong**, James Won-Ki Hong, “OpenAPI-based Message Router for Mashup Service Development”, 19th Asia-Pacific Network Operations and Management Symposium (APNOMS 2017), Seoul, Korea, Sep. 27-29, 2017, pp. 94-99.

11. **Seyeon Jeong**, Doyoung Lee, Jonghwan Hyun, Jian Li, James Won-Ki Hong, “Application-aware Traffic Engineering in Software-Defined Network”, 19th Asia-Pacific Network Operations and Management Symposium (APNOMS 2017), Seoul, Korea, Sep. 27-29, 2017, pp. 315-318.

12. Doyoung Lee, **Seyeon Jeong**, Taeyeol Jeong, Jae-Hyoung Yoo, James Won-Ki Hong, “ICBMS SM: A Smart Mediator for Mashup Service Development”, 18th Asia-Pacific Network Operations and Management Symposium (APNOMS 2016), Kanazawa, Japan, Oct. 5-7, 2016, pp. 1-6.

13. **Seyeon Jeong**, Doyoung Lee, Junemuk Choi, Jian Li, James Won-Ki Hong, “Application-aware Traffic Management for OpenFlow Networks”, 18th Asia-Pacific Network Operations and Management Symposium (APNOMS 2016),

Kanazawa, Japan, Oct. 5-7, 2016, pp. 1-5.

Publications: Domestic Journal Papers

1. **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “Proactive Virtual Network Function Live Migration using Machine Learning”, KNOM Review vol. 24, No. 1, August 2021.

2. Jibum Hong, **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “Design and Implementation of eBPF-based Virtual TAP for Inter-VM Traffic Monitoring ”, KNOM Review, Vol. 21, No. 2, December 2018.

3. Jian Li, Doyoung Lee, **Seyeon Jeong**, James Won-Ki Hong, “A study on a Distributed Open Source SDN Controller – ONOS for Service Provider Network”, Korean Institute of Communications and Information Sciences (KICS) Information Communications Magazine, Vol. 34, No. 12, Dec. 2017, pp. 10-19.

4. **Seyeon Jeong**, Jibum Hong, James Won-Ki Hong, “Design of Virtual TAP for Traffic Monitoring in Server Virtualization Environment”, KNOM Review, Vol. 20, No. 1, August 2017, pp. 1-8.

5. Doyoung Lee, **Seyeon Jeong**, Jonghwan Hyun, Jian Li, James Won-Ki Hong, “Application-aware Traffic Engineering in SDN”, KNOM Review, Vol. 19, No. 2, Dec.2016, pp. 1-12.

6. **Seyeon Jeong**, Doyoung Lee, Jian Li, Jae-Hyoung Yoo, James Won-Ki Hong, “A Study of Control Plane Monitoring and Switch Placement Scheme in SDN Multi-controller Environment”, KNOM Review, Vol. 18, No. 1, Aug.2015, pp. 11-24.

Publications: Domestic Conference Papers

1. **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “Dynamic Service Placement in Multi-access Edge Computing using Reinforcement Learning-based Live Migration”, KNOM Conference 2022, Chuncheon, Korea, May 12, 2022, pp. 3-6.

2. **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “A VNF Live Migration method based on Server Anomaly Prediction using Machine Learning”, KNOM Conference 2021, On-line KNOM Conference Venue, Korea, April 30, 2021, pp. 9-12.

3. **Seyeon Jeong**, Doyoung Lee, Jae-Hyoung Yoo, James Won-Ki Hong, “Design of AI-based NFV Management System and Testbed Construction”, KNOM Conference 2019, Daegu, Korea, May. 31, 2019, pp. 40-42.

4. Heegon Kim, **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “A Study on Machine Learning based VNF Resource Prediction”, KNOM Conference 2019, Daegu, Korea, May. 30, 2019, pp. 78-79.

5. **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “A Study on Machine Learning based VNF Resource PredictionDesign and Implementation of Virtual TAP for SDN-based OpenStack Networking”, KNOM Workshop 2018, Seoul, Korea, Nov. 30, 2018, pp. 20-21.

6. **Seyeon Jeong**, Doyoung Lee, Ga-Yeon Kim, Jae-Hyoung Yoo, James Won-Ki Hong, “Design of Monitoring Framework for NFV Management based on Machine Learning”, KNOM Conference 2018, Jeju, Korea, May 10-11, 2018, pp. 3-4.

7. Doyoung Lee, **Seyeon Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, “Design and Requirements of Artificial Intelligence-based NFV Management Platform”, KNOM Conference 2018, Jeju, Korea, May 10-11, 2018, pp. 51-52.

8. Doyoung Lee, **Seyeon Jeong**, James Won-Ki Hong, “A Study on OpenAPI based Message Router for Mashup Service Development”, KNOM Conference 2017, Gwangju, Korea, June 2-3, 2017, pp. 19-20.

9. **Seyeon Jeong**, Doyoung Lee, June-Muk Choi, James Won-Ki Hong, “SDN Traffic Management System using DPI”, KNOM Conference 2016, Chuncheon, Korea, May 12-13, 2016, pp. 6-10.

10. Doyoung Lee, **Seyeon Jeong**, Tae-Yeol Jeong, James Won-Ki Hong, “A Study on Smart Mediator for Integration of ICBMS Platforms and Development of Mashup Services”, KNOM Conference 2016, Chuncheon, Korea, May 12-13, 2016, pp. 71-75.

Domestic Patent Application

1. James Won-Ki, Jae-Hyoung Yoo, Doyoung Lee, **Seyeon Jeong**, “METHOD FOR DUPLICATING PACKET AND APPARATUS THEREOF”, Application No.: 10-2019-0160735, 2019.12.05 (Applicant: POSTECH) (pending).

2. James Won-Ki Hong, **Seyeon Jeong**, Doyoung Lee, “METHOD AND APPARATUS FOR IMPLEMENTING VIRTUAL TEST ACCESS POINT”, Application No.: 10-2018-0012705, 2018.02.01 (Applicant: POSTECH) (pending).

3. James Won-Ki Hong, **Seyeon Jeong**, Doyoung Lee, “APPARATUS AND METHOD FOR CONTROLLING TRAFFIC IN OPENFLOW NETWORK”, Application No.: 10-2017-0115036, 2017.09.08 (Applicant: POSTECH) (pending).

Work Experience

Open Networking Foundation

Research Intern

Jan. 2020 – Jul. 2020

1. Development of an SDN troubleshooting application that dumps SDN/OpenFlow network information from cache and pinpoints problematic flow rules.

2. Development of extending the troubleshooting application to work for P4-based programmable switch.

Awards

Title	Organizations	Date
Best Paper Award	KNOM Conference 2022	2021.05
Best Paper Award	KNOM Conference 2021	2021.04
Best Paper Award (3rd author)	APNOMS 2019	2019.09
Best Paper Award	KNOM Conference 2019	2019.05
IEEE Student Grant Award	IFIP/IEEE IM 2019	2019.04
Hackathon Winner	Open Network Korea (ONK) 2017	2017.11
Best Paper Award (2nd author)	APNOMS 2017	2017.09
Hackathon Winner	ONOS Build 2017	2017.09
Best Paper Award	KNOM Conference 2016	2016.05

Teaching Assistance

Title	Courses	Date
Teaching Assistant (Dept. of CSE, POSTECH)	CSED353: Computer Networks	2021 Spring
	CSED353: Computer Networks	2019 Spring
	POSTECH MOOC: OpenStack	2019 Spring
	POSTECH MOOC: SDN-NFV	2018 Spring
	CSED499: Senior Research Projects	2016 Fall

References

Prof. James Won-Ki Hong

Department of Computer Science and Engineering

Pohang University of Science and Technology, Pohang, Korea

Email: jwkhong@postech.ac.kr

Prof. Jae-Hyoung Yoo

Department of Computer Science and Engineering

Pohang University of Science and Technology, Pohang, Korea

E-mail: jhyoo78@postech.ac.kr