BigMEC: Scalable Service Migration for Mobile Edge Computing

Florian Brandherm
Telecooperation Lab
TU Darmstadt
Germany
brandherm@tk.tu-darmstadt.de

Julien Gedeon
Telecooperation Lab
TU Darmstadt
Germany
gedeon@tk.tu-darmstadt.de

Osama Abboud
Huawei Technologies
Munich, Germany
osama.abboud@huawei.com

Max Mühlhäuser
Telecooperation Lab
TU Darmstadt
Germany
max@tk.tu-darmstadt.de

***Abstract*— The proximity of Mobile Edge Computing offers the potential for offloading low latency closed-loop applications from mobile devices. However, to repair decreases in quality of service (QoS), e.g., resulting from user mobility, the placement of service instances must be continually updated – essential for mission critical applications that cannot tolerate decreased QoS, for example virtual reality or networked control systems. This paper presents BigMEC, a decentralized service placement algorithm that achieves scalable, fast, and high-quality placements by making local service migration decisions immediately when a drop in QoS is detected. The algorithm relies on reinforcement learning to adapt to unknown scenarios and to approximate long-term optimal placement updates by taking future transition costs into account. BigMEC limits each decentralized migration decision to nearby edge sites. Thus, decision computation times are independent of the number of nodes in the network and well below 10ms in our experimental setup. Our ablation study validates that, using its scalable approach to decentralized resource conflict resolution, BigMEC quickly approaches optimal placement with increasing local view size, and that it can reliably learn to approximate long-term optimal migration decisions, given only a black-box optimization objective.**

***Index Terms*—mobile edge computing, service migration, reinforcement learning, distributed algorithms**

I. Introduction

Offloading computation and storage for closed-loop applications from mobile devices to cloud services offers much potential to conserve devices' production cost, battery power, and weight [1]. Examples of such applications are Virtual/Augmented Reality (VR/AR) [2], wearable cognitive assistance [3], networked control systems [4], self-driving cars [5], or video analytics for drones [6].

Despite substantial latency reductions in recent years, centralized cloud data centers may still be unable to deliver the required reliable low latency for this type of application. The reason is their physical distance to clients [7]. To offer substantially lower latency to mobile users, there are currently large efforts to augment cloud data centers with a network of geographically distributed micro data centers at the edge of the Internet, e.g., at 5G access points [8]. This computing paradigm is referred to as Mobile Edge Computing (MEC).

A major challenge in MEC is to decide which edge site processes which client's services, as the relative inelasticity of individual edge sites demands smart and location-aware load balancing to resolve resource conflicts [9]. Further, to maintain a high quality of service (QoS) for mobile users, the placement of service instances must be continually updated [10] through service migration.

Nevertheless, what ideal resource allocation and service migration means and how it is achieved depends on the nature of the edge application. Similar to the cloud, where a myriad of different service models are offered for different use cases, a wide variety of applications for MEC has been proposed with vastly different requirements [11]. Thus, a variety of service models and placement/migration schemes have been proposed.

Some works investigated stateless systems, where computations are scheduled on a request-by-request-basis. For example Urgaonkar et al. [12] and Ma et al. [9] proposed mechanisms that route requests either to an edge site, or to the cloud, while simultaneously optimizing which applications are deployed on which edge node. This deployment paradigm is related to cloud-based serverless frameworks.

However, as noted by Hellerstein et al., the current serverless offerings suffer from poor latency for storage access [13]. The inelasticity and decentralization of MEC likely exacerbate storage access issues [9]. Thus, the request-by-request approach is not suitable for all types of applications, such as applications that need dedicated resources or hold client-specific state to provide continuous closed loop services.

Therefore the remainder of this article focuses on the placement and migration of continuous client-specific service instances, an approach that is followed by a large number of works on service placement in MEC.

A. Overview of Existing Service Migration Methods

As many authors have recognized, long-term optimal migration decisions are advantageous in MEC because of clients' mobility. An optimal placement configuration for a given point in time might not be optimal in the long term. Consider, for example, an autonomous car driving along a freeway. If the objective is to minimize latency, then the optimal placement location of its MEC service instances is not necessarily always the edge site with the lowest round trip time. In reality, there are migration costs that have to be factored in, e.g., a short service interruption or the transmission of large amounts of state data from one edge site to the next. Thus, such a naïve

strategy might not be optimal in the long term if the trade-off between all future costs and benefits is taken into account. A better strategy may be to migrate the service instances further ahead in the direction of travel, and thus, lower the required number of migrations. Future costs are much more substantial in edge clouds than in centralized data centers, where migrations are more predictable and migration costs are much lower due to the fast interconnections.

Most works considering this need for anticipatory migration decisions have used Markov Decision Processes (MDPs) to optimize placement decisions for the long term. Some works solved the MDPs model-based, using provable online placement algorithms [12], [14], or analytical solutions [15]. However, to become feasible, such solutions typically require relaxations of the optimization objective [12] or very simplifying assumptions about the system [15].

However, to compute model-based solutions to MDPs without such problem simplifications requires good predictive models for client mobility and network behavior. Therefore, similar to related fields, such as job scheduling for the cloud [16], [17], reinforcement learning has recently been embraced for its ability to solve MDPs in a model-free way, adapting to client mobility and other network events while treating them as a black box. Model-free, learning approaches have the additional advantage that they are more reusable because it's easy to change, e.g., the optimization objective (also treated as a black box), without needing a manual redesign of the optimization algorithm. Such flexibility is an important property, since the objectives of service placement in upcoming MEC deployments may be changing rapidly until established infrastructure and business models exist.

One example of such a reinforcement learning based approach was given by Tang et al., where a reinforcement learning based policy determines which container should migrate to which node [18]. However, while effective in small scenarios, this approach scales badly, as the entire network state is used as input for the neural network.

Many of these service migration mechanisms rely on centralized decision making [14], [15], [19], [20]. However, service placement and migration are generally NP-hard. While the proposed service migration mechanisms scale much better than exact solutions, none of these mechanisms can scale to large numbers of edge nodes with constant or near-constant placement computation time. Besides the algorithmic complexity, the communication with far away edge sites can hamper the decision speed as sites' distances from the central placement decision maker increase.

These scalability issues are solved using decentralized service migration mechanisms. For example, the reinforcement learning based approach by Zhang and Zheng uses an MDP formulation that would be highly scalable if deployed distributedly [21]. However, this is achieved by making decisions for individual services with very limited input data and disregarding any resource conflicts. Gao et al. proposed to decide for each service instance, where to migrate, disregarding any nodes that are fully occupied [22]. Similarly, Brandherm et al. used a single agent reinforcement learning method in a multi-agent scenario [23], where each agent's view of the network is limited to its neighborhood. Although this approach is also highly scalable, the used learning method tends to be unstable in multi-agent settings. The above three mechanisms resolve resource conflicts between service instances on a "first come, first served" basis. Yet, such non-prioritizing resource conflict resolution can lead to degraded overall placement quality if resources are saturated. This issue is further illustrated in Section II-C and we propose a novel resource conflict resolution mechanism for decentralized service migration in Section II-D.

Alternatively, some decentralized service migration mechanisms with consideration for resource conflicts between services have been proposed. One example is the work of Abouaomar et al., where each MEC node uses a learned policy to decide which services to host; a central node then resolves any conflicts [24]. While the first part of this approach is fully distributed, the centralized conflict resolution limits its scalability. Another notable example is the work by Liu et al., that introduced a multi-agent-learning-based placement mechanism that, after training, allows each node to make independent local decisions in constant time [25]. Nevertheless, the potential for scalability is still not fully realized, as the used learning algorithm, COMA [26], requires globally synchronous migration decisions and a global state snapshot to compute a baseline during training. Thus, resolving resource conflicts between services in fully decentralized service placement for MEC has remained a hurdle toward high scalability. In Sections II-D and III-B, we propose a learning, asynchronous approach that solves these remaining scalability issues by brokering resource conflicts locally, without the need to learn this multi-agent interaction.

B. Contributions

This paper studies learning decentralized service migration mechanisms for offloaded services in MEC and presents BigMEC, a novel decentralized migration mechanism with three key innovations:

a) Local pre-selection of migration destinations: By limiting each migration decision to a local pre-selection of edge sites, BigMEC makes learning decentralized service migration strategies feasible for large-scale MEC infrastructures (Section II-B). In Section IV-A, we show that this approach can reach close to optimal solutions, despite considering only a fraction of all edge sites for each decision.

b) Decoupling learning and agent-agent-interaction: BigMEC addresses the stability issues of single-agent reinforcement learning in multi-agent scenarios through a separation of concerns: The learned policy deliberately ignores all other agents, assuming a single-agent scenario that can be learned reliably by single-agent reinforcement learning methods (Section III-B). Resource conflicts between service instances are then brokered decentrally by an agent interaction policy that approximates a globally optimal trade-off between the agents' goals.

Table I
LIST OF KEY SYMBOLS AND NOTATIONS.

Symbol	Description
$\square^{(i)}$	state of $\square$ at time step i
$G^{(i)}$	graph of the underlay network
$\kappa \in K$	edge site
$\sigma \in \Sigma^{(i)}$	service
$x^{(i)}_{\sigma,\kappa} \in \{0,1\}$	placement variable; 1 if, and only if σ at κ
κ_σ	current edge site of service instance σ (i.e, $x^{(i)}_{\sigma,\kappa_\sigma} = 1$)
c^{req}_σ	required number of CPU cores of service σ
$c^{\max}_\kappa$ / c^{avl}_κ	maximum/available CPU cores at edge site κ
$\mathcal{N}^{(i)}_\sigma \subseteq K$	heuristic; probable migration targets for σ
$s_\sigma = \bigcup_{\kappa \in \mathcal{N}^{(i)}_\sigma} s^\kappa_\sigma$	locally observable state information for σ
a^κ_σ	migration action: service σ to cloudlet κ
$Q(s^\kappa_\sigma, a^\kappa_\sigma)$	utility of executing a^κ_σ, given s^κ_σ
$\mathbf{a} \in \mathcal{A}$	migration action sequence $(a^\kappa_\sigma, a^{\kappa'}_{\sigma'}, \dots)$
$\pi(\mathcal{S}_\mathcal{N}) \to \mathcal{A}$	local service migration algorithm
$C^{\text{free}}_\kappa(c)$	utility cost of freeing c CPU cores from κ
$C^{\text{disp}}_\kappa(a^\kappa_\sigma)$	utility cost of displacing service σ to site κ
D_κ	set of all possible displacement actions of services at κ
$D^{\min}_\kappa \subseteq D_\kappa$	set of displacement actions with lowest combined $C^{\text{disp}}_\kappa(a^{\kappa'}_\sigma)$ that frees at least c^{free} memory from κ; output of BigMEC
$\Delta Q(s^\kappa_\sigma, a^\kappa_\sigma)$	global utility gain of executing a^κ_σ and the displacement actions $D^{\min}_\kappa$ to free space at κ

c) Displacement Mechanism: As the agent interaction policy, BigMEC implements a novel decentral service displacement mechanism (Section II-D) that allows to prioritize the assignment of computation resources to service instances and demonstrably (see Section IV-A) avoids the degradation of placement decision quality in saturated situations in which few or no resources are available.

II. DECENTRALIZED SERVICE PLACEMENT AND MIGRATION

A. System Model

We model an edge network as a graph of network nodes. Some nodes host an edge site which can each serve a limited number of service instances.

a) Network: We model the physical network at a point in time $t^{(i)}$ as a graph $G^{(i)} = (V^{(i)}, E^{(i)})$. The nodes $V^{(i)} = U^{(i)} \cup K \cup N$ of the underlay network are composed of a set of mobile clients $U^{(i)}$ (e.g., mobile phones, IoT devices, or connected cars), a set of nodes with edge sites K, and a set of nodes without edge sites N. The undirected edges $E^{(i)} = \{\langle u, v\rangle|_{u,v \in V}\}$ describe the network topology at time $t^{(i)}$. The index $i \in \mathbb{N}$ enumerates time steps. For conciseness, we omit the notation $\square^{(i)}$ for variables that change over time whenever possible.

Because mobile clients can join, leave, or change their access point, the graph $G^{(i)}$ changes over time. We model these changes as atomic transitions from a graph $G^{(i)}$ to $G^{(i+1)}$, where exactly one of the following operations occurs: a client (i) joins, (ii) leaves, or (iii) changes its access point.

b) Service Placement: Let $\Sigma^{(i)}$ denote the set of service instances at time $t^{(i)}$. In an atomic transition $\Sigma^{(i)}$ to $\Sigma^{(i+1)}$, a service instance can either be created or destroyed.

To specify at which edge site each service instance is placed, we introduce a placement variable $x^{(i)}_{\sigma,\kappa} \in \{0,1\}$ for each combination of service instance $\sigma \in \Sigma$ and edge site $\kappa \in K$. We define $x^{(i)}_{\sigma,\kappa} = 1$ if and only if service instance σ is placed at edge site κ. The entire placement configuration at time $t^{(i)}$ can be summarized as matrix $\mathcal{X}^{(i)} \in \{0,1\}^{|\Sigma^{(i)}| \times |K|}$.

c) Placement Constraints: A service instance can only be placed on a single edge site and every service instance needs a set of dedicated resources, e.g., CPU cores, memory, or GPUs. For simplicity and without loss of generality, we limit ourselves to a single resource: the required number of CPU cores $c^{\text{req}}_\sigma \in \mathbb{R}^+$ (non-integer values allowed). The combined CPU requirements of all instances at an edge site must not exceed the available number of cores $c^{\max}_\kappa \in \mathbb{N}$. In other words, the constraints

$$\forall \sigma, \sum_{\kappa \in K} x^{(i)}_{\sigma,\kappa} = 1, \tag{1}$$

$$\forall \kappa, \sum_{\sigma \in \Sigma} x^{(i)}_{\sigma,\kappa} \cdot c^{\text{req}}_\sigma \leq c^{\max}_\kappa \tag{2}$$

must always hold for any valid placement configuration. It is straightforward to extend this model to multiple types of resources, e.g., memory, by introducing additional constraints, analogous to the CPU constraint (2). We use the notation c^{avl}_κ for the number of available CPU cores at edge site κ.

d) Service Migration: The placement quality of a service instance can deteriorate due to client mobility or varying network load. For example, the client could change its access point, increasing network latency. Thus, whenever such a change is detected for a service instance, e.g., through QoS monitoring, a migration decision to re-evaluate its placement is triggered immediately. Hence, the time $t^{(i+1)} - t^{(i)}$ between consecutive time steps i and $i+1$ is not fixed, but is the time between two events that trigger service migration decisions — migration decisions happen asynchronously. When triggered, a service migration algorithm $\pi(\mathcal{S}) \to \mathcal{A}$ computes a sequence of migration actions $\mathbf{a} = (a^\kappa_\sigma, a^{\kappa'}_{\sigma'}, \dots) \in \mathcal{A}$, based on the current state $\mathbf{s}^{(i)} = (G^{(i)}, \Sigma^{(i)}, \mathcal{X}^{(i)}) \in \mathcal{S}$. In this notation, a^κ_σ denotes the migration of service σ from its current edge site to edge site κ (if both sites are identical, no migration is performed). All migration actions $a^\kappa_\sigma, a^{\kappa'}_{\sigma'}, \dots$ are sequentially applied to the placement configuration $\mathcal{X}^{(i)}$. After each migration, the placement constraints (1) and (2) must hold, otherwise the sequence is invalid. After applying all migration actions, time step i is followed by step $i+1$ with the next state $\mathbf{s}^{(i+1)} = (G^{(i+1)}, \Sigma^{(i+1)}, \mathcal{X}^{(i+1)})$ at the time $t^{(i+1)}$ of the next event that triggers a migration decision.

B. Decentralized Service Migration

The goal of our approach is scalability, i.e., to make good service migration decisions quickly, regardless of the total number of edge sites. We achieve this through decentralized decision making, where each edge site is responsible to make

migration decisions on behalf of its hosted service instances. Thus, each edge site's migration decisions can be made locally, which allows confining each decision to a local neighborhood of edge sites (which may be defined in different ways). This also means that migration decisions for an individual service can be triggered locally, immediately when a change in QoS is detected – migration decisions happen asynchronously.

The idea of confining migration decisions to nearby edge sites stems from two observations: First, proximity is already a good heuristic for service placement in edge computing. Second, local service migrations are unlikely to strongly impact further away service instances.

The principle of confining decentralized migration decisions to a bounded set of nearby edge sites was introduced in our previous work [23], where the possible edge sites to migrate a service to were confined to the 10 closest sites in the network. However, we found that this heuristic performs poorly for the star topology typical of access networks, for which graph distance is a poor approximation of geographical distance. Therefore, we generalize the approach into confining migration decisions to a set of candidate edge sites that can be selected ad-hoc, based on the current context of the service instance to be migrated, e.g., by considering the client's current location.

A heuristic selects a set $\mathcal{N}_\sigma^{(i)} \subseteq K$ of candidate edge sites as possible migration destinations for service instance σ. This candidate selection is performed anew for every service migration decision, based on the state of the service instance, its client, and its edge site. Ideally, this candidate set contains the optimal migration destination with high probability.

Undoubtedly, a good heuristic for candidate sites aligns well with the given optimization objective. For example, if the goal is low latency, a good heuristic selects a set of edge sites with the lowest expected latency to the service's client. The heuristic we have used in our experiments, denoted as $\mathcal{N}_{\sigma,n}^{(i)} \subseteq K$, always includes the current site of the service instance (always allowing instances to not migrate), a central cloud node (always allowing instances to migrate away), and the $n-1$ cloudlets with the fewest number of hops from the client's current base station. If not otherwise stated, $n = 10$; an investigation into the choice of n follows in Section IV-A.

We could imagine other heuristics to choose the set of candidate sites $\mathcal{N}_\sigma^{(i)}$, but we leave their exploration to future works. However, it is worth noting that the choice of $\mathcal{N}_\sigma^{(i)}$ can also be used to enforce hard placement constraints, e.g., by excluding all edge sites whose latency exceeds a threshold.

a) Decentralized Migration Decisions: Due to the selection of a small set of suitable migration destinations, the decision where to migrate a service instance only depends on this small set of edge sites. In other words, migration decisions of individual edge sites only depend on a subset s_σ of the global state $\mathbf{s}^{(i)}$ that is visible within the set of nearby candidate edge sites $\mathcal{N}_\sigma^{(i)}$.

Thus, the decentralized service migration policy $\pi(\mathcal{S}_{\mathcal{N}^{(i)}}) \rightarrow \mathcal{A}$ only depends on the observable state $s_\sigma = \bigcup_{\kappa \in \mathcal{N}_\sigma^{(i)}} s_\sigma^\kappa \in \mathcal{S}_{\mathcal{N}^{(i)}}$ within the candidate set $\mathcal{N}_\sigma^{(i)}$.

Algorithm 1 Decentralized Greedy Algorithm

```
# Trigger: service instance σ affected by QoS change.
# Executed on κ_σ, the current edge site of σ:
function GREEDY(σ, κ_σ)
    for all κ ∈ N_σ^(i) \ κ_σ do
        if c_κ^avl ≤ c_σ^req then
            s_σ^κ ←(from κ)— GETSTATE(κ, σ)
    # greedy migration decision:
    κ^dst ← argmax_{κ∈N_σ^(i)} Q(s_σ^κ, a_σ^κ)
    execute migration action a_σ^{κ^dst}
```

Therefore, only a small, bounded subset of the system's state, which is available nearby, is required for each migration decision—making the computation and communication time of each migration decision bounded and independent of the overall size of the network.

b) Migration Action Utility: Out of the set of candidate edge sites to migrate a service instance to, the most advantageous site must be chosen. To decide which migration action is best in a particular situation, a utility function $Q(s_\sigma^{\kappa(i)}, a_\sigma^{\kappa(i)})$, quantifies the utility of migrating service instance σ from its current edge site to candidate site $\kappa \in \mathcal{N}_\sigma^{(i)}$. If κ refers to the current site of σ, it quantifies the utility of not migrating. The utility depends on (i) the migration action in question $a_\sigma^{\kappa(i)}$ (i.e., the destination edge site κ), and (ii) the locally observable state $s_\sigma^{\kappa(i)}$ of the service instance σ (including its client). The utility can be thought of as the opposite of a cost function and captures the overall long-term costs and benefits of the migration execution and the new placement location. For now, we assume the utility function $Q(s_\sigma^{\kappa(i)}, a_\sigma^{\kappa(i)})$ to be given. However, it is much easier to define myopic cost functions that assess only the value of migrations/non-migrations at the current instant of time, than creating a good model for their long-term values, considering all possible future, yet unknown, events. Therefore, in Section III, we propose using reinforcement learning to automatically derive good long-term utility functions from easy to hand-define myopic cost functions.

The optimization objective at each step is to maximize the summed utilities of all service migration actions (including the utilities of service instances that were not migrated). Nevertheless, computing the optimal global solution is an NP-hard combinatorial optimization problem and is only feasible in a centralized architecture that has a global view. Hence, we only approximate the optimization objective.

C. Decentralized Greedy Algorithm

A naïve approach to such a decentralized service migration algorithm is to always greedily migrate service instances to the edge site that yields the highest utility (see Algorithm 1). However, as in similar previous approaches [21], [23], service instances cannot be migrated to edge sites with insufficient free resources, due to the lack of a mechanism to resolve resource conflicts. Although this greedy algorithm meets the scalability

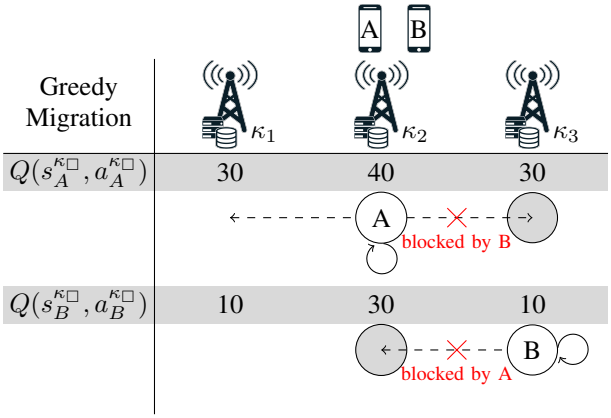


Figure 1. If all good migration destination sites are occupied, service instances cannot migrate in the greedy algorithm. To maximize utilities globally, instance A should migrate to κ_1 to free space for the more important instance B. Yet, there is no incentive to do so from the local view of either instance.

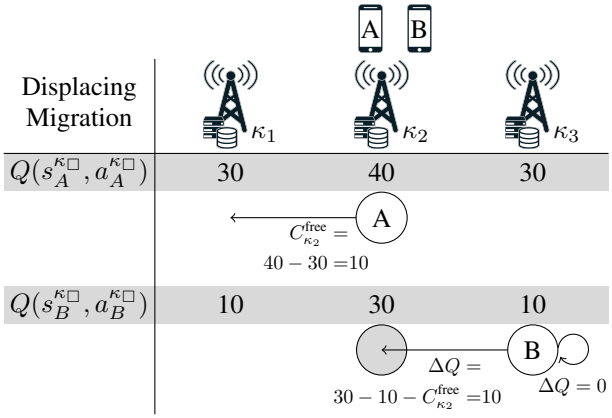


Figure 2. In the greedy algorithm, the presence of instance A prevented service instance B from migrating to κ_2 which is globally suboptimal. However, including $C^{\text{free}}_{\kappa_2}$, the global gain in utility ΔQ of migrating A to κ_1 and B to κ_2 is higher than not migrating B (not displacing A).

goal, allocating resources on a first come, first served basis causes two problems that lead to placement inefficiencies:

a) Resource Saturation: If demand in an area exceeds the available MEC resources, the system becomes locally saturated, meaning that little to no unoccupied resources are available nearby. Hence, service instances' migration candidate sets $\mathcal{N}_\sigma^{(i)}$ do not contain any edge sites that can accommodate it besides the current site. Thus, service instances can become immobile, as illustrated in Figure 1. Localized saturation is likely to occur in dense MEC deployments, e.g., in flash crowd events, or if an area is under-provisioned. Although including a virtually unlimited cloud in every $\mathcal{N}_\sigma^{(i)}$ always enables migrating away, the QoS characteristics of any edge site are likely to be preferred to those of a central cloud. Thus, if (i) there are no free resources within the local area at the edge, and (ii) migrating to the cloud is not advantageous for any service instance, no service instance has an incentive to migrate and all service instances are immobilized until a client leaves the saturated area.

b) Lack of Resource Prioritization: MEC services differ in their performance objectives and their importance [11]. We encode this in the utility function. For example, a safety-relevant service for autonomous vehicles requires lower latency and higher availability than a service that improves advertisement delivery, and thus, obtains a much higher utility from edge sites. Nevertheless, the greedy approach has no mechanism to assign resources to those services that gain the highest utility from them in the greedy approach. Instead, resources are allocated on a first come, first served basis. Allowing service instances to displace instances that profit less from their occupied resources is the intuitive solution to this problem. However, this is not trivial, since it might not be worth displacing a service instance from the global perspective, e.g., if displacing an instance with lower utility would incur a substantial migration cost to it and the alternative is to migrate to an unoccupied site with slightly worse utility.

D. BigMEC: Displacing Service Instances

To eliminate the above inefficiencies of the greedy approach, we introduce BigMEC, a refinement of the above greedy algorithm that allows the displacement of other service instances if it increases the global sum of utilities. Algorithm 2 lists the complete algorithm that is described in the following. With the greedy algorithm, a migration destination that would lead to a large gain in utility for one service instance can be occupied by another instance that would only incur a small loss of utility by migrating away (see Figure 1). However, by considering the combination of these two migration actions together, (i.e., adding the large utility gain of the first service instance to the small utility loss of the second instance) the global sum of utilities can be improved.

We denote this gain in global utility $\Delta Q(s_\sigma^\kappa, a_\sigma^\kappa) = Q(s_\sigma^\kappa, a_\sigma^\kappa) - Q(s_\sigma^{\kappa_\sigma}, a_\sigma^{\kappa_\sigma}) - C_\kappa^{\text{free}}(c_\sigma^{\text{req}})$, where $Q(s_\sigma^\kappa, a_\sigma^\kappa) - Q(s_\sigma^{\kappa_\sigma}, a_\sigma^{\kappa_\sigma})$ is the gain in utility of migrating service instance σ from its current edge site κ_σ to another site κ, and $C_\kappa^{\text{free}}(c_\sigma^{\text{req}})$ is the minimum utility cost to free enough resources at the destination site κ. Thus, for a no-migration action, $\Delta Q(s_\sigma^{\kappa_\sigma}, a_\sigma^{\kappa_\sigma}) = 0$. We have depicted an example of this basic principle in Figure 2.

The set of all possible migration actions that must be considered for displacement from a destination site κ is $D_\kappa = \{a_\sigma^{\kappa'} | \sigma \in \Sigma_\kappa, \kappa' \in \mathcal{N}_\sigma^{(i)} \setminus \kappa\}$. In other words, it is necessary to determine the utility costs of displacing every service instance at the destination site κ to every one of its destination site candidates, in order to then determine the most advantageous combination of displacement actions $D_\kappa^{\min} \subseteq D_\kappa$ as follows.

The cost of an individual displacement action is $C_\kappa^{\text{disp}}(a_\sigma^\kappa) = Q(s_\sigma^{\kappa_\sigma}, a_\sigma^{\kappa_\sigma}) - Q(s_\sigma^\kappa, a_\sigma^\kappa)$, the difference in utility between leaving service instance σ at the current edge site κ_σ, and migrating it to the destination candidate κ. Finally, the minimum cost to free up c_σ^{req} CPU cores at site κ is $C_\kappa^{\text{free}}(c_\sigma^{\text{req}}) = \sum_{a_\sigma^{\kappa'} \in D_\kappa^{\min}} C^{\text{disp}}(a_\sigma^{\kappa'})$, the combined utility costs of all displacement actions in the set $D_\kappa^{\min}$.

However, potential resource conflicts between the service instances associated to displacement actions in D_κ complicate finding $D_\kappa^{\min}$. Given the required amount of CPU capacity to free c_σ^{req}, this leads to a constrained optimization problem

$$D_\kappa^{\min} = \underset{\hat{D}_\kappa \subseteq D_\kappa}{\operatorname{argmin}} \sum_{a_{\sigma'}^{\kappa'} \in \hat{D}_\kappa} \underbrace{Q(s_{\sigma'}^{\kappa}, a_{\sigma'}^{\kappa}) - Q(s_{\sigma'}^{\kappa'}, a_{\sigma'}^{\kappa'})}_{C^{\text{disp}}(a_{\sigma'}^{\kappa'})}, \tag{3}$$

$$\text{s.t.} \quad \forall \kappa \in K: \quad c_\kappa^{\max} \geq c_\kappa^{\text{avl}} + \sum_{a_{\sigma'}^{\kappa'} \in \hat{D}_\kappa} \delta_{\kappa,\kappa'} \cdot c_\sigma^{\text{req}} \tag{4}$$

$$\text{and} \quad c_\sigma^{\text{req}} \leq c_\kappa^{\text{avl}} - \sum_{a_{\sigma'}^{\kappa'} \in \hat{D}_\kappa} c_\sigma^{\text{req}}, \tag{5}$$

where the set of displacement actions with the lowest combined utility cost (Equation (3)) must be selected, subject to the following constraints. First (Equation (4)), no target edge site's CPU capacity can be exceeded after executing all displacement actions ($\delta_{\kappa,\kappa'} = 1$ if $\kappa = \kappa'$, else $\delta_{\kappa,\kappa'} = 0$). Second (Equation (5)), all displacement actions together must free enough CPU capacity (more than the required c_σ^{req})

This optimization problem is equivalent to a Knapsack problem and can be approximated by one of many approximation schemes. For simplicity, we chose to approximate this optimization problem with a greedy packing heuristic. First, the cost per CPU core $C_\kappa^{\text{disp}}(a_\sigma^{\kappa'})/c_\sigma^{\text{req}}$ is computed for each displacement action. Then, until there are no actions with negative cost per core left and the required amount c_σ^{req} is freed (see Equation (5)), the action with the lowest cost per core is added to $D_\kappa^{\min}$. However, an action is only added if it doesn't conflict with any action already in $D_\kappa^{\min}$ for resources (see Equation (4)). This strategy also ensures, that viable displacement actions with negative cost (i.e., displacement is actually desired), are always executed, even if that means freeing more resources than necessary. Lines 26 to 44 of Algorithm 2 summarize the above packing heuristic.

a) Extension to Multiple Types of Resources: As mentioned before, the system model can easily be extended to consider multiple types of resources, say, memory, storage, or GPUs. To extend BigMEC to multiple resource types, (i) constraints analogous to Equations (4) and (5) must be added to extend $C_\kappa^{\text{free}}(\cdot)$ to multiple resources, and (ii) a different packing heuristic (see line 26 to 40 of Algorithm 2) must be implemented, as the Knapsack problem becomes multidimensional.

The displacing behavior solves the efficiency problems of the greedy first come, first serve approach (see Section IV-A). Even though no service instance might benefit from migrating to the cloud in a saturated area, service instances can be forced to free edge resources for other instances that yield higher utility from them.

It seems natural to extend the algorithm with recursive displacement so displaced service instances can displace others themselves. However, recursion quickly becomes infeasible due to an explosion of potential resource conflicts: Conflicts arise not only between displaced service instances originating from one site, but also between instances originating from other simultaneous displacement processes. A recursive variant only appears feasible in two special cases that are not realistic in MEC scenarios: The first case is that all service instances require the exact same amount of

Algorithm 2 BigMEC

1: *# Trigger: service instance σ affected by QoS change.*
2: *# Executed on κ_σ, the current edge site of σ:*
3: **function** BIGMEC(σ, κ_σ)
4: **for all** $\kappa \in \mathcal{N}_\sigma^{(i)} \setminus \kappa_\sigma$ **do**
5: $C_\kappa^{\text{free}}, D_\kappa^{\min} \xleftarrow{\text{from } \kappa}$ FREERESOURCES(κ, c_σ^{req})
6: $s_\sigma^\kappa \xleftarrow{\text{from } \kappa}$ GETSTATE(κ, σ)
7: $\Delta Q(s_\sigma^\kappa, a_\sigma^\kappa) \leftarrow Q(s_\sigma^\kappa, a_\sigma^\kappa) - Q(s_\sigma^{\kappa_\sigma}, a_\sigma^{\kappa_\sigma}) - C_\kappa^{\text{free}}$
8: $\Delta Q(s_\sigma^{\kappa_\sigma}, a_\sigma^{\kappa_\sigma}) \leftarrow 0$
9: $D_{\kappa_\sigma}^{\min} \leftarrow \varnothing$
10:
11: # migrate σ to the destination site with max. ΔQ:
12: $\kappa^{\text{dst}} \leftarrow \operatorname{argmax}_{\kappa \in \mathcal{N}_\sigma^{(i)}} \Delta Q(s_\sigma^\kappa, a_\sigma^\kappa)$
13: execute all displacement actions $a_{\sigma'}^{\kappa'} \in D_{\kappa^{\text{dst}}}^{\min}$
14: execute migration action $a_\sigma^{\kappa^{\text{dst}}}$
15:
16: *# Returns $C_\kappa^{free}(m^{req})$ and the corresponding D_κ^{min}.*
17: *# Executed on κ, the edge site in question:*
18: **function** FREERESOURCES(κ, c_σ^{req})
19: **if** $c_\sigma^{\text{req}} \leq c_\kappa^{\text{avl}}$ **then** # already enough space?
20: **return** $(C_\kappa^{\text{free}} = 0, D_\kappa^{\min} = \varnothing)$
21: # 1) compute costs of all possible displacements:
22: **for all** $\sigma \in \Sigma_\kappa$ **do** # for all service instances at κ
23: **for all** $\kappa' \in \mathcal{N}_\sigma^{(i)}(s_\sigma^\kappa) \setminus \kappa$ with $c_\kappa^{\text{avl}} \geq c_\sigma^{\text{req}}$ **do**
24: $s_\sigma^{\kappa'} \xleftarrow{\text{from } \kappa'}$ GETSTATE(κ', σ)
25: $C_\kappa^{\text{disp}}(a_\sigma^{\kappa'}) \leftarrow Q(s_\sigma^\kappa, a_\sigma^\kappa) - Q(s_\sigma^{\kappa'}, a_\sigma^{\kappa'})$
26: $\rho_\sigma^{\kappa'} \leftarrow C_\kappa^{\text{disp}}(a_\sigma^{\kappa'})/c_\sigma^{\text{req}}$ # cost density
27: # 2) greedily select displacement actions for $D_\kappa^{\min}$:
28: $c \leftarrow 0$ # total freed CPU cores
29: $C_\kappa^{\text{free}} \leftarrow 0$ # total cost of freeing CPU cores
30: $D_\kappa^{\min} \leftarrow \varnothing$ # set of migration operations
31: $\Sigma_{\text{migrated}} \leftarrow \varnothing$ # set of migrated service instances
32: **for all** σ, κ' in ascending order of $\rho_\sigma^{\kappa'}$ **do**
33: **if** $c > c_\sigma^{\text{req}} \wedge \rho_\sigma^{\kappa'} \geq 0$ **then**
34: **return** $(C_\kappa^{\text{free}}, D_\kappa^{\min})$ # freed enough cores
35: **if** $\sigma \notin \Sigma_{\text{migrated}}$ **then**
36: **if** $c_\kappa^{\text{avl}} \geq c_\sigma^{\text{req}} + \sum_{a_{\sigma''}^{\kappa''} \in D_\kappa}^{\min} \delta_{\kappa',\kappa''} c_{\sigma'}^{req}$ **then**
37: # ensured there are no resource conflicts
38: $c \leftarrow c + c_\sigma^{\text{req}}$
39: $C_\kappa^{\text{free}} \leftarrow C_\kappa^{\text{free}} + C_\kappa^{\text{disp}}(a_\sigma^{\kappa'})$
40: $D_\kappa^{\min} \leftarrow D_\kappa^{\min} \cup \{a_\sigma^{\kappa'}\}$
41: $\Sigma_{\text{migrated}} \leftarrow \sigma$
42: **if** $c > c_\sigma^{\text{req}}$ **then** # success: found a valid $D_\kappa^{\min}$
43: **return** $(C_\kappa^{\text{free}}, D_\kappa^{\min})$
44: **else** # failure: impossible to free enough resources
45: **return** $(C_\kappa^{\text{free}} = \infty, D_\kappa^{\min} = \varnothing)$

resources, and thus, there is exactly one instance to displace at every recursion depth. The second case is that only a very low number of service instances fit on one edge site, again limiting the number of options to displace other instances. Besides, our experiments revealed that non-recursive BigMEC already performs close to an exact solution (see Section IV-A).

III. Learning Long-Term Utility Functions

In the introduction, we have stated that it's important to optimize migrations for the long term, and thus, the question that needs to be answered is "Are the migration costs now worth the benefit of a better placement later?". To answer it, the utility $Q(s_\sigma^\kappa, a_\sigma^\kappa)$ must be defined by a model that estimates the expected value of the migration action a_σ^κ (and its consequences) over the foreseeable future. Such a model is difficult to define by hand, as it would require accurate predictive models about the environment, e.g., the clients' movement patterns (unknown a priori). Fortunately, reinforcement learning is a proven tool to estimate an environment-adapted model of the the utility $Q(s_\sigma^\kappa, a_\sigma^\kappa)$ by observing the long-term effects of migration decisions "in the wild".

A. Markov Decision Process

To formally express the long-term utility function $Q(s_\sigma^\kappa, a_\sigma^\kappa)$ for one service instance σ, we define a Markov Decision Process (MDP) from the local perspective of service instance σ. An MDP is a framework for modeling decision making using a sequence of states, where each state transition decision is based only on the current state. The MDP of a service instance σ is defined by a tuple $(\mathcal{S}_\sigma, \mathcal{A}_\sigma, \mathcal{R}_\sigma, \mathcal{T})$, where

- $\mathcal{S}_\sigma$ is the state space and includes all possible states that service instance σ could encounter in its local view $\mathcal{N}_\sigma^{(i)}$. However, those states must not contain any information about other service instances, for reasons described later in this section.
- $\mathcal{A}_\sigma$ is the action space and includes all possible migration actions to edge sites in $\mathcal{N}_\sigma^{(i)}$ that service instance σ may have to consider.
- $\mathcal{R}_\sigma : \mathcal{S}_\sigma \times \mathcal{A}_\sigma \to \mathbb{R}$ is the myopic reward function of service instance σ and returns the *immediate* reward (= negated myopic cost) after executing an action. This includes both the cost of being placed at the current edge site and the cost of migrating (if applicable). Since this reward/cost function is determined by the MEC operator according to its specific pricing/business model, we consider it to be a black box and unknown a priori. Hence, our approach is flexible regarding the cost model.
- $\mathcal{T} = \mathbb{P}\left(s_\sigma{}^{(i+1)} \middle| s_\sigma{}^{(i)}, a_\sigma^\kappa{}^{(i)}\right)$ is the probabilistic transition function that represents transitions from one state to the next, i.e., it encompasses all external events between two migration decisions. Since the probability distribution of events in the real world, such as client mobility, is difficult to model by hand and can differ from region to region (e.g., urban vs. rural), we consider the transition function a black box as well. Reinforcement learning allows BigMEC to adapt to such unknown or changing environments.

In this MDP, we define the optimal long-term utility function

$$Q^*(s_\sigma^\kappa, a_\sigma^\kappa) = \mathbb{E}_{\pi^*}\left[\sum_{j=0}^{\infty} \gamma^j \mathcal{R}_\sigma\left(s_\sigma{}^{(i+j)}, a_\sigma{}^{(i+j)}\right) \middle|_{\substack{s_\sigma^{(i)} := s_\sigma^\kappa, \\ a_\sigma^{(i)} := a_\sigma^\kappa}}\right],$$

which is the expected discounted sum of rewards over all future time steps, assuming all actions are chosen by the optimal policy $\pi^* = \operatorname{argmax}_\pi \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^j \mathcal{R}_\sigma\left(s_\sigma{}^{(i+j)}, a_\sigma{}^{(i+j)}\right)\right]$, the policy that achieves the highest expected long-term reward. How $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$ is estimated trough trial and error via deep Q-learning is explained in the next Section III-B. In reinforcement learning parlance, $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$ is also known as the optimal state-action value function, or optimal Q-function.

It is important to note that the MDP we have just defined implies that, given the optimal utility function $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$, the optimal policy is $\pi^*(s_\sigma^\kappa) = a_\sigma^\kappa = \operatorname{argmax}_{a_\sigma^\kappa} Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$, i.e., the optimal policy of this MDP acts greedily, always selecting the action with the highest utility. However, unlike the greedy algorithm which has no mechanism to resolve resource conflicts, BigMEC does not act greedily with regards to $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$, but employs its displacement mechanism to explicitly resolve resource conflicts between service instances. Yet, it requires the above MDP definition, defined under the pretense that no other service instances exist. The reason is that the utility cost of displacing or blocking other service instances is already explicitly scrutinized in BigMEC and must not be considered a second time in $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$. That implies that any influence of other service instances on $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$ must be avoided, and thus, the state must not include any information about other instances.

This separation of concerns into (i) estimating the long-term value of a single service instance migration, and (ii) resolving conflicting interests between instances also means that our approach effectively avoids a competitive multi-agent learning scenario. If other service instances were part of the above MDP, the optimal greedy policy would have to consider their presence and actions in its decisions, i.e., it would incorporate these factors into the optimal utility function $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$. This inter-dependency of agents' decisions in multi-agent scenarios is a very difficult problem for reinforcement learning, since it creates a feedback loop between agents' current behavior and the optimal strategy to exploit that current behavior. Using non-multi-agent methods that disregard other agents[1] leads to instability (i.e., service instances continuously trying to outsmart each other's strategies). Unfortunately, multi-agent reinforcement learning (MARL) methods scale poorly due to the curse of dimensionality [27]. Our novel approach circumvents the challenges of multi-agent-systems and allows the use of standard (non-MARL) reinforcement learning approaches.

B. Reinforcement Learning to Estimate Long-Term Utilities

Reinforcement learning refers to methods for estimating optimal policies for MDPs with unknown transition functions

[1] this approach is called independent reinforcement learning

and reward functions through trial and error [28], [29]. In value-based methods, the optimal state-action-value function (i.e, the optimal long-term utility function) $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$ is approximated. For large state-action spaces, $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$ cannot be represented by a lookup table of all possible state-action pairs, and thus, has to be represented by an approximation, e.g., a neural network. In the following, we denote the neural network as $Q_\theta(s_\sigma^\kappa, a_\sigma^\kappa)$, where θ represents the network parameters. In our experiments, we have used a densely connected feed-forward network with 3 hidden layers of width 20. These and the following hyper-parameters of the learning method were found through a manual hyper-parameter search.

Our learning approach has a distributed architecture, inspired by the parallel asynchronous reinforcement learning methods first introduced by Mnih et al. [30]. Figure 3 conceptualizes how reinforcement learning is integrated into BigMEC. The Q-network $Q_\theta(s_\sigma^\kappa, a_\sigma^\kappa)$ is deployed on each edge site, where it is used to estimate the long-term utilities of outgoing migration decisions within BigMEC. After each invocation of BigMEC for a service instance σ, with probability p^{record} (in our case $p^{\text{record}} = 0.0125$) the recorded *experience* (i.e., the decision inputs and outputs) is archived in an *experience buffer*. This means that the following is inserted into the experience buffer of a central learner (FIFO queue, limited to 100.000 elements): (i) the state ${s_\sigma}^{(i)}$, (ii) the chosen action ${a_\sigma^\kappa}^{(i)}$, (iii) the resulting reward ${r_\sigma}^{(i)} = \mathcal{R}_\sigma({s_\sigma^\kappa}^{(i)}, {a_\sigma^\kappa}^{(i)})$, and (iv) the following state ${s_\sigma}^{(i+1)}$. After every 100 recorded experiences $({s_\sigma}^{(i)}, {a_\sigma^\kappa}^{(i)}, {r_\sigma}^{(i)}, {s_\sigma}^{(i+1)})$, a central learner trains the neural network with a random sample of 1000 experiences from the experience buffer, and then disseminates the updated neural network parameters θ' among the edge sites (inspired by [30]). Once the training process has converged, the central learner can be turned off. However, continuous online training is also possible. Contrary to the state-of-the-art multi-agent learning methods that need a central critic, BigMEC scales well during training, since migration decisions and training are asynchronous, and thus, training does not affect edge sites' decision rate.

To approximate the optimal utility function $Q^*(s_\sigma^\kappa, a_\sigma^\kappa)$, we used Clipped Double Q-learning [31], which is a state of the art reinforcement learning technique for neural network Q-functions. There is only one modification of the technique for BigMEC, as described in Algorithm 2: Instead of greedily selecting the action with the highest utility (see line 10 of Algorithm 1), the policy acts according to BigMEC, greedily selecting the actions with the highest global gain in utility $\Delta Q_\theta(s_\sigma^\kappa, a_\sigma^\kappa)$. Using a different policy than greedy action-selection is possible because Q-learning is an off-policy algorithm, i.e., the acting policy can differ from the learned policy π^* (which pretends that no other service instances exist).

Learning can only occur if new strategies are explored, i.e., if the actions are not strictly chosen according to BigMEC, but randomized to a degree. This is realized by replacing the greedy action selection in line 12 of Algorithm 2 (and line 8 of Algorithm 1) with an ϵ-greedy action selection. With probability ϵ, a random action is chosen instead of the greedy decision:

1: **if** RANDOM$(0 \dots 1) < \epsilon$ **then**
2: $\quad \kappa^{\text{target}} \leftarrow$ RANDOMCHOICE$(\mathcal{N}_\sigma^{(i)})$
3: **else**
4: $\quad \kappa^{\text{target}} \leftarrow \text{argmax}_{\kappa \in \mathcal{N}_\sigma^{(i)}} \Delta Q(s_\sigma^\kappa, a_\sigma^\kappa)$

For our experiments, we used $\epsilon = 0.05$ while training and $\epsilon = 0$ for the evaluation.

The reason for storing only a subset of the experiences is to be able to sample data over a longer period (storing all experiences at $p^{\text{record}} = 1$ quickly becomes a bottleneck), making the learned utility functions generalize better. The sampling rate p^{record} must be balanced with the size of the experience buffer and number of edge sites in the system. Due to the low sampling rate, online training becomes feasible with negligible negative impact from exploratory (random) behavior – only the recorded decisions must be subjected to the ϵ-greedy action selection, i.e. only one out of $\frac{1}{\epsilon \cdot p^{\text{record}}}$ (in our case $\frac{1}{0.05 \cdot 0.0125} = 1600$) decisions must be randomized for sufficient exploration.. Although we have used a centralized learner for simplicity, it could be replaced by federated learning architectures in the future [32]. Although federated learning is typically not entirely decentralized either, it could be advantageous to process more training data and to let each edge site specialize to its location.

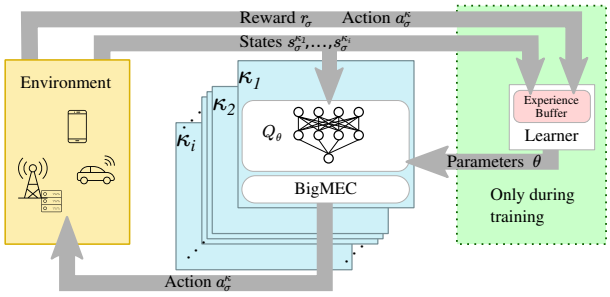


Figure 3. Centralized learning for distributed BigMEC.

IV. EXPERIMENTS

We have evaluated BigMEC in a large scale simulation and performed an ablation study investigating the impacts of the displacing strategy (Section IV-A) and the learned long-term utility (Section IV-B). To this end, we have analyzed the global cost (i.e., placement quality) and the number of migrations triggered by BigMEC in comparison to the greedy baseline algorithm, a no-migration policy, the exact, globally optimal solution, and the exact solution where the migration options are restricted to the same set of destination site candidates $\mathcal{N}_\sigma^{(i)}$ as the local approaches. The exact placement solutions were calculated in one second intervals using the state of the art Gurobi ILP solver[2]. As it is infeasible to calculate exact solutions for long-term optimal placement, all exact solutions are myopic, i.e., they disregard any future effects of their placement decisions.

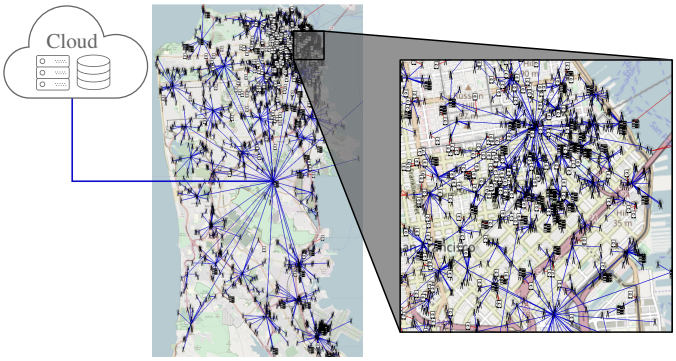


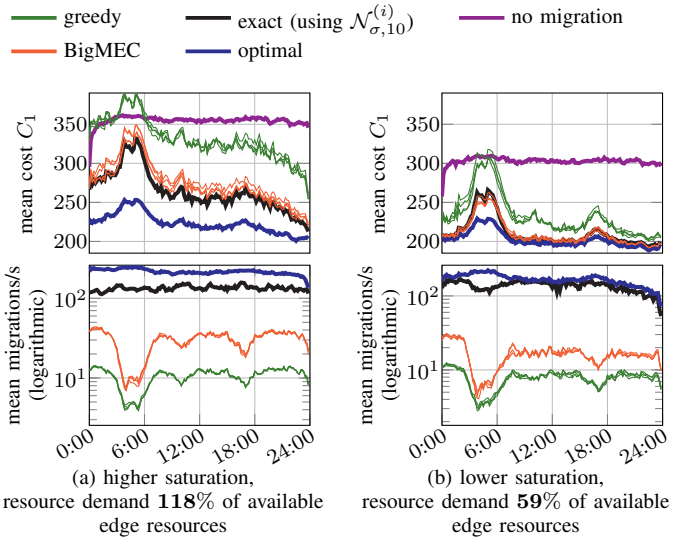
Figure 4. Simulation area and network topology.

a) Simulation Setup: For the sake of realism, we have used real-world data sets as much as possible. Nevertheless, there are still open questions that could have a major impact on real-world results, e.g., the business model that defines the objective function, or which virtualization technology will be used. However, since BigMEC was designed to be flexible regarding these open questions, the insights of our analysis are still valuable.

To simulate the clients' movements, we have used a one-day time slice of the cabspotting data set [33], which contains mobility traces of roughly 500 taxi cabs in San Francisco. For the wireless access point locations, we have used the coordinates of all LTE base stations of one mobile network operator in the San Francisco area, which we have gathered from the crowd-sourced database of OpenCellId[3]. The simulated area contains 1999 base stations. We have modeled the access network as a 2-tier hierarchical star topology (see Figure 4), as is typical for today's cellular networks. In reality, the end-to-end latency of an MEC service instance depends on many factors, such as the transmitted data volume, transmission delay, background traffic, congestion, protocol etc. Thus, latency would be measured end-to-end in a real-world deployment. As many of those factors are currently unknown due to the lack of MEC infrastructure and applications, we have conservatively assumed that each link of the access network adds $1\,\mathrm{ms}$ of end-to-end latency. We have also assumed that edge sites are deployed sparsely. Some are deployed at a base station, whereas others are deployed at aggregation sites. There are 50 edge sites at random nodes in the mobile network, i.e., their geographical distribution corresponds to the distribution of access points, which is in line with demand. Also, there is one nearby cloud node with a latency of $10\,\mathrm{ms}$ from the root of the hierarchical access network. If not otherwise stated, the set of possible migration destinations $\mathcal{N}^{(i)}_{\sigma,10}$ of a service instance σ comprises its current edge site, the 9 sites with the fewest number of hops to the instance's client plus the central cloud. Since the discovery of edge resources from the client device is an ongoing field of research [34] and beyond the scope of this work, we start each new service instance at the cloud

[2]https://www.gurobi.com/

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



(a) higher saturation, resource demand **118**% of available edge resources

(b) lower saturation, resource demand **59**% of available edge resources

Figure 5. Greedy algorithm compared to BigMEC, using a manually defined utility function. The cost function C_1 contains no migration cost.

and immediately trigger a migration decision to determine its best placement. Because BigMEC and the greedy algorithm are affected by random seeds, we have always simulated and plotted them four times to show the randomization's effects.

A. Service Instance Displacement

To analyze the displacement mechanism of BigMEC in isolation from the effects of reinforcement learning, we have first evaluated it with a manually defined myopic utility function instead of learning the long-term utility. Our setup for this experiment is as follows: Each edge site has 20 CPU cores, whereas the cloud has practically infinite resources. There is one service instance per client and their CPU requirements are uniformly distributed integers between 1 and 4. Thus, each edge site can host 5 to 20 service instances simultaneously, which we believe is realistic for container-based services. Since there are up to 471 simultaneous clients in the data set, the MEC system is saturated – at peak demand, the combined resource requirements of all services exceed the available resources at the edge by 18%. To compare this to a less saturated scenario, we have conducted a second experiment where we have doubled the edge site capacities to 40 CPU cores, meaning that only up to 59% of the available edge resources are needed.

The manually defined utility function

$$Q(s^{\kappa}_{\sigma}, a^{\kappa'}_{\sigma}) \coloneqq -C_1(a^{\kappa'}_{\sigma}) = -p_{\sigma} \cdot \lambda_{\sigma@\kappa'}$$

is determined by the importance-weighted service latency, where p_σ represents the importance of σ, and $\lambda_{\sigma@\kappa'}$ is the latency of the service instance if placed at the destination site κ'. To simulate heterogeneous services, each service instance has been assigned a random importance value p_σ between 1 and 100. The inverse of the utility, C_1, is the cost function used to assess performance over a $1s$ time interval: the mean cost of all migration and (explicit as well as implicit) non-migration

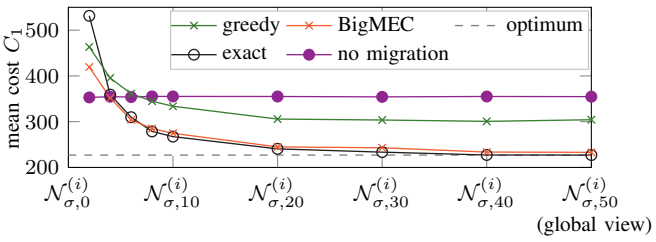


Figure 6. Global cost C_1 achieved by different myopic methods for different sizes of $\mathcal{N}^{(i)}_{\sigma,n}$ (scenario with higher saturation).

actions. Minimizing C_1 is also the optimization objective of the exact solvers.

Figure 5 compares the mean cost and number of migrations resulting from the greedy algorithm, BigMEC, and the baselines over one day, for the two levels of saturation. We have executed each experiment 4 times (our simulation handles simultaneously invoked migration decisions in random order). While the greedy algorithm can be even worse than not migrating at all during peak demand between 3:00 and 6:00, BigMEC's novel displacement mechanism makes it perform close to the exact solutions, given the same restriction to $\mathcal{N}^{(i)}_{\sigma}$. The performance difference between BigMEC and the greedy algorithm can be fully explained by the displacement mechanism, which is their only difference in this experiment. Thus, BigMEC's displacement mechanism provides a clear advantage over previous fully decentralized service placement methods, such as [20]–[22].

Although BigMEC's performance is close to the exact solution, it needs 5 to 10 times fewer service migrations, which indicates that further service migrations have a strongly decreasing benefit. This observation confirms our hypothesis that adding recursive displacement to BigMEC could only yield negligible improvements. That BigMEC executes substantially more migrations than the greedy algorithm illustrates how its displacement mechanism keeps service instances mobile, confirming it solves the saturation issues mentioned in Section II-C.

To assess the impact of the size of $\mathcal{N}^{(i)}_{\sigma,n}$, we have evaluated the more saturated experiment for different values of n (results in Figure 6). Constrained to the same candidate sites $\mathcal{N}^{(i)}_{\sigma,n}$, BigMEC is very close to the exact solution and quickly approaches the optimum when increasing the number of candidates. Clearly, increasing the size of $\mathcal{N}^{(i)}_{\sigma,n}$ yields diminishing improvements beyond a certain point. Restricting migration decisions to a fixed-size set of nearby edge sites therefore makes BigMEC highly scalable with negligible loss of placement quality compared to optimal placement. Interestingly, for very small sizes of n, even not migrating yields lower average cost than repeatedly optimizing within the confines of $\mathcal{N}^{(i)}_{\sigma,n}$ (a badly informed placement solution for one step can result in a less desirable starting position in the next step).

B. Learned Long-Term Utility

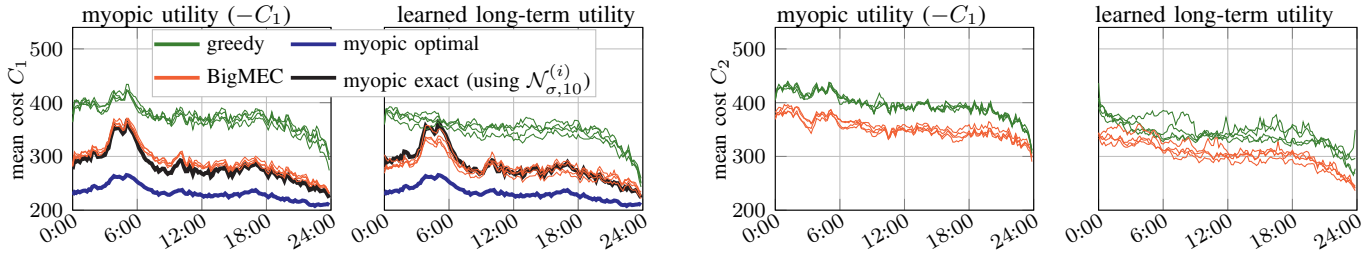
Next, we have assessed the impact of estimating the long-term utility. To accelerate our simulation we have simplified the scenario in such a way that each service instance requires 1 CPU core and each edge site has 8 cores. This again results in a maximum demand of 118% of the available resources.

First, we have tested how well our learning approach can approximate an optimal strategy, given only the indirect feedback of the reward function. To this end, we have designed an experiment in which there are no migration costs that could affect the long-term value of migration actions. Assuming that there are no future costs, a myopic hand-defined utility function can serve as a benchmark: $Q(s^{\kappa}_{\sigma}, a^{\kappa'}_{\sigma}) \coloneqq -C_1(a^{\kappa'}_{\sigma})$. C_1 also acts as the cost function (i.e., $\mathcal{R}(s^{\kappa}_{\sigma}, a^{\kappa'}_{\sigma}) = -C_1(a^{\kappa'}_{\sigma})$) for reinforcement learning. In the absence of future costs, this hand-defined utility function is optimal. We have trained the neural network for the first six hours of simulated time (not depicted), and then evaluated the entire day without further training. Again, we have executed each experiment 4 times to assess the repeatability of the reinforcement learning process. Figure 7a compares the cost of using the myopic utility function (left) to the cost of using the learned long-term utility function (right). For reference, each plot also displays the myopic optimal solution and the myopic exact solution that is constrained to $\mathcal{N}^{(i)}_{\sigma,10}$. Due to the absence of transition costs, we expected the impact of future events to be so negligible that the hand-defined myopic utility would not be outperformed by a utility function that is learned using only trial and error. Yet, the learned utility functions clearly provide a slightly lower cost than the hand-defined utility for both BigMEC and the greedy algorithm. This result demonstrates that our approach can reliably learn very accurate utility functions.

Next, we have evaluated BigMEC with an objective function that requires much more long-term reasoning because it includes substantial migration costs:

$$C_2(a^{\kappa'}_{\sigma}) = \underbrace{p_{\sigma} \cdot \lambda_{\sigma@\kappa'}}_{C_1} + c_m \cdot (1 - \delta_{\kappa,\kappa'}),$$

where c_m is the cost of a single migration, and $\delta_{\kappa,\kappa'} = 1$ if κ equals κ' (otherwise, $\delta_{\kappa,\kappa'} = 0$). Thus, the migration cost c_m is incurred only when a migration is executed. Figure 7b shows the results for $c_m = 1000$, which is substantial enough to hamper the effectiveness of myopic methods. For reference, Figure 7b (left) shows the cost C_2 over time when re-using the myopic $-C_1$ as the hand-defined utility function for the greedy algorithm and BigMEC. Figure 7b (right) shows that, for both BigMEC and the greedy algorithm, the cost is substantially lower when using learned long-term utility functions. An interesting observation is that the performance gap between BigMEC and the greedy algorithm is less pronounced than in the previous experiment without migration costs. This is because both methods learn to perform fewer migrations when service migration is less desirable, and thus their strategies align. This finding is also supported by Section IV-B, which shows the achieved cost, latency, and number of migrations



(a) migration cost $c_m = 0$

(b) migration cost $c_m = 1000$

Figure 7. Myopic utility function compared to learned, long term utility functions.

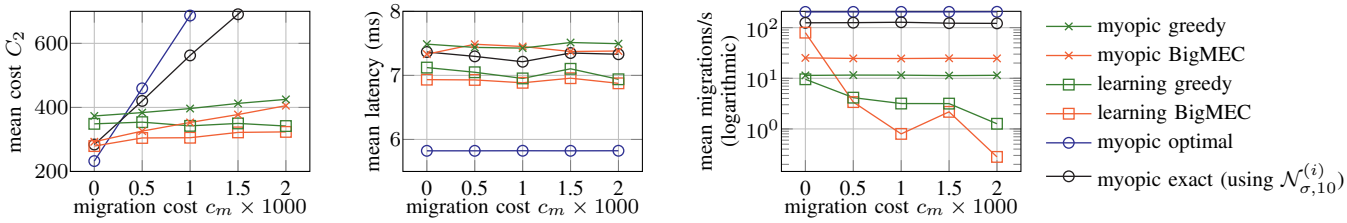


Figure 8. Mean cost, service latency, and number of migrations/s for different migration costs.

Table II
MEAN COMMUNICATION TIMES REQUIRED FOR A SINGLE MIGRATION DECISION BY DIFFERENT MIGRATION/PLACEMENT METHODS, BASED ON THE USED DATA SETS.

	all service locations (mean)	if service at edge (mean)	if service at cloud (mean)
greedy	12.9 ms	7.5 ms	19.9 ms
BigMEC	21.8 ms	14.6 ms	27.5 ms
centralized	26.0 ms	26.0 ms	26.0 ms

of the myopic and learning algorithms for a range of migration costs c_m. Further, Section IV-B (left) confirms that, the higher the future migration costs, the better the learned utility functions perform relative to the myopic baselines. All in all, our experiments have validated that BigMEC's combination of single-agent reinforcement learning with the new displacing mechanism for conflict resolution reliably learns good long-term migration strategies.

a) Decision Time: Through local decision making, BigMEC can quickly react to unforeseen events and repair degraded QoS, e.g., caused by an access point change. We have measured the computation time of migration decisions, which is dominated by the neural network inference. Whereas the greedy algorithm takes on average $0.8\,\mathrm{ms}$ on our test system, BigMEC, having to consider many more migration options, takes $7.2\,\mathrm{ms}$. One training update takes $190\,\mathrm{ms}$ to compute. These times were measured using a single thread of a Intel Intel Core i9–10900KF CPU @ 3.70GHz.

We have also modeled the communication times required by service migration/placement algorithms to collect the required data and communicate the subsequent migration decision to the involved nodes, and compiled them in Table II. These times were recorded in the $c_m = 0$ scenario, assuming that the data from the cloud need not be collected since we consider it to be infinitely elastic. While centralized approaches always need the worst-case latency to capture their global view, the decentralized algorithms typically only need local communication. Nonetheless, the worst-case communication time of BigMEC in the simulated scenario is longer than that of centralized approaches due BigMEC's cascading communication pattern.

V. DISCUSSION

For a real distributed deployment of BigMEC, several practical challenges still have to be overcome. First, viable business and pricing models must be identified and translated into an objective function. This fundamental problem remains open, as there are currently no proven business models for MEC operators and no proven closed-loop MEC applications. Second, the capability to monitor all aspects of the desired objective function must be present. Third, while our simulation executes migration decisions atomically, BigMEC must eventually be implemented as a distributed communication protocol that ensures that overlapping decision processes can neither block each other nor allocate resources more than once. Nevertheless, such collisions between concurrent migration decisions should be rare, given the fast decision time ($< 10\,\mathrm{ms}$) and comparatively infrequent real-world events, such as access point change due to mobility. The simplest approach to mitigate such collisions would be for all edge sites to reject requests to participate in migration decisions while another decision is in progress.

a) Future Research Directions: While our experiments have demonstrated that BigMEC generally performs much better than the greedy algorithm, the faster decision speed of the greedy algorithm cannot be overlooked. This leads to the

idea of deciding on a case-by-case basis if a migration decision has to be displacing, or if a faster greedy decision is preferred. However, we leave this trade-off decision to future work.

Moreover, we could imagine BigMEC nodes of multiple MEC operators inter-operating in a single system, akin to the roaming mechanism in cellular networks. Although that is possible due to the distributed architecture of BigMEC, further research into the stability of multiple interacting (and competing) learning processes is needed.

There are limitations that affect all learning based service placement/migration techniques. Although experience has shown them to work well, it is generally impossible to make performance guarantees, and the performance of learning methods depends on many hyper-parameters that must be tuned. Further research into verifiable and less hyper-parameter-dependent learning methods is needed.

VI. Conclusion

This paper presented BigMEC, a new approach to decentralized, reinforcement learning based service migration. We have solved several problems of previous approaches, using three key innovations: First, we have demonstrated that a local preselection of destination sites is sufficient for near-optimal migration decisions and allows decentralized migration algorithms to be highly scalable with negligible performance impact. Second, we have presented an approach that decouples learning from resource conflict resolution. That approach circumvents the need for multi-agent learning methods, which still suffer from scalability problems. Third, we have demonstrated that BigMEC's displacing policy for resource conflict resolution yields near-optimal migration decisions, despite the fully decentralized approach. This work does not raise ethical issues.

Acknowledgment

This work has been funded by the German Research Foundation (DFG) within the Collaborative Research Center 1053 (MAKI) and and by the German Federal Ministry of Education and Research (BMBF) – 01IS17050.

References

[1] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *Communication Survey Tutorials*, pp. 1628–1656, 2017. [Online]. Available: https://doi.org/10.1109/COMST.2017.2682318

[2] M. S. ElBamby, C. Perfecto, M. Bennis, and K. Doppler, "Toward low-latency and ultra-reliable virtual reality," *IEEE Network*, vol. 32, no. 2, pp. 78–84, 2018.

[3] J. Wang, Z. Feng, S. A. George, R. Iyengar, P. Pillai, and M. Satyanarayanan, "Towards scalable edge-native applications," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC 2019, Arlington, Virginia, USA, November 7-9, 2019*, S. Chen, R. Onishi, G. Ananthanarayanan, and Q. Li, Eds. ACM, 2019, pp. 152–165. [Online]. Available: https://doi.org/10.1145/3318216.3363308

[4] X.-M. Zhang, Q.-L. Han, X. Ge, D. Ding, L. Ding, D. Yue, and C. Peng, "Networked control systems: a survey of trends and techniques," *IEEE/CAA Journal of Automatica Sinica*, vol. 7, no. 1, pp. 1–17, 2020.

[5] V. Cozzolino, A. Y. Ding, and J. Ott, "Edge Chaining Framework for Black Ice Road Fingerprinting," in *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*. ACM, 2019, pp. 42–47.

[6] J. Wang, Z. Feng, Z. Chen, S. A. George, M. Bala, P. Pillai, S. Yang, and M. Satyanarayanan, "Bandwidth-efficient live video analytics for drones via edge computing," in *Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 159–173. [Online]. Available: https://doi.org/10.1109/SEC.2018.00019

[7] B. Varghese, E. de Lara, A. Y. Ding, C. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele, and P. Willis, "Revisiting the arguments for edge computing research," *IEEE Internet Comput.*, vol. 25, no. 5, pp. 36–42, 2021. [Online]. Available: https://doi.org/10.1109/MIC.2021.3093924

[8] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys and Tutorials*, pp. 1657–1681, 2017.

[9] X. Ma, A. Zhou, S. Zhang, and S. Wang, "Cooperative service caching and workload scheduling in mobile edge computing," in *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*. IEEE, 2020, pp. 2076–2085. [Online]. Available: https://doi.org/10.1109/INFOCOM41043.2020.9155455

[10] F. Aït-Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 65:1–65:35, 2020. [Online]. Available: https://doi.org/10.1145/3391196

[11] J. Gedeon, F. Brandherm, R. Egert, T. Grube, and M. Mühlhäuser, "What the Fog? Edge Computing Revisited: Promises, Applications and Future Challenges," *IEEE Access*, pp. 152 847–152 878, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2948399

[12] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. S. Chan, and K. K. Leung, "Dynamic service migration and workload scheduling in edge-clouds," *Performance Evaluation*, pp. 205–228, 2015. [Online]. Available: https://doi.org/10.1016/j.peva.2015.06.013

[13] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[14] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, 2018.

[15] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. S. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," in *Proceedings of the 14th IFIP Networking Conference*. IEEE, 2015, pp. 1–9. [Online]. Available: https://doi.org/10.1109/IFIPNetworking.2015.7145316

[16] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *HotNets*, 2016, pp. 50–56.

[17] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," *arXiv preprint arXiv:1810.01963*, 2018.

[18] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Trans. Serv. Comput.*, vol. 12, no. 5, pp. 712–725, 2019. [Online]. Available: https://doi.org/10.1109/TSC.2018.2827070

[19] K. Ray, A. Banerjee, and N. C. Narendra, "Proactive microservice placement and migration for mobile edge computing," in *5th IEEE/ACM Symposium on Edge Computing, SEC 2020, San Jose, CA, USA, November 12-14, 2020*. IEEE, 2020, pp. 28–41. [Online]. Available: https://doi.org/10.1109/SEC50012.2020.00010

[20] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1002–1016, 2017.

[21] C. Zhang and Z. Zheng, "Task migration for mobile edge computing using deep reinforcement learning," *Future Gener. Comput. Syst.*, vol. 96, pp. 111–118, 2019. [Online]. Available: https://doi.org/10.1016/j.future.2019.01.059

[22] Z. Gao, Q. Jiao, K. Xiao, Q. Wang, Z. Mo, and Y. Yang, "Deep reinforcement learning based service migration strategy for edge computing," in *13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, San Francisco,*

CA, USA, April 4-9, 2019. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/SOSE.2019.00025

[23] F. Brandherm, L. Wang, and M. Mühlhäuser, "A Learning-based Framework for Optimizing Service Migration in Mobile Edge Clouds," in *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*. ACM, 2019, pp. 12–17. [Online]. Available: http://doi.acm.org/10.1145/3301418.3313939

[24] A. Abouaomar, Z. Mlika, A. Filali, S. Cherkaoui, and A. Kobbane, "A deep reinforcement learning approach for service migration in mec-enabled vehicular networks," in *46th IEEE Conference on Local Computer Networks, LCN 2021, Edmonton, AB, Canada, October 4-7, 2021*. IEEE, 2021, pp. 273–280. [Online]. Available: https://doi.org/10.1109/LCN52139.2021.9524882

[25] C. Liu, F. Tang, K. Li, Z. Tang, and K. Li, "Distributed task migration optimization in mec by extending multi-agent deep reinforcement learning approach," *Transactions on Parallel and Distributed Systems*, pp. 1–1, 2020.

[26] J. N. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual Multi-Agent Policy Gradients," in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI-18)*, S. A. McIlraith and K. Q. Weinberger, Eds. AAAI Press, 2018, pp. 2974–2982. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17193

[27] L. Busoniu, R. Babuska, and B. D. Schutter, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, pp. 156–172, 2008. [Online]. Available: https://doi.org/10.1109/TSMCC.2007.913919

[28] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, pp. 26–38, 2017. [Online]. Available: https://doi.org/10.1109/MSP.2017.2743240

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

[30] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33nd International Conference on Machine Learning, (ICML)*. JMLR.org, 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html

[31] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," pp. 1582–1591, 2018. [Online]. Available: http://proceedings.mlr.press/v80/fujimoto18a.html

[32] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y. Liang, Q. Yang, D. Niyato, and C. Miao, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Commun. Surv. Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020. [Online]. Available: https://doi.org/10.1109/COMST.2020.2986024

[33] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, "CRAWDAD dataset epfl/mobility (v. 2009-02-24)," Downloaded from https://crawdad.org/epfl/mobility/20090224, Feb. 2009.

[34] A. Abujoda and P. Papadimitriou, "MIDAS: middlebox discovery and selection for on-path flow processing," in *7th International Conference on Communication Systems and Networks (COMSNETS)*. IEEE, 2015, pp. 1–8. [Online]. Available: https://doi.org/10.1109/COMSNETS.2015.7098686