POLICY DESIGN AND VERIFICATION FOR MULTI-ACCESS EDGE COMPUTING: A FORMAL METHODS PERSPECTIVE

Kaustabha Ray

Policy Design and Verification for Multi-Access Edge Computing: A Formal Methods Perspective

Thesis submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy in Computer Science by

Kaustabha Ray

under the supervision of

Dr. Ansuman Banerjee

Professor

Advanced Computing and Microelectronics Unit

Indian Statistical Institute

Kolkata, India

February 2022

Dedicated to my family and my supervisor

Acknowledgements

I would like to express my gratitude towards my supervisor Dr. Ansuman Banerjee for his valuable advice and guidance from time to time. I would like to thank my parents for their support throughout the PhD journey. Finally, I would like to thank Indian Statistical Institute for providing me the opportunity to pursue my graduate studies without which all of this would not have been possible.

Kaustabha Ray

Kaustabha Ray
Indian Statistical Institute
Kolkata

Abstract

In recent times, Multi-Access Edge Computing (MEC) is showing much promise as the preferred application service provisioning model to facilitate convenient access to services for mobile users. The central idea of MEC is to have service providers deploy application services as containers on MEC servers located near mobile base stations. User service invocations are typically routed to, and served from nearby MEC servers on their route as they move around, with improved latency and turnaround times. This provisioning model is increasingly being acknowledged as a near-user low latency convenient alternative to traditional cloud computing. Driven by new innovations in MEC, the number of application services (e.g. Object recognition, obstacle identification, navigation, maps, games, e-commerce etc.) hosted at edge servers to be used by mobile users (e.g. autonomous vehicles, drones, users on the move) is also growing at a considerable pace.

A typical MEC deployment involves the orchestration of a number of policies for several tasks like service allocation, service placement, service migration, service replication, user management and resource scheduling. These policies coordinate as a whole to ensure low latency access and continued availability to end users. Indeed, designing MEC policies taking into consideration different scenarios and optimization metrics is an active area of research in recent times. However, most design approaches proposed in recent literature either resort to traditional optimization techniques to ensure optimality, thereby limiting scalability or resort to learning based approaches without any formal guarantees on the synthesized policies. The objective of this thesis is to leverage formal methods for the design and verification of MEC policies to aid scalability and provide formal guarantees on their performance.

A key challenge in MEC is to devise an efficient service placement policy which determines the availability of application services on MEC servers. This is a non-trivial challenge, considering the fact that typical edge servers are not as resource-equipped as their cloud server counterparts – thus, the trivial deployment of hosting all application services at all edge servers is infeasible. This necessitates judicious planning and allocation of the service-server-user mapping over time as different service requests come in from different users. This is aggravated by the fact that users accessing these application services are typically mobile, moving in and out of the service zones of different edge servers. This often necessitates service migrations to preserve continuity of service provisioning and a steady acceptable Quality of Experience (QoE) and user perceived latency. While a number of researchers have addressed the service placement and allocation problem, they considered traditional monolithic applications where all services of an application are deployed as a single container. Microservice based applications, on the other hand, are split into a set of interacting microservices with each microservice independently deployed. Service placement and allocation for microservice based applications thus add further intricacies to the service placement and service allocation landscape. To address this challenge, we design a user service allocation policy for microservice based applications and demonstrate the benefits of prefetching microservices to improve user perceived latency.

MEC servers comprise co-located heterogeneous applications each with individual latency requirements. To improve load balancing, a number of application replicas are deployed on MEC servers, automatically spawned or retracted by an auto-scaling policy. As new service requests arrive, we have a challenging task of deciding whether to spawn a new thread in an already existing container with added resource contention and possible application latency violations or create a new service container instance to reduce contention amongst resources with lower user perceivable latency. We propose the design of an auto-scaling policy that automatically determines when to retain/add/remove container instances of an application while at the same time ensuring that the probability of latency violations is minimized.

State of the art MEC policies work oblivious to the presence of MEC server failures. In the event of a failure, a fault-recovery policy ensures service continuity by re-initializing application containers that were deployed on MEC servers prior to failure. Fault Tolerance approaches in pervasive computing environments typically deal with faults in an adhoc manner with no application specific distinction, executing recovery decisions with each occurrence of failure. To cater to the real time MEC ecosystem, we propose a prioritized approach wherein we leverage Probabilistic Model Checking to quantify the potential impact of multiple server failures, application priority and potential resource contention.

While designing MEC policies has been the centrepiece of focus in MEC literature, a much less explored avenue is verification of policies, that is important for establishing if a given policy conforms to a desired performance specification. An additional contribution of this thesis is a framework for modeling and verification of MEC policies. Traditional approaches to performance verification either develop analytical models and derive performance bounds mathematically on each performance metric under consideration or resort to simulation resulting in inadequate representation of non-deterministic behaviour. In contrast, in this thesis, we adopt a formal modeling and verification approach, offering accurate representation of non-determinism quantifying its impact on all possible executions of the model. Our framework is able to deduce quantitative guarantees on policy performance under varying request distributions.

We validate our proposed approaches on publicly available benchmark datasets to demonstrate the effectiveness of our proposals. We believe that our work will open up a lot of new research directions in different aspects of MEC through the formal methods lens.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of mobile and Internet-of-Things (IoT) devices has exploded in recent years, with the market estimated to hit 32 billion devices by 2023. As a consequence, the number of applications offering a wide assortment of features to such devices has skyrocketed dramatically. Features such as object detection, facial recognition, and natural language processing, are commonplace on such devices and are becoming increasingly computationally intensive. The computing capacity and battery life of such devices are, however, constrained by their physical size and resource constraints. Quite evidently, the standalone performance of executing computationally intensive operations locally on the devices is insufficient to ensure a seamless Quality-of-Experience (QoE) for mobile and IoT device users.

For the past few years, cloud computing has been used as the main technology to overcome the limited computational power and battery life of mobile and IoT devices. Cloud computing offers on-demand availability of computer system resources over the internet from centrally located data centers. Computationally intensive processes are offloaded to cloud data centers instead of being executed locally on the devices. However, since offloading to the cloud through a multi-hop backbone network results in high long-haul latencies, the cloud computing model often fails to meet the latency requirements of latency-sensitive real-time features. To mitigate these effects, a new paradigm, Multi-Access Edge Computing (MEC) was envisioned.

Multi-Access Edge Computing was standardized by the European Telecommunications Standards Institute (ETSI) and is also referred to as Mobile Edge Computing in literature [1]. MEC is an emerging paradigm in which computing and storage facilities in the form of MEC servers are placed at multiple locations between the endpoint Internet of Things (IoT) devices and a traditional centralized cloud data center. Mobile and IoT device end-users connect to MEC servers in one network hop, via a wireless link. MEC thus provides a distributed computing environment for application task processing and service hosting. As users move around, applications being utilized by them on their devices trigger service requests to execute computationally intensive processes on MEC servers, thereby reducing the backbone network latencies incurred in offloading tasks to the cloud. MEC is predicted to become the next multi-billion dollar technology market and an enabler towards a truly distributed Internet-of-Everything (IoE). Recently, MEC has been deployed in several cities in United States of America with strong collaboration between traditional cloud providers and telecommunication organizations.

1.1 Challenges in MEC

In the MEC environment, several policies such as computation offloading, service placement, service allocation, auto-scaling, fault tolerance, each governing different facets, coordinate as a whole, to ensure low latency access and continued availability. In this thesis, we deal with the design and verification of MEC policies leveraging formal methods. In particular, we explore the design of service placement, service allocation, auto-scaling, and fault-recovery policies. Additionally, we propose a framework for characterizing service allocation policies through the formal verification lens. In the following discussion, we present a high level overview of some of the important MEC components that are related to this thesis.

Service Placement and Service Allocation: The objective of a service placement policy is to determine application service deployments on MEC servers. For a particular application service request, a service allocation policy determines the binding between the service request and an MEC server. Thus, service placement and service allocation are intrinsically linked to each other with a service allocation policy regulating request-server bindings contingent on the deployment of application services on MEC servers dictated by a service placement policy.

Service Migration: Due to end-user mobility, service placements and service allocations need to be dynamically reconfigured to maintain end-user low latency access to services. Service migration refers to such dynamic re-configurations considering factors such as service invocation and mobility patterns.

Computation / Task Offloading: A computation/task offloading policy determines whether to execute a task locally or a part thereof on the device itself or to execute the task on an MEC server. Such a policy has to consider different factors such as current load associated with MEC servers, network congestion, device battery-life and computational capabilities.

Task Scheduling: Correlated to a computation offloading policy is a task scheduling policy that indicates the execution order of the tasks on the server. Thus, while a computation offloading policy determines where to execute tasks, a task scheduling policy determines the order of task execution. MEC complicates task scheduling with the possibility of task execution on both devices as well as servers.

Auto-Scaling: The number of application instances deployed on MEC servers is automatically adjusted by an auto-scaling policy to alleviate resource contention and improve load balancing. With MEC servers not as resource-equipped as their cloud server counterparts, a trivial deployment of hosting all application services at all edge servers is infeasible. This necessitates meticulous planning and deployment of application instances in close coordination with service request traffic patterns.

Fault Tolerance: State of the art service placement, service allocation, service migration, computation offloading, task scheduling and auto-scaling policies operate agnostic to the possibility of MEC server failures. A fault-recovery policy ensures the continued availability of services in the event of a failure by re-initializing applications deployed at faulty servers.

MEC for Microservice based Applications: Applications are either monolithic or microservice based. While a monolithic application is a single unified executable unit, a microservice architecture breaks down individual functionality into a collection of smaller independent units. Monolithic applications are simpler to develop, test, debug and deploy but provide less flexibility for incorporating changes. Microservice based applications, on the other hand, offer segregated units, where each unit can be independently developed, managed, scaled, and modified. Individual microservices typically communicate via Remote Procedure Calls (RPC). Such flexibility transpires with the added complexity of managing inter-service communication between the microservices as well as managing individual microservice deployments on MEC servers. The microservice structure leaves room for improvement in the service placement and migration model from a latency perspective, considering the fact that the independent constituents of a given microservice based workflow can be independently hosted and provisioned.

MEC servers, applications, and end-users, thus, form a large scale distributed system, that has garnered significant research interest in recent times. Indeed, algorithms and architectures for enabling the MEC paradigm have reached a fair level of sophistication today but are still replete with several challenges. Our thesis is motivated by some observations on the current context of MEC deployment as outlined in the following section. We summarize the contributions of this thesis thereafter.

1.2 Motivation for this dissertation

In an MEC environment, the design of policies plays a crucial role in determining the end-user perceivable latencies and Quality-of-Experience. An inefficiently designed policy, if deployed, can lead to an aggravated Quality-of-Experience for end-users. The primary motivation of this thesis is to design policies concerning the different dimensions of the MEC environment to improve end-user latencies over existing designs. In particular, we consider the design of policies in the following contexts:

- Service placement and service allocation have received a lot of attention in recent years. Authors have investigated service placement and allocation both in individual [2] as well as joint contexts [3]. Several authors have investigated the role of static [2] service placement schemes as well as dynamic [4] placement schemes where the service availability on MEC servers vary over time. Traditional service placement and allocation schemes consider monolithic applications [5, 6, 7, 8]. Recently, some policies have been proposed considering microservice based applications [9]. However, these proposals do not consider the intercommunication dependency structure existing in microservice based applications. The geospatial distribution of end-users and microservice deployments on MEC can have a critical role in the overall user-perceived latencies. Additionally, traditional service placement policies are typically reactive, i.e, deploy services when requested by end-users. Exploiting the role of the microservice architecture coupled with proactive placement remains largely unexplored.

- In the MEC environment, multiple heterogeneous service requests are provisioned by MEC servers simultaneously. Consequently, resource contention becomes a critical issue. An auto-scaling policy automatically adjusts the number of application replicas deployed on MEC servers to alleviate the resource contention with enhanced load balancing. Auto-Scaling policies have been investigated in traditional cloud computing contexts [10, 11, 12]. However, auto-scaling policies in MEC have received less attention. The authors in [13] studied energy-aware auto-scaling in the context of energy-harvesting devices in MEC. Traditional auto-scaling policies are either rule-based [10], rely on low overhead learning-based approaches with no guarantees on the incurred latencies [13] or utilize Probabilistic Model Checking [11] with formal guarantees but incur a high runtime overhead rendering them unsuitable for real-time scenarios [14] encountered in the MEC environment. Designing reliable learning enabled auto-scaling policies with low runtime overheads is thus crucial towards ensuring effective load balancing.

- Owing to the large-scale and distributed nature of the MEC environment, MEC servers are in fact more susceptible to failures as compared to their cloud counterparts due to their distributed nature [15]. Fault Tolerance is traditionally handled by application replicas [15] or by failure prediction [16]. In [15], the authors proposed a method for evaluating edge server resilience. They studied the trade-off between replica deployment and the associated cost by utilizing dependency based failure prediction. However, they did not cater to fault-recovery measures. Recently, the authors in [17] proposed an Edge-based IoT architecture for IoT-Edge environments that caters to faults by re-allocation. Such techniques target traditional pervasive computing environments and neither consider the latency variability in an MEC environment nor the possibility of simultaneous failures.

While designing MEC policies is the centerpiece of focus in MEC literature, a much less explored avenue is quantitative verification of policies. Quantitative verification aims at establishing robust guarantees of properties with respect to all possible executions of a model of a system. Several authors studied modeling and verification in various domains such as sensor networks [18, 19, 20], cloud computing [10, 21], services computing [22], wireless networks [23, 24, 25], software usage [26, 27], task allocation [28], power management [29], autonomous vehicles [30], amongst others. Simulation, which is utilized as the de-facto methodology for analyzing state-of-the-art policies, can neither provide formal guarantees nor adequately support non-deterministic behaviour, thereby affecting accuracy [31]. Quantitative verification of policies pertains to establishing the extent to which a particular policy conforms to a desired specification. As a concrete example, quantitative verification can help us verify requirements such as for a given service allocation policy, a user request can always be allocated to some MEC server within a desired time limit. Quantitative verification has been demonstrated as an effective technique for analyzing the characteristics of wireless network protocols [31, 32, 33]. In [10], the authors utilized Probabilistic Model Checking to quantitatively verify auto-scaling policies in traditional cloud computing environments. However, this line of study has received relatively less attention in the MEC context. In this thesis, we develop a generic verification framework for service allocation policies.

1.3 Contributions of this dissertation

The objective of this thesis is to a) design policies for addressing some of the MEC challenges outlined above with a motivation of optimizing end-user latencies and b) to develop a framework for verifying the performance characteristics of MEC policies. The contributions of this thesis are briefly summarized below.

1.3.1 Design

We first explore the design of service allocation, service placement, auto-scaling and fault-recovery policies aimed at reducing end-user latency as summarized below.

- *Service Allocation for Microservice based Applications:* In the first contributing chapter, we propose a service allocation policy for microservice based applications. We consider service allocations with a pre-deployed service placement scheme, where multiple functionally equivalent microservices are available to aid fault tolerance and load balancing. This induces microservice bundles amidst microservice Application Programming Interface (API) compatibility constraints. We present a novel multi-partite hyper-graph visualization of the allocation problem and analyze its hardness. Utilizing a novel combination of Integer Linear Programming (ILP) and abstraction refinement as a potential solution, our approach determines optimal service allocations.

- *Proactive Microservice Placement and Allocation:* In this work, we design a proactive microservice placement and allocation policy that automatically adjusts the service placement and allocation configurations depending on the mobility and usage patterns of end-users. We consider how proactive deployment of microservices can aid in improving end-user latencies. We model the deployment of microservice based applications using a Markov Decision Process (MDP). We utilize Dyna-Q Learning, a combination of model-free and model-based Reinforcement Learning (RL) which utilizes dynamic real-time interactions with the environment to learn the associated rewards integrated with simulations on the learnt model [34, 35, 36]. Additionally, we design a heuristic to cater to server capacity constraints.

- *Horizontal Auto-Scaling for Applications:* We propose a Safe Reinforcement Learning based auto-scaling policy that can efficiently adapt to MEC request load variations. We model the MEC environment using a MDP. We express latency requirements in Linear Temporal Logic (LTL) [37], which act as a guide to automatically learn auto-scaling decisions that maximize the probability of satisfying the LTL specification. We introduce a quantitative reward mechanism based on the LTL formula to tailor service specific latency requirements. Further, we prove that our reward mechanism ensures convergence of standard Safe Reinforcement Learning approaches.

- *Two-Fold Fault-Recovery:* MEC servers are susceptible to various types of failures such as communication link failures, hardware failures, and so on. We propose a priority driven two-fold fault recovery policy. We propose a formal methods driven local recovery policy for high-priority applications. We use Stochastic Multi-Player Games (SMGs) as a formal model to characterize the interactions between the different components in an MEC environment. We use objectives specified in Probabilistic Alternating-Time Temporal Logic [38] with a verification tool to derive recovery strategies considering all possible execution scenarios of the model. For lower priority applications, we resort to a global recovery strategy with a greedy heuristic.

1.3.2 Verification

In addition to the contributions outlined above, we also put forward a framework for quantitative verification of service allocation policies. We propose a trace driven approach to derive a formal model of allocation policies. The interactions between the MEC environment components are modeled as a Stochastic Multi-Player Game utilizing which we define quantitative properties to produce probabilistic guarantees on performance metrics of allocation policies. We believe that verifying service allocation policies can aid a policy designer in gaining insights about how a specific policy performs in different scenarios.

In all the above scenarios, we use real-world benchmark datasets and state-of-the-art policies to demonstrate the effectiveness of our design and verification framework. The central theme of this thesis is to develop and use formal methods that allow efficient modeling, characterization and verification of MEC system descriptions in varying user service invocation scenarios, service request traffic conditions, and server failures.

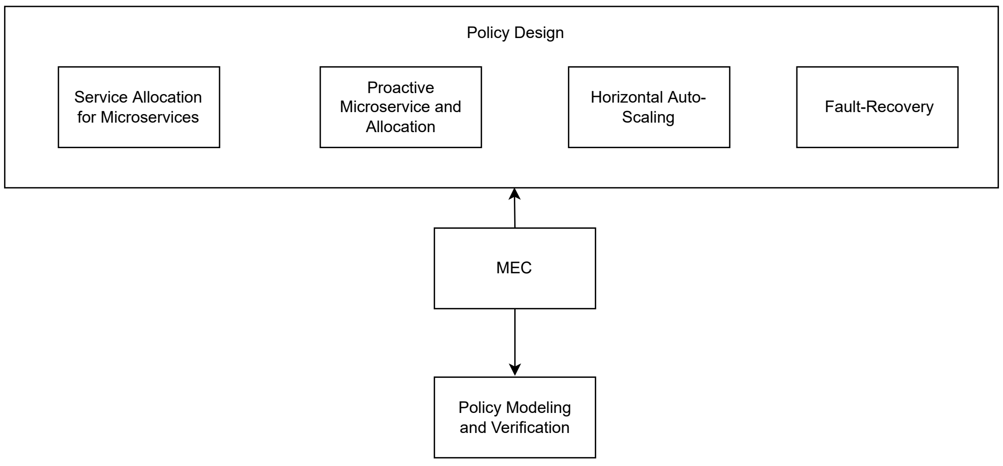

Policy Design
Service Allocation for Microservices
Proactive Microservice and Allocation
Horizontal Auto-Scaling
Fault-Recovery
MEC
Policy Modeling and Verification

FIGURE 1.1: Overview of Thesis Contributions

1.4 Organization of the dissertation

This dissertation is organized into 8 chapters. A summary of each chapter is as follows:

Chapter 1: This chapter contains an introduction and a summary of the major contributions of this work.

Chapter 2: A detailed study of relevant research in MEC and background concepts utilized in this work are presented here.

Chapter 3: This chapter presents an allocation policy for microservice based applications given an already existing service placement deployment.

Chapter 4: This chapter describes a proactive service placement and allocation policy for microservice based applications.

Chapter 5: This chapter presents a horizontal auto-scaling policy which ensures conformance to pre-specified latency requirements.

Chapter 6: This chapter describes a two-fold fault-recovery strategy for MEC.

Chapter 7: This chapter presents a framework for verification of service allocation policies.

Chapter 8: We summarize with conclusions and future directions on the contributions of this dissertation.

Chapter 2

Preliminaries and Background Work

In this chapter, we first present a few preliminary concepts related to the MEC architecture and MEC policies. We then present a discussion of the background techniques utilized in this thesis. Finally, we discuss about the datasets and benchmark applications utilized in our work.

2.1 MEC Architecture

MEC employs edge sites to allow computation, network, and storage resources to be placed in proximity of users of mobile and IoT devices, as shown in Figure 2.1, thereby effectively mitigating the latency experienced in offloading tasks to the cloud [3, 39, 40, 41, 42, 43, 44, 45]. Each edge site is powered by one or many physical machines referred to as edge servers. An

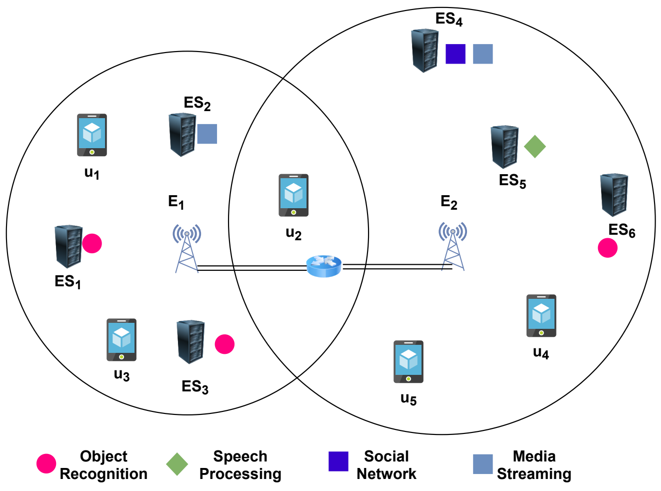


FIGURE 2.1: Multi-Access Edge Computing Architecture

Edge Site	Edge Server	Users
E_1	ES_1, ES_2, ES_3	u_1, u_2, u_3
E_2	ES_4, ES_5, ES_6	u_2, u_4, u_5

Server	Applications
ES_1	Object Recognition
ES_2	Media Streaming
ES_3	Object Recognition
ES_4	Social Network, Media Streaming
ES_5	Speech Processing
ES_6	Object Recognition

TABLE 2.1: MEC Architecture Configuration

edge site covers a specific geographical area so that mobile users within its coverage can connect to it via a one-hop wireless access point [3, 13, 46, 47] thereby enabling low latency access. The coverage areas of adjacent edge sites usually intersect to avoid blank areas not covered by any edge site. A mobile user located in an intersection area can connect to any one of the associated edge sites with low latency access. Edge sites communicate with each other via a backbone network. Users located outside the coverage area of an edge site can thus avail of farther away edge sites via this backbone network at the cost of additional access latency.

Example 2.1.1. In Figure 2.1, there are two edge sites E_1 and E_2, where E_1 is powered by three edge servers ES_1, ES_2 and ES_3 while E_2 is also powered by three edge servers ES_4, ES_5 and ES_6. The coverage areas of E_1 and E_2 are depicted by encompassing circles. User u_1 can access E_1 while u_2 can access both E_1 and E_2 with low latency. User u_1 can also access E_2 via the backbone network, however, with an additional access latency. ■

2.1.1 Application Provisioning Model in MEC

We now discuss the application provisioning model utilized in this thesis. Applications are broadly categorized into two types: monolithic applications and microservice applications [48]. Traditionally, applications utilize the monolithic architecture where the entire application functionality is encapsulated in a single binary. Recently, there has been a shift from the monolithic architecture to a microservice architecture where the application is split into a set of interacting microservices with inter microservice communication dependencies. Microservices are typically represented as a Directed-Acyclic Graph (DAG) depicting the workflow of the sequence of microservice based operations. In this thesis, we focus on applications whose workflows are defined by a linear sequence of microservices. Figure 2.2 depicts the workflow sequence of a representative MediaStreaming microservice based application. Service providers deploy application services on edge servers in the form of containers [3, 49]. Traditional monoliths are deployed as single containers on an MEC server while applications designed using the microservice architecture deploy each microservice as a separate container on MEC servers.

Example 2.1.2. Table 2.1 lists representative monolithic application services deployed on the various edge servers shown in Figure 2.1. For edge site E_1, two application instances of the Object Recognition Service are deployed on MEC servers ES_1 and ES_3. Such a deployment aids in load balancing and fault tolerance. Figure 2.3 depicts the containerized deployment of the various microservices corresponding to the MediaStreaming application in Figure 2.2. At edge site E_2, all containers for the MediaStreaming application are deployed on the same MEC server ES_5 while at edge site E_1, the containers are distributed amongst ES_1 and ES_2.

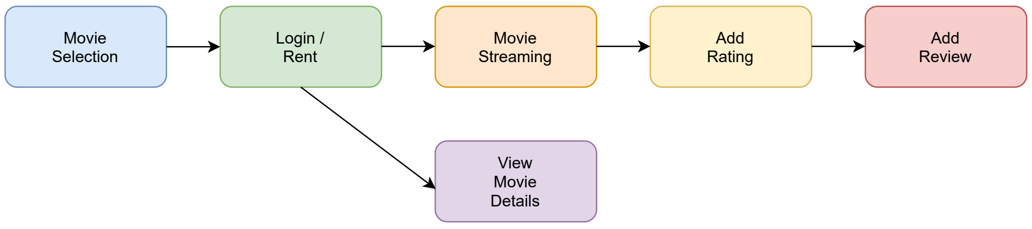


FIGURE 2.2: Representative Media Streaming Microservices Application

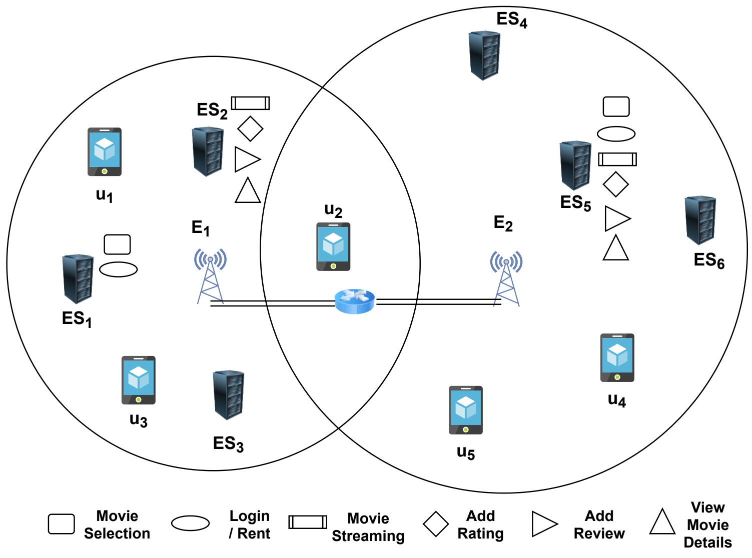


FIGURE 2.3: Representative Media Microservices Application on MEC servers

Note that each edge site in this example has a single application instance. Multiple application instances can also be deployed similar to monolithic applications. ■

2.1.2 Containerized Deployment

In monolithic applications, all services of an application are deployed as a single container, whereas in microservice based applications, each microservice is deployed as a singleton container. When a particular user invokes a service request pertaining to an application that is not deployed, the corresponding container has to be initialized on an MEC server. Additionally, the corresponding service registry has to be updated on a container orchestration system to reflect the deployment state of the services. On the other hand, if the corresponding container already exists on the edge server, a new task is spawned out of the existing container. Deploying containers and creating new tasks incur non-negligible latencies. MEC policies operate on top of the underlying MEC architecture governing the spectrum between users and MEC servers. In the following section, we discuss the various policies proposed in MEC literature.

2.2 MEC Policies

2.2.1 Service Placement

A service placement policy determines which application services are to be deployed on which MEC servers across edge sites. Figures 2.1 and 2.3 depict representative MEC placement configurations for monolithic application services and microservice based applications respectively. In recent times, several MEC service placement schemes have been proposed incorporating the static [2, 50] and dynamic [4, 51, 52, 53, 54] service contexts. In [2], the authors derived an approximation by incorporating rewards that are awarded when user requirements are honoured. In [4], the authors formulated a polynomial time approximation on a time-slotted model by jointly optimizing service placement and request scheduling. The works in [5, 55] considered data transfer and availability for making placement decisions. In [56] and [57], the authors studied the impact of base stations collaborating with each other. In [53], the authors considered multi-network scenarios as well to optimize service placement by incorporating network communication costs. The authors in [54] considered a Virtual-Reality based application and studied service placement strategies optimizing for the same. They proposed service placement of a Virtual-Reality application demonstrating gains in an application-specific scenario and studied service placement strategies optimizing for the same. They demonstrated their effectiveness over generic approaches in an application-specific scenario. Similarly, [52] proposed a service placement scheme in Software Defined Networks. In [58], a joint approach for network selection and service placement was considered with multiple network operators.

2.2.2 Service Allocation

Service allocation/routing deals with determining the assignment of service requests from users to already deployed services on MEC servers [6, 7, 8]. As a result, a service allocation approach presumes a service placement deployment.

Example 2.2.1. Consider the scenario in Figure 2.4, where user u_5 invokes the Object Detection application. A service allocation policy determines whether the service request is assigned to server ES_1, ES_3 or ES_6. ■

A number of allocation policies have been proposed in recent literature considering various optimization metrics such as the number of users allocated, QoE/QoS maximization, energy optimization, optimizing the number of re-allocations as users move about, and so on. Authors in [59] proposed optimal and approximate approaches for the network resource allocation

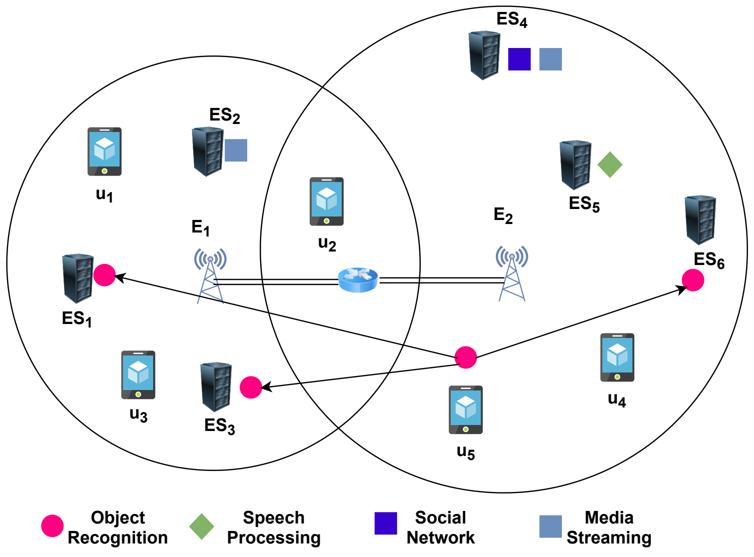


FIGURE 2.4: Service Allocation Policy

problem in MEC. In [60], the authors formulated a game-theoretic approach for the service allocation problem. In [8], the authors presented an Integer Linear Programming based approach for maximizing the average number of users allocated to MEC servers while minimizing the number of MEC servers on which a service provider would have to deploy the applications. They formulated the problem as a bin packing problem. In [7], instead of static QoS values, the authors considered dynamic QoS parameters. In [6], the authors studied dynamic scenarios where edge devices are mobile and re-allocations are possible between MEC servers. In [9], the authors demonstrated the benefits of a learning-based allocation strategy. In [61], the authors studied the trade-off between accuracy and processing times in Augmented Reality applications. In [47] the authors utilized a game-theoretic approach for user allocation. In [50], the authors proposed Linear Programming approaches to provide a fast approximation for solving the service placement problem.

2.2.3 Service Migration

Though service placement entails deciding which servers to use to deploy services, MEC adds to the complexity with user mobility. As users move between locations, static service placements can no longer provide QoS benefits. Service relocation is a method of dynamically moving resources to accommodate end-user mobility. An allocation policy also incorporates a migration component. In the context of service allocation, migration refers to state transfer (user-specific runtime data utilization) due to change in the request-server binding, i.e., movement of an already allocated service request to a different server as opposed to migrating services placed on a server [62].

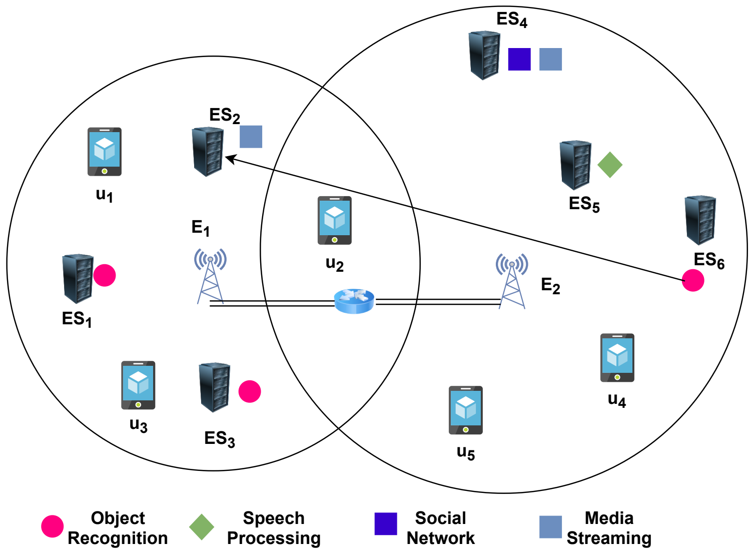
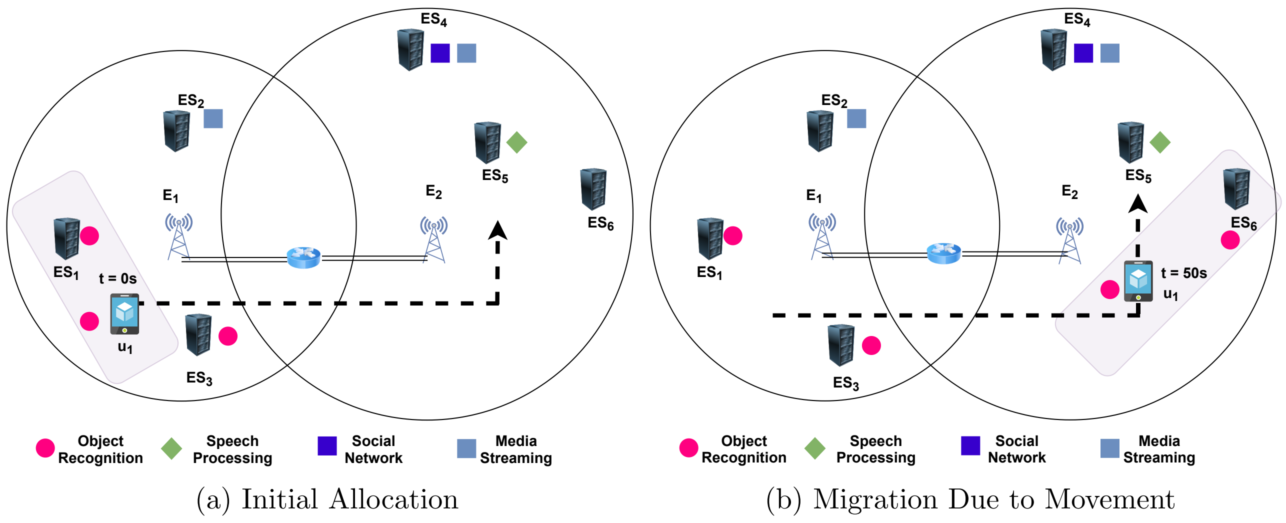


FIGURE 2.5: Service Migration Policy

Example 2.2.2. Consider the scenario in Figure 2.5, where the Object Detection application is migrated from server ES_6 of edge site E_2 to server ES_2 of site E_1. Such a scenario could be triggered when the number of service invocations of the Object Detection service is higher in the vicinity of E_1. Additionally, consider the scenario in Figure 2.6a where the Object Recognition service request from u_1 has been allocated to ES_1 at time $t = 0s$. Consider the timepoint $t = 50s$ during the course of the trajectory followed by u_1 indicated by the dotted line. The migration component of an allocation policy performs a state-aware migration of the Object Recognition task executing at ES_1 to server ES_6 as depicted in Figure 2.6b. ■



(a) Initial Allocation (b) Migration Due to Movement

FIGURE 2.6: Service Migration in the context of Service Allocation

The authors in [63] formulated service placement and migration using the MDP model and developed heuristics for multi-user and multi-service models. However, they assumed a monolithic service being used throughout the entire duration of the users' invocation which is not necessarily true for a microservice architecture. In [6], a mobility-aware monolithic service migration strategy was proposed taking a direction vector approach. In [64] the authors studied the benefits of deep reinforcement learning approaches for service migration, but only for single-user migrations. In [9] a microservice reinforcement learning-based approach was considered.

2.2.4 Computation Offloading and Task Scheduling

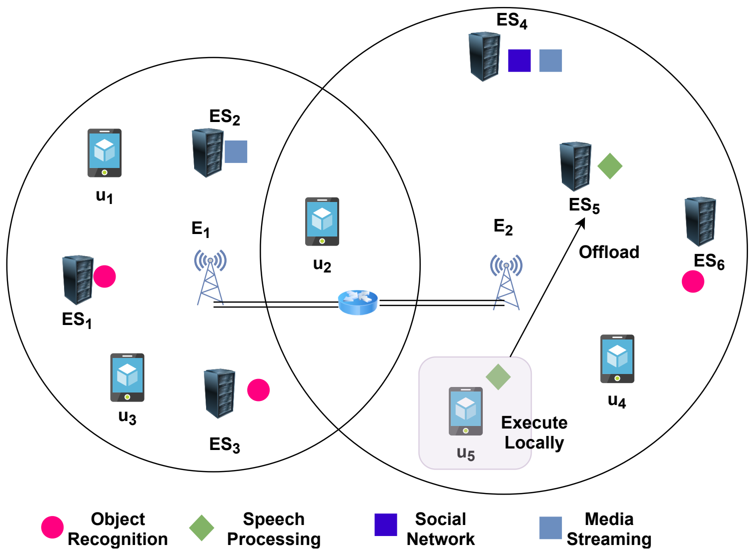


FIGURE 2.7: Computation Offloading Policy

A computation offloading policy determines whether a particular task should be performed locally on the devices or offloaded to an MEC server for execution. A computation offloading policy operates in conjunction with a task scheduling policy which determines the order of execution of tasks. While classical task scheduling policies target individual devices/servers, MEC complicates such scenarios by partitioning task execution between devices and servers.

Example 2.2.3. Consider the scenario in Figure 2.7, where user u_5 invokes the speech processing application. A particular offloading policy in such a circumstance would decide whether to execute the speech processing task on u_5 itself or to offload the processing task to ES_5 since the Speech Recognition application service is deployed on the server ES_5. ■

Offloading in Mobile Cloud Computing [65, 66, 67] and Multi-Access Edge Computing [68, 69, 70, 71, 72, 73] have both been extensively studied concerning what/when/how to offload workloads from handheld devices to the cloud or edge [55]. A number of different approaches to offloading have been proposed. In [74], the authors focused on the energy efficiency of edge devices. Wang et al. [75] considered minimizing each MEC server's energy consumption while satisfying QoS requirements. In [76], the authors dealt with vehicular networks to propose domain-specific algorithms for offloading, involving data centers in vehicular services. In [67], the authors considered the objective of minimizing cost. Authors in [70] introduced a unique perspective on energy consumption with simultaneous harvesting considering devices utilizing battery resources while harvesting other energy sources. In [77] the authors considered joint offloading and service placement where dependencies exist between tasks. Several approaches to task scheduling have been proposed both in traditional contexts (in the absence of offloading schemes) [78, 79, 80, 81, 82] as well as in the MEC context [83, 84, 85, 86]. In [78] the authors studied the impact of temperature on reliability and proposed a task partitioning scheme for embedded systems. In [87], the authors proposed a model checking based approach for temperature and energy consumption aware scheduling for multi-tasking systems. In [83, 84, 86] joint offloading and task scheduling approaches were considered. In [85], a minimum energy consumption based task scheduling was proposed.

2.2.5 Auto-Scaling

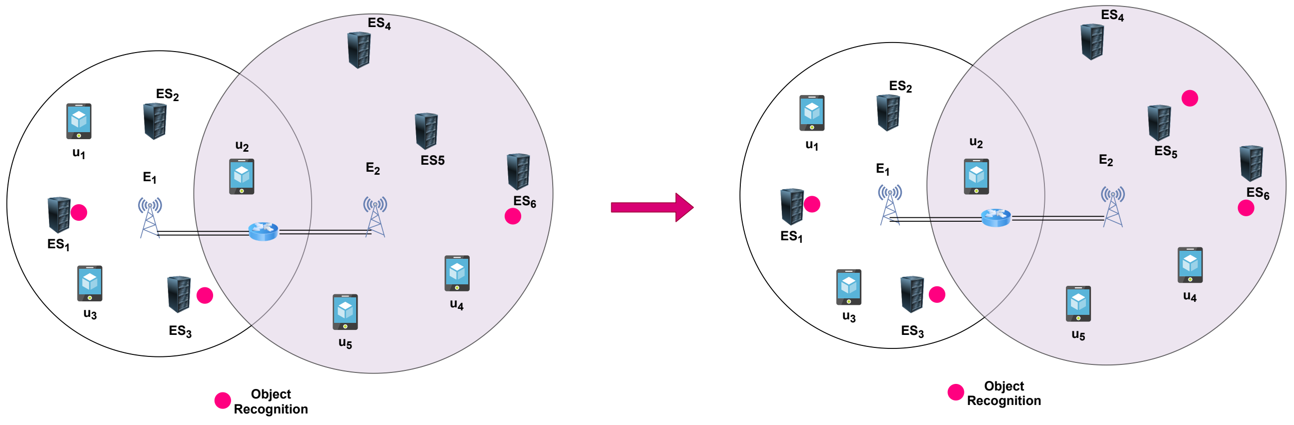


FIGURE 2.8: Auto-Scaling Policy

While service placement, service allocation, and computation offloading deal with optimizing latency and other parameters, fine-grained latency control is easier said than done, since the actual latencies incurred by the users not only depend on the network characteristics but also on various other variable aspects at the edge server, some of the factors being the server's computational capability, current resource contention, current service workload, service request payloads, requesting traffic density, and so forth. Thus, it is quite difficult to optimize to a fixed latency for best response/performance by analytically modeling each such aspect [88]. Auto-scaling policies are effective tools to characterize such dynamic aspects [12, 13] by observing the runtime latencies and adjusting container instances accordingly at a coarser level than tuning latencies back and forth depending on the variabilities mentioned above. An auto-scaling policy adapts to traffic changes dynamically by automatically provisioning and de-provisioning resources [10]. A vertical auto-scaling policy mandates the dynamic addition or removal of resources such as CPU or memory to a container. A horizontal auto-scaling policy, on the other hand, dynamically provisions/de-provisions container instances of an application. Since edge servers are relatively more resource-constrained as compared to traditional data-center servers, vertical auto-scaling is not well suited to an MEC environment [6]. A horizontal auto-scaling policy thus serves as a viable mechanism to alleviate resource contention with enhanced load balancing with newly spawned container instances [13].

Example 2.2.4. Consider the scenario in Figure 2.8 where the Object Recognition application is deployed on several MEC servers. In Figure 2.8, for edge site E_2, initially 1 Object Recognition instance is deployed on server ES_6. Depending on user traffic variation and the incurred latencies within the edge site, the policy increases the number of container instances to 2 with an additional application instance deployed at ES_5. The auto-scaling policies for other edge sites work similarly. ■

Traditionally, Rule-Based auto-scaling policies have been used by major cloud service providers. However, such policies result in over or under-provisioning of resources and can lead to inefficient resource utilization [10]. Additionally, traditional rule based auto-scaling policies incorporate a *static* set of rules governing the provisioning/de-provisioning of resources. Thus, rule-based policies also require a designer to manually enumerate a complex set of rules to consider system characteristics. Designing a set of rules exhaustively covering all such scenarios is practically infeasible. To avert such scenarios, Reinforcement Learning based *dynamic* solutions have been proposed [12]. However, such approaches do not provide any guarantees to ensure latency requirements are adhered to [89]. Probabilistic Model Checking is utilized to synthesize auto-scaling policies to ensure such latency requirements [11]. However, such an approach is not suitable for MEC environments where the computational overhead of Probabilistic Model Checking renders it unsuitable for real-time applications [14]. The authors in [13] considered joint offloading and auto-scaling in MEC where an additional energy harvesting mechanism is available. However, they neither considered application-specific latency threshold requirements nor considered access latencies between MEC servers. The authors in [12] designed an RL-based horizontal auto-scaling strategy for traditional cloud computing environments, but did not take into account the variable intercommunication latencies between MEC servers.

2.2.6 Fault Tolerance

A fault tolerance policy addresses the resilience mechanism of an MEC environment to faults. Typically, fault tolerance is implemented by replicating application container instances across multiple MEC servers such that in the event of a failure, other servers ensure continued availability. An integral component of a fault tolerance policy is a fault recovery policy which determines the MEC servers to be utilized to re-initialize containers which were deployed prior to the occurrence of a failure.

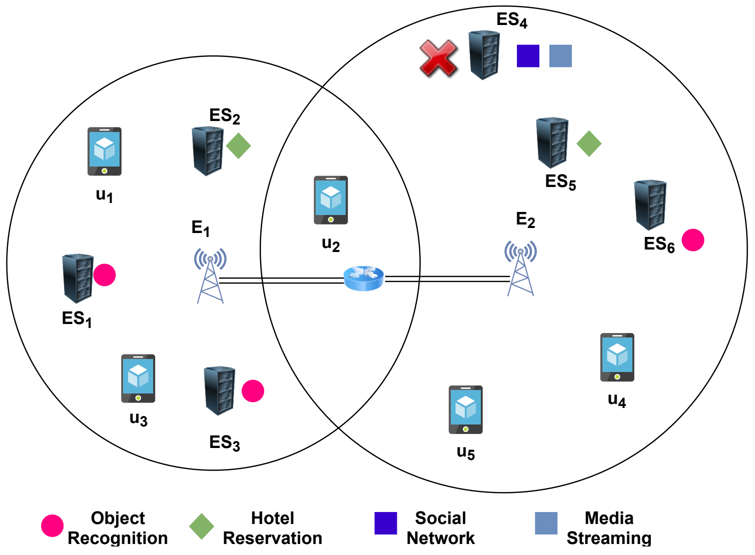


FIGURE 2.9: Fault Recovery Policy

Example 2.2.5. In Figure 2.9, multiple application instances of the Object Detection application are deployed on servers ES_1, ES_3 and ES_6 to cater to fault tolerance. When server ES_4 fails, the Social Network and the Media Streaming containers have to be re-initialized. ■

The issue of fault tolerance has received relatively less attention in the context of MEC. Prior work on fault tolerance in pervasive computing environments either considered replication techniques in which identical applications are deployed at multiple servers [15] or considered failure avoidance by forecasting the occurrences of failures [16]. All the above techniques targeted traditional pervasive computing environments. They neither considered the impact of geographical locations of MEC servers nor the impact of deploying multiple applications contending for shared resources at such servers. Several works have investigated the issue of fault prediction [15, 90, 91, 92] in both traditional cloud and edge computing environments. However, such studies considered the issue of *when* faults occur. Fault-Recovery, on the other hand, which deals with *how* to handle faults, has been relatively less examined. Recently, the authors in [17] proposed an Edge-based IoT architecture catering to fault tolerance in IoT-Edge environments.

They incorporated on-the-fly fault handling by re-allocation. However, the authors only examined singleton failures and ignored the possibility that multiple points of failure can co-exist in an edge computing distributed heterogeneous environment.

In this thesis, we first propose service placement and service allocation policies for microservice based applications by exploiting the intricacies arising from the dependency structure of microservices. Secondly, we design an auto-scaling policy to ensure adherence to application specific latency requirements. Thirdly, we propose a fault recovery policy by proactively considering the possibility of multiple simultaneous failures. Finally, we present a framework for modeling and verification of service allocation policies.

In the following section, we present some of the background formalisms, specifications and algorithms that serve as the foundation for our work. We first discuss Probabilistic Model Checking (PMC) and then present an overview of formal representations of specifications utilized in PMC.

2.3 Probabilistic Model Checking

A formal technique for automated verification and quantitative analysis of probabilistic systems is probabilistic model checking. A probabilistic model checking algorithm's goal is to determine whether a probabilistic model of a system satisfies a probabilistic temporal logic property or, depending on the type of property, to which numerical value (probability or reward) the property evaluates. We first discuss about probabilistic models of systems.

2.3.1 Discrete-Time Markov Chain (DTMC)

Definition 2.1 [DTMC:]
A DTMC is a tuple $\mathcal{D} = (S,\ P,\ A,\ L,\ s_0)$ where:

- *S is a finite set of discrete states where s_0 is the initial state.*
- *$P : S \times S \rightarrow [0, 1]$ denotes the probability distribution on transitions such that for all states s : $\sum_{s' \in S} P(s, s') = 1$.*
- *AP is a set of Boolean Atomic Propositions (AP).*
- *$L : S \rightarrow 2^{AP}$ is an AP labelling function, labelling states with the APs true in that state.*

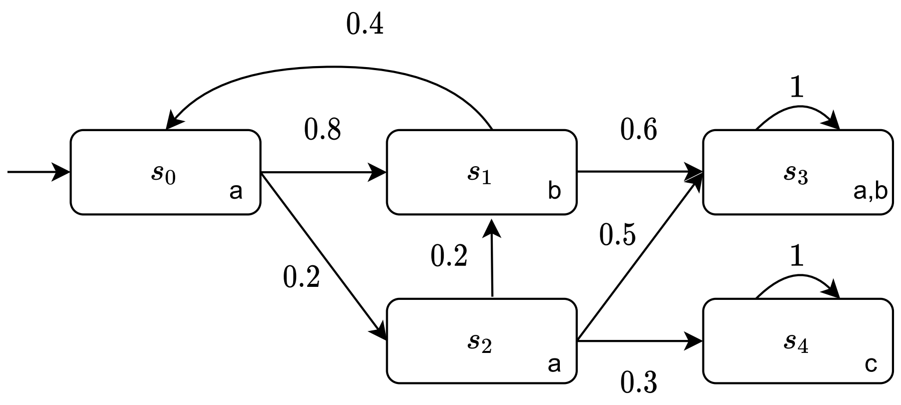


FIGURE 2.10: Example DTMC

This definition implicitly assumes that $\mathcal{D}$ is time-homogeneous [93], i.e., P and L do not vary over time. The transition probability $\mathcal{D}$ is solely dependent on the past through the present state, which is known as the Markov Property. At the outset, the DTMC is in the initial state $s_0 \in$ S. At each point of time, the successor state of the current state is chosen according to the probability distribution P.

Example 2.3.1. Figure 2.10 shows a DTMC with 5 states. The set of outgoing transitions from each state form a probability distribution. The labelling L associated with each state is denoted at the right-hand bottom corner of each state. L is a mapping from the set of states S to the set AP = {a,b,c}. ■

2.3.2 Markov Decision Processes (MDPs)

While DTMCs incorporate variations induced by probability distributions over discrete time-steps, MDPs allow incorporation of non-deterministic choices.

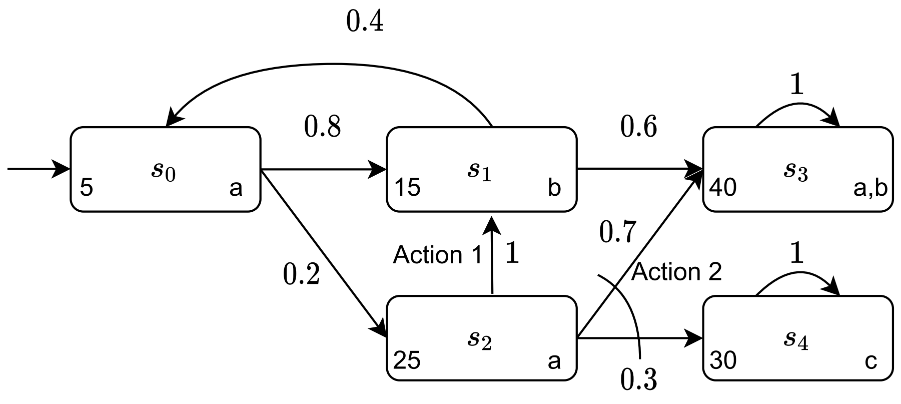


FIGURE 2.11: Example MDP

Definition 2.2 [**MDP:**]
An MDP is a tuple $\mathcal{M} = (S, \Lambda, P, AP, L, R, s_0)$ *where*

- S *is a finite set of states.*
- Λ *is a set of actions.*
- $P : S \times \Lambda \times S \to [0,1]$ *is the transition probability function such that for all states* $s \in S$ *and actions* $\lambda \in \Lambda$*:* $\sum_{s \in S} P(s, \lambda, s') = 1$.
- AP *is a set of Boolean Atomic Propositions.*
- $L : S \to 2^{AP}$ *is an AP labelling function, labelling states with the APs true in that state.*
- $R : S \to \mathbb{R}$ *is the reward associated with each state* $s \in S$.
- s_0 *is the initial state.*

Example 2.3.2. Figure 2.11 shows an MDP similar to the DTMC depicted in Figure 2.10 with 5 states. In State s_2, there are two Actions, Action 1 and Action 2, each characterized by an individual probability distribution. The labelling function is identically represented like a DTMC with the set of AP = {a, b, c}. The left hand bottom corner of each state depicts the reward associated with each state. ■

When modeling complex systems, it is more favorable to use a number of MDPs to capture the behaviours of different system components instead of adopting a monolithic approach. MDP composition can then be used to merge those models after defining the synchronization rules among them. In our work, we utilize parallel composition of two MDPs defined as follows:

Definition 2.3 [**Composition of MDP:**]
Let $\mathcal{M}_1 = (S_1, \Lambda_1, P_1, AP_1, L_1, R_1, s_0^1)$ *and* $\mathcal{M}_2 = (S_2, \Lambda_2, P_2, AP_2, L_2, R_2, s_0^2)$ *be two MDPs where* s_0^1 *and* s_0^2 *denote the initial states. The parallel composition of the MDPs is defined as:* $\mathcal{M}_1 \mid\mid \mathcal{M}_2 = (S_1 \times S_2,\ \Lambda_1 \cup \Lambda_2,\ AP_1 \cup AP_2,\ L_{\mathcal{M}_1 \mid\mid \mathcal{M}_2},\ R_{\mathcal{M}_1 \mid\mid \mathcal{M}_2},\ P,\ (s_0^1, s_0^2))$. *Consider two states* $(s_a, s_b) \in S_1 \times S_2$, $(s_c, s_d) \in S_1 \times S_2$. *A probabilistic transition with value* $P_1(s_a, s_c) \times P_2(s_b, s_d)$ *exists from* (s_a, s_b) *to* (s_c, s_d) *iff there exists a probabilistic transition from* s_a *to* s_c *with probability* $P_1(s_a, s_c)$ *in* S_1 *and a probabilistic transition from* s_b *to* s_d *with probability* $P_2(s_b, s_d)$ *in* S_2.

The composition of multiple MDPs is carried out by composing two MDPs at a time. Intuitively, MDPs are composed by synchronizing on common actions (the probability function is then the product of distributions for P_1 and P_2) and interleaving otherwise. More about MDP semantics and composition can be found in [37].

2.3.3 Probabilistic Computation Tree Logic (PCTL)

Properties of a probabilistic model are specified using an extension of temporal logic called PCTL [37]. We first formally define a PCTL formula as follows:

Definition 2.4 [**PCTL:**]
The syntax of PCTL is:

$$\Phi ::= \mathit{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid P_{\sim p}[\phi]$$
$$\phi ::= \mathrm{X}\Phi \mid \Phi\ \mathrm{U}^{\leqslant k}\ \Phi$$

where a is an atomic proposition, the operator $\sim \in \{<, \leqslant, \geqslant, >\}, p \in [0,1]$ and $k \in \mathbb{N} \cup \{\infty\}$, where $\mathbb{N}$ is the set of natural numbers.

Using the above syntax of a PCTL formula, additional operators can be defined. One such operator is the "eventually" operator denoted as $F\ \Phi = \mathit{true}\ U\ \Phi$. We now discuss model checking algorithms for verifying properties expressed in PCTL.

2.3.4 Model Checking PCTL properties on DTMCs and MDPs

We first define the Satisfaction relation on DTMCs using which the model checking algorithm computes the probability of satisfaction of a particular PCTL property.

Definition 2.5 [**DTMC Path:**]
Let $\mathcal{D}$ be a labelled DTMC. A path of $\mathcal{D}$ is defined as a sequence of states $s_0, s_1, \ldots$ such that $\forall s_i, s_{i+1}, P(s_i, s_i + 1) > 0$. We utilize $\omega(i)$ to denote the i-th state of the path ω , $\forall i \geq 0$. We denote by $\mathrm{Path}^{\mathcal{D}}(s)$ the set of all paths of $\mathcal{D}$ from state s. For any state $s \in S$, the satisfaction relation $\models$ is defined inductively by:

$$\begin{aligned}
s \models \mathit{true} &\quad \text{for all } s \in S \\
s \models a &\Leftrightarrow a \in L(s) \\
s \models \neg\Phi &\Leftrightarrow s \not\models \Phi \\
s \models \Phi \wedge \Psi &\Leftrightarrow s \models \Phi \wedge s \models \Psi \\
s \models P_{\sim p}[\phi] &\Leftrightarrow P^{\mathcal{D}}(s, \phi) \sim p \sim \in \{<, \leqslant, \geqslant, >\}
\end{aligned}$$

where: $P^{\mathcal{D}}(s,\phi) \overset{def}{=} P_s\left\{\omega \in \text{Path}^{\mathcal{D}}(s) \mid \omega \models \phi\right\}$ *and for any path* $\omega \in \text{Path}^{\mathcal{D}}(s)$:

$$\omega \models \mathrm{X}\Phi \Leftrightarrow \omega(1) \models \Phi$$
$$\omega \models \Phi\ \mathrm{U}^{\leqslant k}\Psi \Leftrightarrow \exists i \in \mathbb{N}(i \leqslant k \wedge \omega(i) \models \Psi \wedge \forall j < i.(\omega(j) \models \Phi)).$$

Intuitively, PCTL formulae are interpreted on the Atomic Propositions associated with the states of paths of a DTMC or MDP as depicted in Figure 2.12. The operator Next (X) is utilized to describe the AP labelling the immediate subsequent state of a path. Similarly, the Until (U) operator characterizes a subpath. Figure 2.12 illustrates the satisfaction relation of PCTL operators: the singleton AP, the next operator (X) and the until operator (U). Note that the next operator considers the AP in the immediate next state $\omega(1)$ of the start state on a path while the until operator considers a fragment of the path with APs a and b.

Theorem 2.1. *The Model Checking Algorithm with inputs as a DTMC $\mathcal{D}$ and a PCTL formula Φ is sound and complete.* ♦

The model checking algorithms for PCTL are well-established and detailed in [37, 94, 95]. We reproduce here from [93] a summary of the model checking algorithms for the until operator since this is used in our contributory chapters. Note that the model checking of the eventually (F) operator follows since the eventually operator can be written in terms of the until operator.

The inputs to the algorithm are a labelled DTMC $\mathcal{D}$ and a PCTL formula Φ. The output is the set of states $Sat(\Phi) = \{s \in S, s \models \Phi\}$, defined as the set of states of the model which satisfy Φ. The algorithm proceeds by checking whether each state in S satisfies the formula.

Model Checking $P_{\sim p}[\Phi\ \mathrm{U}^{\leqslant k}\ \Psi]$: For such formulae we need to determine the probabilities $Prob^{\mathcal{D}}(s, \Phi\ \mathrm{U}^{\leqslant k}\ \Psi)$ for all states s where $k \in \mathbb{N} \cup \{\infty\}$, where $k = \infty$ is utilized for infinite path lengths.

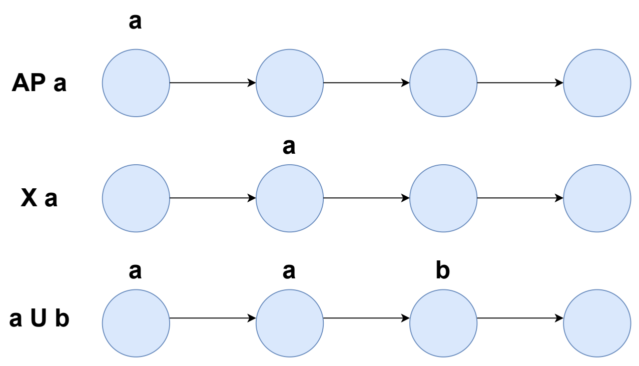


FIGURE 2.12: PCTL Properties with respect to the Atomic Proposition Set {a, b}

We denote by $\pi^{\mathcal{D}}_{s,k}(s')$ the transient probability in $\mathcal{D}$ of being in state s' after k steps when starting in s, that is:

$$\pi^{\mathcal{D}}_{s,k}(s') = \mathrm{Pr}_s \left\{ \omega \in \text{ Path }^{\mathcal{D}}(s) \mid \omega(k) = s' \right\}.$$

The probabilities $Prob^{\mathcal{D}}(s, \Phi \text{ U}^{\leqslant k} \Psi)$ can then be expressed as the transient probabilities of a DTMC utilizing a PCTL driven transformation of the DTMC defined as follows:

Definition 2.6 [**PCTL driven DTMC Transformation:**]
For any DTMC $\mathcal{D}$ and PCTL formula Φ, let $\mathcal{D}[\Phi]$ denote the transformed DTMC where, if $s \not\models \Phi$, then $\mathbf{P}[\Phi](s, s') = P(s, s')$ for all $s' \in S$, and if $s \models \Phi$, then $\mathbf{P}[\Phi](s, s) = 1$ and $\mathbf{P}[\Phi](s, s') = 0$ for all $s' \neq s$ Using the transient probabilities and this transformation we define the vector $Prob^{\mathcal{D}}(s, \Phi \text{ U} \leqslant^k \Psi)$ as follows:

$$Prob^{\mathcal{D}}\left(s, \Phi \text{U}^{\leqslant k} \Psi\right) = \sum_{s' \models \Psi} \pi^{\mathcal{D}[\neg\Phi \vee \Psi]}_{s,k}(s')$$

These probabilities can now be computed using the following matrix and vector multiplications from the original probability transitions of the DTMC $\mathcal{D}$:

$$Prob^{\mathcal{D}}\left(\Phi \text{ U}^{\leqslant k} \Psi\right) = (P[\neg\Phi \vee \Psi])^k \cdot \underline{\Psi}$$

where $\underline{\Psi}$ is a column vector such that:

$$\underline{\Psi} = \begin{cases} 1, \text{if } s \in Sat(\Psi) \\ 0, \text{otherwise} \end{cases}$$

This product is computed in an iterative fashion as follows:

$$P[\neg\Phi \vee \Psi] \cdot (\cdots (P[\neg\Phi \vee \Psi] \cdot \underline{\Psi}) \cdots)$$

Case when $k \in \mathbb{N}$: For $s \in S$ and $k \in \mathbb{N}$: the probabilistic satisfaction of $P^{\mathcal{D}}(s, \Phi \text{ U}^{\leqslant k} \Psi)$ is defined as:

$$P^{\mathcal{D}}(s, \Phi \text{ U}^{\leqslant k} \Psi) = \begin{cases} 1 \text{ ; if } s \in \text{Sat}(\Psi) \\ 0 \text{ ; if } k = 0 \text{ or } s \in \text{Sat}(\neg\Phi \wedge \neg\Psi) \\ \sum_{s' \in S} P(s, s') \cdot Prob^{\mathcal{D}}\left(s', \Phi \text{ U}^{\leqslant k-1} \Psi\right) \text{ ; otherwise.} \end{cases}$$

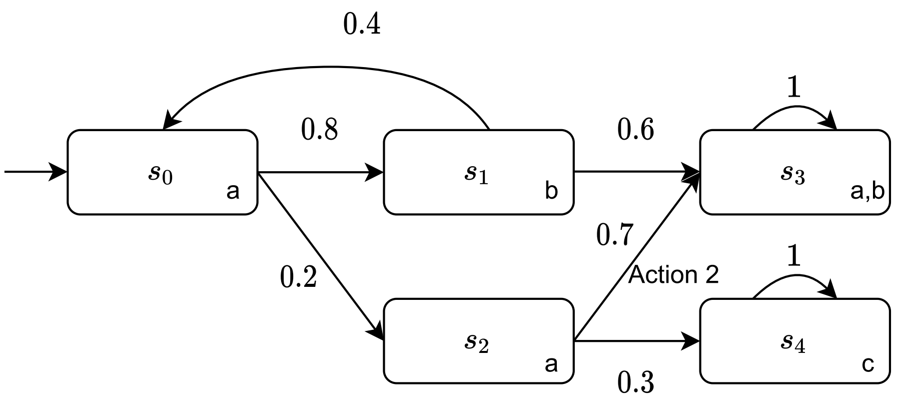


FIGURE 2.13: Embedded DTMC of MDP

Case when $k = \infty$: The probabilities $Prob^{\mathcal{D}}(s, \Phi \text{ U } \Psi)$ can be computed as the solution of the linear equation system using least squares where $Prob^{\mathcal{D}}(s, \Phi \text{ U } \Psi)$ is defined as:

$$Prob^{\mathcal{D}}(s, \Phi \text{ U } \Psi) = \begin{cases} 1 & \text{if } s \in \text{Sat}(\Psi) \\ 0 & \text{if } s \in \text{Sat}(\neg\Phi \wedge \neg\Psi) \\ \sum\limits_{s' \in S} P(s, s') \cdot Prob^{\mathcal{D}}(s', \Phi \text{ U } \Psi) & \text{otherwise.} \end{cases}$$

The solutions of this system of equations determine the satisfaction probabilities of the operator until when $k = \infty$ [93]. Model Checking of the F operator is computed using the algorithm for the until operator. For an MDP M, the resolution of non-deterministic choices at each state produces an embedded DTMC $\mathcal{D}$. Figure 2.13 depicts an embedded DTMC for the MDP depicted in Figure 2.11. In this case, in state s_2, the non-deterministic choice is resolved by selecting Action 2. Thus, from state s_2, the outgoing transitions now form a probability distribution over s_3 and s_4. The Model Checking algorithms for MDPs consider all such embedded DTMCs to calculate the satisfaction relation of each PCTL formula using the DTMC satisfaction set construction outlined earlier.

2.3.5 Stochastic Multi-Player Games (SMGs)

The DTMC and MDP formulations characterize individual systems. However, they do not capture the dynamic interactions between the different entities of a large system. A Stochastic Multi-Player Game (SMG) is a generalization of an MDP which allows each state of the MDP to be controlled by entities called *players*, allowing effective formal characterization of the outcome of actions of a player on the other players. We first formally define a SMG as follows:

Definition 2.7 [**SMG:**]
A SMG is defined as a tuple $G = (Y, \Gamma, Act, P)$ *where:*

- Y *is a finite set of players.*
- Γ *is a finite set of states, partitioned into disjoint sets of states* Γ_v*, where* $v \in Y$*.*
- Act *is a finite set of actions.*
- $P : \Gamma \times Act \times \Gamma \to [0, 1]$ *is a partial transition function.*

A state $\gamma \in \Gamma_v$ is controlled by player v, if the actions from $\gamma \in \Gamma_v$ are controlled by v. SMGs are thus a generalization of MDPs where each state is controlled by a particular player [38].

2.3.6 Probabilistic Alternating-Time Temporal Logic with Rewards (rPATL)

Although PCTL can be used to design properties characterizing DTMCs and MDPs, it does not generalize to SMGs where properties quantifying behaviours of individual players are involved. To express properties for SMGs we use rPATL - Probabilistic Alternating-Time Temporal logic with Rewards [38]. Properties in rPATL are specified identically to PCTL with the additional annotation of explicitly specifying a player for whom the property is to be evaluated.

Definition 2.8 [**rPATL:**]
The syntax of rPATL is given by the grammar:

$$\begin{aligned} \phi &::= true|a|\neg\phi|\phi \wedge \phi|\langle\langle C\rangle\rangle \mathrm{P}_{\bowtie q}[\psi] \mid \langle\langle C\rangle\rangle \mathrm{R}^{r}_{\bowtie x}\,[\ \mathrm{F}^{\star}\phi] \\ \psi &::= \mathrm{X}\phi\ \left|\phi \mathrm{U}^{\leq k}\phi\right|\ \phi\mathrm{U}\phi \end{aligned}$$

where $a \in AP, C \subseteq Y, \bowtie \in \{<, \leq, \geq, >\}, q \in \mathbb{Q} \cap [0, 1], x \in \mathbb{Q}_{\geq 0}, r$ *is a reward structure,* $\star \in \{0, \infty, c\}$ *and* $c, k \in \mathbb{N}$*,* $\mathbb{Q}$ *represents the set of rational numbers,* $\mathbb{N}$ *represents the set of natural numbers and* $[0, 1]$ *represents the closed interval comprising all rational numbers in the closed interval* $[0, 1]$ *(both inclusive).*

In a rPATL property, $\langle\langle C\rangle\rangle$ specifies that we are interested in analyzing the rewards associated with the PCTL formula ϕ for the player C.

2.3.7 Model Checking rPATL Properties on SMGs

The model checking algorithm for rPATL verification on SMGs proceeds in a manner similar to the model checking algorithm for a PCTL property by exploring the satisfaction relation of states with respect to the property with respect to a player v. It then utilizes a value iteration approach to compute the associated reward values. Value iteration proceeds by repeatedly solving the set of equations while considering the difference in values in between iterations until the values obtained between successive iterations are within a specific bound. We reproduce from [38] the following theorem that asserts the generation of a unique solution of a given SMG for a given rPATL property using model checking.

Theorem 2.2. *The Model Checking Algorithms [38] with inputs as a SMG G and the rPATL formula $\Phi = \langle\langle v \rangle\rangle R_{\sim r}[F\phi]$ produces the unique solution to the game G.* ♦

In our following chapters, we use these model checking algorithms as verification procedures. In the following section, we briefly discuss about Reinforcement Learning where MDP models are utilized, however, the probability distributions and reward functions are unknown.

2.4 Reinforcement Learning

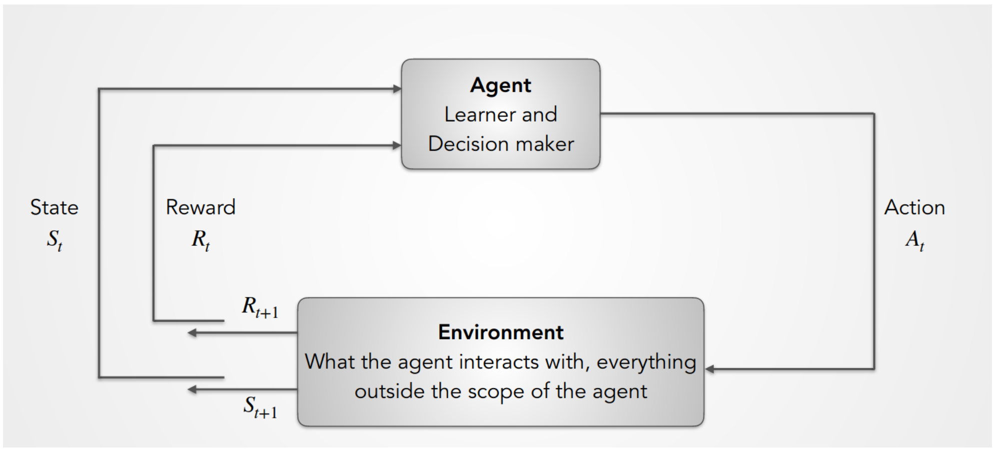


FIGURE 2.14: Overview of Reinforcement Learning

Reinforcement learning (RL) [34] is a branch of machine learning that studies how intelligent agents can operate in a given environment to maximize the reward accumulated by repeatedly interacting with the environment. RL differs from supervised learning in that it does not need the presentation of labelled input/output pairings or the explicit correction of sub-optimal behaviours. Instead, achieving a balance between exploration (of unexplored region) and exploitation (of information accumulated) is the priority of RL based approaches. The goal of reinforcement learning is for the agent to discover an optimal (or nearly optimal) strategy that maximizes the "reward function" or other user-provided reinforcement signals that accumulate from the immediate rewards. In RL, an agent interacts with the environment by executing decisions and adjusting the decision making over time in accordance with the reward signal. RL algorithms are either model-based or model-free. Model-free algorithms rely on direct interactions with the environment while a model-based algorithm makes use of previous interactions with the environment to predict rewards and next states. As demonstrated in Figure 2.14, the agent A at timepoint t in state S_t, executes an action A_t, as a result of which it transitions to a state S_{t+1} and receives a reward R_{t+1}. Such actions correspond to real time interactions with the environment in model-free algorithms. A model-based algorithm utilizes these interactions to build a model of the environment.

2.4.1 Q-Learning

Q-Learning is a model-free RL algorithm. It is utilized to learn the reward of an action in a particular environment state by continuous interaction in the form of exploration-exploitation. Q-learning finds an optimal policy for any finite MDP by maximising the expected value of the total reward across consecutive steps, commencing from the current state.

Algorithm 1: Q-Learning

```
1 Initialize $Q(S, \Lambda)$, $\forall s \in S, \forall \lambda \in \Lambda$
2 $s \leftarrow$ initialize state
3 while true do
4     $\lambda \leftarrow \epsilon$-greedy$(s, Q)$
5     Observe the next state $s'$ and the reward obtained
6     $Q(s,\lambda) \leftarrow Q(s,\lambda) + \zeta\,[r + \gamma\ \text{argmax}_\lambda[Q(s',\lambda)] - Q(s,\lambda)]$
7     $s \leftarrow s'$
```

Q-learning essentially estimates the optimal Q-function, Q, by its sample averages. A simple ϵ-greedy action selection method is utilized where at any decision step i, with probability ϵ, Q-learning chooses a random action to improve its knowledge of the application, whereas, with probability $1-\epsilon$, it chooses the action greedily by exploiting its knowledge about the application, i.e., $\lambda = \text{argmax}_\lambda\ Q(s, \lambda)$. Most of the time, the ϵ-greedy policy selects the best known action for a particular state, while it favors the exploration of sub-optimal actions with low probability.

At the end of each time slot i, $Q(s, \lambda)$ is updated as follows:

$$Q(s, \lambda) \leftarrow Q(s, \lambda) + \zeta\,[r + \gamma \ \text{argmax}_\lambda[Q(s', \lambda)] - Q(s, \lambda)] \tag{2.1}$$

where ζ is the learning rate assigned to the agent. The equation updates the Q-value of state s by determining the action corresponding to the highest Q-value among all successor states ($\text{argmax}_\lambda Q(s', \lambda)$), which is discounted by γ and updated according to the reward r observed from the environment. The Q-Learning algorithm is summarized in Algorithm 1.

2.4.2 Dyna-Q Algorithm

Algorithm 2: Dyna-Q

```
1  Initialize $Q(S, \Lambda)$ and $Model(S, \Lambda)$, $\forall s \in S, \forall \lambda \in \Lambda$
2  while true do
3      $s \leftarrow$ observe the application state
4      $\lambda \leftarrow \epsilon$-greedy$(s, q)$
5      Observe the next state $s'$ and the reward obtained
6      $Q(s, \lambda) \leftarrow Q(s, \lambda) + \zeta\,[r + \gamma \ \text{argmax}_\lambda[Q(s', \lambda)] - Q(s, \lambda)]$
7      Model $(s, \lambda) \leftarrow r, s'$
8      for $i = 0 \dots n$ do
9          $s \leftarrow$ random state previously observed
10         $\lambda \leftarrow$ random action previously taken in $s$
11         $r, s' \leftarrow Model(s, \lambda)$
12         $Q(s, \lambda) \leftarrow Q(s, \lambda) + \zeta\,[r + \gamma \ \text{argmax}_\lambda[Q(s', \lambda)] - Q(s, \lambda)]$
```

The Dyna-Q [34] RL algorithm is a model-free RL paradigm that consists of a combination of the model-based and Q-learning paradigms. The Dyna-Q Algorithm is summarized in Algorithm 2. Unlike Q-learning, Dyna-Q aims to speed up the learning process by simulating the system interaction with the environment. At run-time, Dyna-Q observes the application state and selects an adaptation action using the estimates of $Q(s, \lambda)$, as Q-learning does. At the end of the time step i, DynaQ exploits a sampled model of the system, $Model(S, \Lambda)$, where $Model(S, \Lambda)$ refers to the MDP model of the environment with the updated reward values in each iteration of the Q-Learning algorithm, to simulate the interaction between the application and the environment (lines 8 - 12). Dyna-Q updates $Model(S, \Lambda)$ at runtime, by storing the next state s' and reward r for the explored state-action pair (s, λ) at line 7. Dyna-Q updates the Q-function akin to Q-learning using Equation 2.1 and resorting to the state-action pairs previously observed and proceeds to simulate updation using previous interactions.

```
mdp

const double P_SLEEP = 0.15;
const double P_WAKE = 0.85;

const double P_FAIL = 0.3;

...

module device

state: [0..1];
//0 implies wake state
//1 implies sleep state

pkt : [0..3];
//0 implies no packet to send
//1 implies packet to send
//2 implies failed to send packet
//3 implies successfully sent packet


    [] state = 0 & pkt = 0 -> P_SLEEP : (state'=1) + (P_WAKE) : (state'=0);

    [send] state=0 & pkt = 1 -> (P_FAIL) : (pkt' = 2) + (1-P_FAIL) : (pkt' = 3);

    [wait] state=0 & pkt = 1 -> true;

endmodule

...
```

Listing 2.1: Module of a Mobile Device

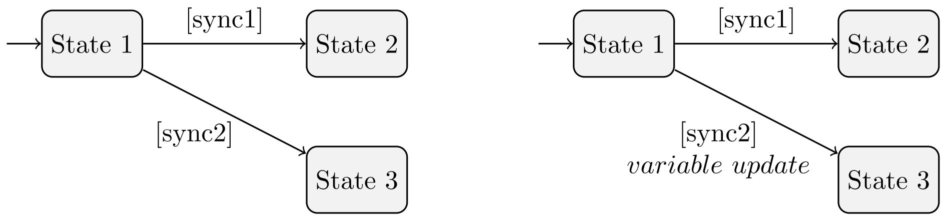
2.5 PRISM Model Checker

PRISM [93] is a Probabilistic Model Checker offering a high level programming language for describing probabilistic systems. For description of probabilistic models, PRISM employs an input language developed from Reactive Modules, a language for process-algebraic expression [93]. Models are defined in PRISM as the parallel composition of a number of modules, each of which has a set of instructions that define transitions. Each module comprises variables describing the possible states of the module. Modules comprise commands (or actions) describing transitions between the various states. Each command consists of an optional action name conditionally executed on arithmetic expressions of variables and constants. The result of execution of each command is a probability distribution over successor states of the variables.

Additionally, in PRISM, each state of an MDP can be associated with a reward value. Thus, a PRISM reward definition for an MDP M is a mapping from each state of the MDP to a real number formally defined as: $\mathcal{R} : \Psi \rightarrow \mathbb{R}$. More about PRISM is detailed in [93]. We utilize PRISM as the probabilistic model checker in this thesis.

Example 2.5.1. Listing 2.1 depicts a sample PRISM specification of a mobile device modeled as an MDP. It comprises three constants, the first two constants define the probability of the device going into sleep mode and remaining awake respectively while the third constant defines the probability of successfully sending a packet over the network. In this example, we define a single module for the device. The module comprises two variables, `state` and `pkt` representing the sleep/wake status of the device and the status of the packet to be sent respectively. The module comprises three actions. The first action characterizes the probability distribution of going into the sleep mode when there is no packet to send (`pkt = 0`). The next two actions characterize non-deterministic choices of whether to send a packet immediately or to wait before sending the packet. ■

PRISM offers labelled synchronization specifications between modules defined with respect to labels associated with each transition.



(a) MDP M_1 (b) MDP M_2

FIGURE 2.15: PRISM Module Synchronization

Example 2.5.2. Consider the two MDPs in Figures 2.15a and 2.15b. Each MDP can be represented by a PRISM module. Each MDP comprises three states. In MDP M_1, whenever the transition from State 1 to State 2 is executed identified by the label [sync1], the transition in M_2 identified by the same label [sync1] is also executed. Similarly, consider the transition in M_1 from State 1 to State 3 labelled by [sync2]. Whenever the transition is executed, the corresponding transition in M_2 labelled by [sync2] is also executed. Additionally, note that in M_2, the [sync2] transition is further augmented with *variable update*. Such statements denote updation of the specified PRISM variable on execution of the transition. ■

In the following section, we discuss some of the datasets and benchmark applications utilized in our contributory chapters.

2.6 Dataset and Benchmark Desciption

We provide below a brief description of each of the datasets and the application benchmark suite utilized in our work.

2.6.1 WS-DREAM dataset

The WS-DREAM [96] dataset comprises real-world QoS measurements, including both response time and throughput values. We utilize this dataset to generate representative latency values for multiple instances of microservice containers in accordance with the QoS values.

2.6.2 San Francisco Taxi Dataset and Tower Locations

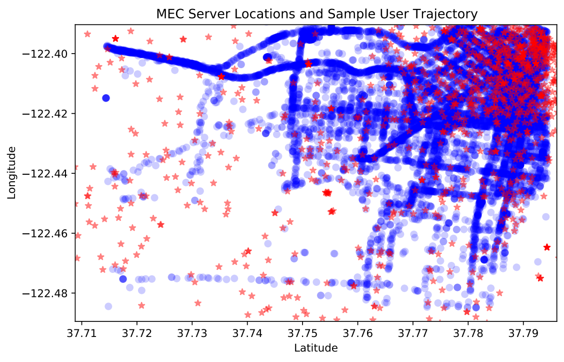


FIGURE 2.16: San Francisco Taxi and Tower Locations

We utilize real-world mobility traces of taxis in San Francisco [97], available publicly, collected over different time points during the day where different numbers of taxis operate at different times of the day. For edge site locations, we use the 'Existing Commercial Wireless Telecommunication Services Facilities in the San Francisco' dataset [98], which is also available publicly, as MEC server locations. The coverage area of each MEC server is randomly generated (while ensuring full coverage of the city area under study) as in [6, 7]. However, this dataset only comprises locations of towers confined mostly to a large segment of the San Francisco area. The San Francisco taxi dataset, however, comprises taxi trajectories distributed over the entire city of San Francisco. Hence, we consider only a portion of the main San Francisco city area with 81 facility locations as edge sites within the dataset since only a portion of the dataset comprises information about Wireless Telecommunication Facilities located in close vicinity of each other that can be utilized to represent adjacent edge sites. From the dataset, we extract for each taxi, the coordinates confined to the region of MEC server locations. The coverage area of each edge site is randomly generated (while ensuring full coverage of the city area under study) as in [6, 7]. In Figure 2.16, we use red stars to depict edge site locations and blue circles to show the trajectory of a sample taxi for the area of the city utilized in our contributory chapters.

2.6.3 DeathStarBench Benchmark Suite

The DeathStarBench benchmark suite [48] presents microservice based applications from domains such as social media and movie streaming. This has been used in a range of studies such as hardware and networking implications of microservices and performance debugging of cloud microservices [99] and predicting QoS values in cloud based services [88]. Application services of the DeathStarBench benchmark suite are deployed in containers. Each application is deployed as a web server which caters to incoming service requests. The Social Network application includes a workload generator *wrk* where the workload can be controlled by means of different parameters. We utilize two such parameters, the workload duration and the number of service requests per second since these are representative of low latency MEC environments [6]. The Media Microservices application caters to users searching and browsing for information about movies, including their plot, photos, videos, cast, and review information. The Media Microservices application also includes a workload generator similar to the Social Network application.

2.6.4 YOLO Application

You Only Look Once (YOLO) is a state-of-the-art, real-time object detection system [100]. YOLO applies a single neural network to an entire image where the network divides the image into regions and predicts the bounding boxes of objects for predicting objects in images. We use the standard pre-trained weights for the object recognition task. YOLO utilizes both CPU and GPU for object detection [100].

2.6.5 PlanetLab Dataset

We use the PlanetLab dataset [101], used in prior work in Fault-Tolerance [15], to simulate server failures. The datasets comprise availability information of various servers each identified by a unique ID. Each edge server in our setup is randomly assigned one of the unique IDs available in the datasets and assigned the corresponding probability of failure from the dataset. Additionally, the PlanetLab dataset also comprises latency measurements between the different machines on the PlanetLab network thereby serving as an indicator of inter-machine communication latency.

2.6.6 Telefonica Mobile Phone Usage Dataset

The Telefonica Dataset [102] contains detailed logs of mobile phone usage of 342 people over the course of 4 weeks. The Telefonica dataset provides logs of invocation of the Google Maps service along with the GPS coordinates of the device at that timepoint. Service requests of the Google Maps application are then used for experimental purposes.

To the best of our knowledge, there are no real-world MEC implementation workload traces that are publicly available and sufficiently representative of the problems we consider in our contributory chapters. Therefore, depending on the problem context, we generate synthetic workloads using some the above real-world datasets.

In contrast to state-of-the-art MEC literature, this thesis makes some unique contributions. A central objective of this thesis is to design and verify MEC policies. In the subsequent chapters, we present in detail our design and verification approaches. We design a microservice based user allocation policy considering the correlation between microservice inter-communication and microservice deployments on MEC servers. We further demonstrate the benefits of proactively prefetching microservices. Additionally, we design an auto-scaling policy that ensures adherence to latency requirements. In the event of failures, we synthesize recovery strategies taking into account the possibility of multiple failures. Finally, we develop a trace-driven verification framework to evaluate service allocation policies. We believe that this work presents some unique approaches to MEC policy design and verification that distinguishes our contributions from state of the art MEC literature.

Chapter 3

Service Allocation for Microservice based Applications

3.1 Introduction

State of the art service allocation policies in MEC literature are geared towards monolithic applications [6, 7, 8]. To the best of our knowledge, allocation policies for microservice based applications have been less relatively explored. Recently, authors in [9] proposed an allocation policy for microservice based applications. However, they did not consider the dependencies arising from a DAG workflow structure and the implications in MEC allocations. In order to enhance fault tolerance and load balancing, identical container instances of a microservice are deployed on several MEC servers. Each such container can have different access and task execution latencies depending on the edge site where the container is deployed as well as the current load associated with the server. In this chapter, we propose an allocation policy for microservice based applications considering the correlations arising from the geo-spatial deployment of microservices on MEC servers with respect to a given microservice workflow structure. We consider two types of correlations among service containers in this work. The first type of correlation arises between

This work is based on:

- Kaustabha Ray, Ansuman Banerjee and Swarup Kumar Mohalik, "Service Selection with Package Bundles and Compatibility Constraints", In IEEE Transactions on Services Computing, 2021, doi: 10.1109/TSC.2021.3075030.

microservice containers deployed on the same server. We term these as microservice bundles that enjoy negligible data transfer latencies between containers. Additionally, we also consider compatibility constraints between containers arising from the diverse platform and library configurations on MEC servers as the second type of correlation. Our objective here is to work on the minimum latency microservice allocation problem in the presence of compatibility constraints and microservice bundles induced by multiple containers hosted for each microservice. For a given microservice based application workflow, our problem deals with selecting a set of containers such that each microservice of a workflow can be realized by at least one deployed container, honoring compatibility constraints, and the resulting latency incurred in executing the workflow is minimized. For simplicity of illustration and ease of explanation in this chapter, we focus on applications with workflows defined by a linear sequence of microservices. We consider scenarios with an existing service placement configuration and known locations of users similar to the approaches utilized in [7, 8].

In this work, we take an abstract view of the MEC context and provide a novel hyper graph model, which lends itself to an efficient allocation modality. The key insight behind our approach is an innovative solution strategy that builds on a unique combination of Integer Linear Programming (ILP) [103] and Abstraction Refinement (AR) [104]. In particular, we explore how latency correlations and compatibility constraints can influence microservice allocation. We present a formal proof of hardness for the user allocation problem in our model, and formulate an Integer Linear Programming (ILP) - based approach for solving the same. Additionally, we use Abstraction Refinement as an enabler to expedite the ILP by systematically creating abstract microservices as part of the abstraction process on top of which we use the baseline ILP to generate optimal solutions. Further, we iteratively refine any artifacts which may have been introduced as a result of the abstraction process. We demonstrate that such an approach eventually terminates generating an optimal solution.

To validate the proposed method, we experiment on synthetically generated representative latency values from real-world datasets, scaling the number of microservices, microservice deployments, microservice bundles and compatibility constraints. Experimental results show that our solution performs well with large workflows, even in large scale scenarios.

The rest of this chapter is organized as follows. Section 3.2 presents an illustrative example. Section 3.3 outlines the problem. Section 3.4 presents a baseline solution using ILP to solve the problem. Section 3.5 presents our ILP + Abstraction Refinement combination approach. Section 3.6 details our results while the next section concludes the chapter.

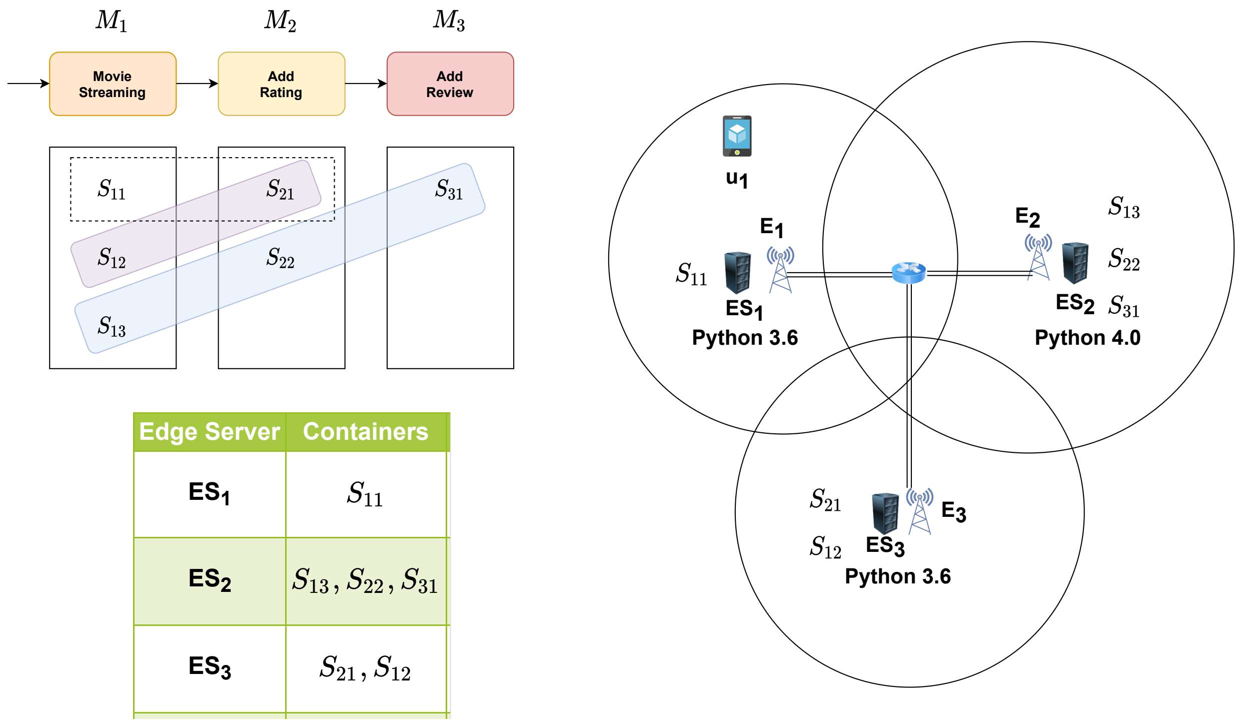
3.2 A Motivating Example

We first describe the access latencies associated with the container based microservice provisioning model discussed in Chapter 2. Microservice DAG workflows comprise vertices denoting individual microservices and edges representing microservice inter-communication. Edges depict the data-flow between consecutive microservices i.e., an edge from a microservice M_i to microservice M_j denotes that the output of M_i is utilized as the input of M_j. Each microservice of the workflow is deployed as a set of *multiple functionally equivalent containers* on different MEC servers to enhance load balancing and fault tolerance. As users invoke these applications, the objective of a service allocation policy is to determine for a particular user for each microservice in his/her workflow, the server container instance where the request will be provisioned. Once the microservice-container-server binding has been determined, a new task is created at the server pertaining to each microservice in the workflow to be provisioned.

Accessing microservice containers deployed at MEC servers is associated with an *access latency* depending on the geographical area where the MEC server is located, a *computation latency* incurred in executing the service request on the container with the user specific inputs, and a *data transfer latency* incurred due to transferring of data between the microservices of the workflow. The latencies incurred due to transferring the output data of M_i to M_j could vary depending on the MEC servers on which the containers corresponding to M_i and M_j are deployed. For example, if M_i and M_j are deployed on the same MEC server, no *data transfer latency* is incurred. We refer to such co-located microservice containers as *microservice-bundles*.

Additionally, there often exists *compatibility constraints* between the microservice container deployments that need to be honored. Compatibility constraints arise from the diverse platform configurations on MEC servers. An an example, MEC servers can run Windows or Linux operating systems, however, intercommunicating microservices often require the same platform and technology stack (Application Programming Interface (API), dynamically linked libraries and development packages) to operate correctly. In our service allocation method, we also consider such compatibility constraints arising from microservice container deployments.

Consider the scenario in Figure 3.1. There are three edge sites E_1, E_2, and E_3 where each edge site is powered by a single edge server. We consider a linear workflow comprising three microservices M_1, M_2, and M_3, as shown in the figure. User u_1 can access the containers deployed on the MEC servers associated with E_1 with low latency since it is located in the coverage area of E_1. User u_1 can also access the containers deployed on MEC servers at other edge sites via the backbone network, however, with an additional backbone network access latency. A set of containers corresponding to each microservice is deployed on the servers as



Edge Server	Containers
ES$_1$	S_{11}
ES$_2$	S_{13}, S_{22}, S_{31}
ES$_3$	S_{21}, S_{12}

Figure 3.1: Example Microservice Deployment Scenario

listed in Table 3.1. Table 3.1 also lists the *total latencies* defined as the sum of the *access latency*, the *computation latency* and the *data transfer latency* (applicable for all microservices $\neq M_1$) associated with u_1 for availing the services of the respective containers. In this work, we assume that an apriori latency estimate is available for each user for the above latencies. In addition to individual containers, Table 3.1 also comprises two *microservice-bundles* induced by the deployment configuration of the containers. Since S_{12} and S_{21} are both deployed on server ES_3, *data transfer latency* is not incurred. Similarly, S_{13}, S_{22} and S_{21} form a bundle. Such bundles are depicted by the purple and blue highlighted rectangles in Figure 3.1. Note that in Table 3.1, we represent singleton containers as bundles as well. We utilize such a representation throughout this chapter. We identify each bundle as q_i. Additionally, there exists a compatibility constraint between S_{21} and S_{31} as shown by the dotted rectangle in Figure 3.1. Such a compatibility constraint stems from both containers S_{21} and S_{31} corresponding to AddRating and AddReview having identical (e.g. Python 3.6) library dependencies. The compatibility constraint occurs for the AddReview microservice utilizing the output of the AddRating microservice to update the database with the ratings and reviews together, utilizing the same database format. Other containers are mutually compatible with each other.

The minimum latency microservice allocation problem is to identify a set of bundles such that at least one service container S_{ij} is chosen for each microservice M_i in the user workflow, compatibility constraints are honored, and the overall latency is minimized. The latency for a

Containers	**Bundle Identifier**	**Latency (in ms)**
S_{11}	q_0	50
S_{12}	q_1	50
S_{13}	q_2	60
S_{21}	q_3	50
S_{22}	q_4	60
S_{31}	q_5	60
S_{12}, S_{21}	q_6	95
S_{13}, S_{22}, S_{31}	q_7	150

TABLE 3.1: Container Deployment on MEC Servers

set of bundles is defined as the sum of the individual latencies of each bundle in the set. In this example, the minimum latency solution for u_1 is 150ms which pertains to the bundle q_7. In terms of service containers, these map to S_{13}, S_{22}, and S_{31}. Note that these containers are all mutually compatible with each other.

Evidently, the size of the solution space for the allocation problem is exponential in the number of bundles, and some of the solutions are invalid as well. For a particular microservice, more than one container can serve the required purpose as well. The naive approach of examining all the solutions, removing the invalid ones, and extracting the minimum is a compute intensive task. Our contribution here is a novel method to systematically navigate the solution space of the allocation problem.

3.3 Problem Formulation

Consider a workflow W comprising a set $M = \{M_1, M_2, \ldots\}$ of microservices and a service container repository S, where each $s \in S$ has a core functionality $cf(s)$. We have a set of bundles $Q = \{q_1, \ldots, q_r\}$, where each q_i has an associated latency denoted as $latency(q_i)$. Containers in a bundle have disjoint core functionalities i.e., for all $s_i, s_j \in q_k, i \neq j$, $cf(s_i) \neq cf(s_j)$. We represent individual container latencies as singleton bundles.

Definition 3.1 [**Container choices for M_i microservice:**]
Given a user specified workflow W as a linear acyclic workflow of microservices M where each microservice M_i is annotated with a core functionality CF_i, the container choices available for the microservice M_i is defined as $S_i = \{s \mid cf(s) = CF_i\}$, with the individual constituents of the set denoted as $S_{i1}, S_{i2}, S_{i3}, ...S_{in_i}$, where, $n_i = |S_i|$.

Notation	Description
S	service repository
s	any service in service repository S
s_i	i-th service in service repository S
$cf(s_i)$	core functionality of i-th service
W	workflow comprising several tasks
T_i	i-th task of workflow W
S_i	set of candidate services for Task T_i
S_{ij}	j-th candidate service of task T_i
n_i	number of candidate services of task T_i
Q	set of service bundles
q	any service bundle in repository S
q_k	k-th service bundle in repository S
$latency(h_i)$	latency of i-th service bundle in repository S
r	total number of bundles in repository S
P	set of compatibility constraints
p_k	k-th compatibility constraint
Z	set of elements used in the set cover problem
C	subset of Z used in the set cover problem
h_k	0/1 indicator variable for q_k
H_i	set of indicator variables for bundles in T_i
H_i^k	set of indicator variables for antecedents of p_k
$\tilde{S}$	task level partition of S
σ_{ij}	j-th partition of i-th task
$[s]$	partition subset of service s
$[q]$	partition subset of bundle q
$\hat{S}$	abstract service repository
$\hat{Q}$	abstract set of service bundles
$\hat{W}$	abstract workflow comprising abstract tasks
$\hat{P}$	abstract set of compatibility constraints
$\hat{T_i}$	i-th abstract task
$\widehat{T_{S_{ij}}}$	abstract task corresponding to S_{ij}
$\hat{T_i'}$	i-th refined compatible abstract task
$\hat{T_i''}$	i-th refined non-compatible abstract task
$\hat{q_k}$	k-th abstract bundle
$\hat{q_k'}$	k-th refined abstract bundle
ρ_k	value of k-th service parameter
ω_k	weight assigned to k-th service parameter

Table 3.2: Table of Notations

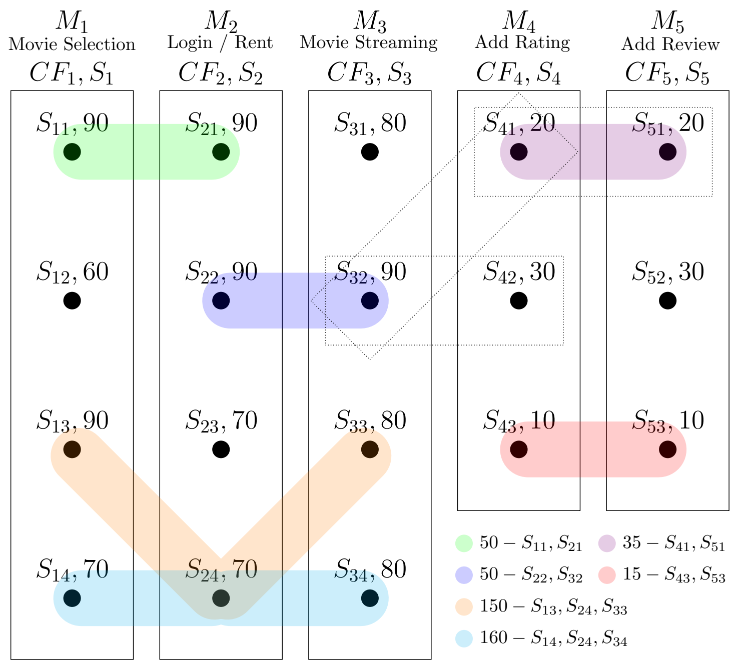


FIGURE 3.2: Service Provider Repository

Definition 3.2 [**Compatibility Constraint:**]
A compatibility constraint between microservices M_i and M_{i+1} denoted by p_i is of the form $S_{ij} \rightarrow \{S_{(i+1)m}\ ,\ S_{(i+1)n}\ ,\ \ldots\}$. This mandates that if S_{ij} is utilized for M_i, at least one among $S_{(i+1)m}, S_{(i+1)n} \ldots$ has to be selected for M_{i+1}.

We are given a set $P = \{p_1, p_2, \ldots p_k\}$ of compatibility constraints across consecutive containers. Further, for a container S_{ij} in M_i that does not have any explicit compatibility constraint with any $S_{(i+1)j}$ in M_{i+1}, we assume an implicit compatibility $S_{ij} \rightarrow \bigcup_{m=1}^{|S_{i+1}|} S_{(i+1)m}$, which denotes S_{ij} is compatibile with all containers of M_{i+1}. We do not further explicitly mention these implicit constraints in the following discussion.

Intuitively, we can visualize the problem space as a multi-partite hyper-graph [105], as in Figure 3.2, with latencies on hyper-edges and constraints on edge-pairs, where each hyper-edge corresponds to a bundle. The container hypergraph is formally defined as follows:

Definition 3.3 [**Container Hypergraph:**]
A container hypergraph $\mathcal{H}$ is a pair $(\mathcal{H}_v, \mathcal{H}_e)$ where $\mathcal{H}_v (= S)$ represents the set of all containers for the different microservices and $\mathcal{H}_e \subset 2^{\mathcal{H}_v}$ where $2^{\mathcal{H}_v}$ represents the powerset of $\mathcal{H}_v$, i.e., all possible microservice bundles.

We do not consider the empty set within $2^{(\mathcal{H}_v)}$. We utilize the example container repository in Figure 3.2 to explain our solution methodology. Table 3.3 lists the bundles and the respective

Bundle Name	containers	Total Latency (ms)
q_1	S_{11}	90
q_2	S_{12}	60
q_3	S_{13}	90
q_4	S_{14}	70
q_5	S_{21}	90
q_6	S_{22}	90
q_7	S_{23}	70
q_8	S_{24}	70
q_9	S_{31}	80
q_{10}	S_{32}	90
q_{11}	S_{33}	80
q_{12}	S_{34}	80
q_{13}	S_{41}	20
q_{14}	S_{42}	30
q_{15}	S_{43}	10
q_{16}	S_{51}	20
q_{17}	S_{52}	30
q_{18}	S_{53}	10
q_{19}	$S_{11}S_{21}$	50
q_{20}	$S_{22}S_{32}$	50
q_{21}	$S_{41}S_{51}$	35
q_{22}	$S_{43}S_{53}$	15
q_{23}	$S_{13}S_{24}S_{33}$	150
q_{24}	$S_{14}S_{24}S_{34}$	160

Table 3.3: Package Bundles and their Latencies

latencies for the container repository in Figure 3.2. Latencies are specified on singleton bundles as well, as in case of rows 2-19 of Table 3.3.

We now formally define the problem of Minimum Latency Container Bundle Allocation with Compatibility Constraints (MLCBACC).

Definition 3.4 [**MLCBACC:**]
MLCBACC problem takes in a tuple $\langle S, Q, W, P\rangle$ with the objective of selecting a subset $Q' \subseteq Q$ such that

- *Q' is a* realizing *solution i.e. for each $M_i \in M$, there is at least one $S_{ij} \in \bigcup Q'$ with matching core functionality cf.*
- *Q' meets all compatibility constraints, i.e., for every constraint $p_k \in P$ of the form $S_{ij} \rightarrow \{S_{(i+1)m}\ ,\ S_{(i+1)n}\ ,\ \ldots\}$, if $S_{ij} \in \bigcup\ Q'$, then at least one among the set $S_{(i+1)m}$, $S_{(i+1)n}$, $\ldots \in \bigcup Q'$.*
- *Total latency of Q' is minimum over all realizing constraint satisfying subsets of Q.*

Observation 3.3.1. We first note a simple fact about the latencies of the bundles in Q: if there is a bundle $q \in Q$ that can be realized by a set of smaller bundles, say $q_1, \ldots, q_k$ i.e. $q = \bigcup_{i=1}^{k} q_i$, then it must be the case that $latency(q) < \sum_{i=1}^{k} latency(q_i)$; or else, any solution can identify containers from the constituent bundles, making q redundant. ⧫

We now prove the NP-Completeness of MLCBACC.

Lemma 3.1. *MLCBACC is NP-complete.*

Proof: Given a workflow W and an arbitrary subset Q' of Q, we can validate in polynomial time if Q' realizes the entire workflow honoring all compatibility constraints, and also check that the latency is less than a given bound. Thus the decision version of MLCBACC is in NP.

Consider a MLCBACC problem $\langle S, Q, W, P\rangle$. The lower bound is proved by a reduction from the set covering problem [106] to the decision version of MCSBSCC by setting P to be empty in $\langle S, Q, W, P\rangle$. We are given a collection $C = \{c_1, \ldots, c_n\}$ of n subsets of a given set of elements $Z = \{z_1, \ldots, z_p\}$, and are required to determine whether for a given integer $k < n$, there exists k subsets in C whose union gives Z. Obviously, each element of Z is contained in one or more subset of C, otherwise a solution does not exist. Given the set $Z = \{z_1, \ldots, z_p\}$, we define a workflow comprising a set of microservices $M = \{M_1, \ldots, M_p\}$. Each microservice has only a single container S_{i1}. Intuitively, we associate microservice M_i with the element z_i of Z. Also we associate each subset c_i, for $1 \leq i \leq n-1$, with a distinct combination bundle, i.e. hyper-edge q_i. We associate a latency of 1 with each hyper-edge thus created. From the construction, it now follows that a set of k hyper-edges is the minimum latency bundle selection *iff* the corresponding subsets in C cover all the elements of Z. We thus conclude from this reduction that MLCBACC is NP-Complete in general. ⧫

In the light of the lower bound proof, it is evident that any solution to MLCBACC has high complexity in the worst case. However, for real-life problem instances, one may still find solutions that perform fairly well. We present two solution strategies in the following and corroborate the efficiency through experimental results on publicly available service benchmarks.

3.4 A Baseline Solution Methodology

We first present an ILP approach for solving the problem. We propose a baseline ILP since state-of-the-art algorithms for bundled service selection deal with only bundles ignoring compatibility constraints [107]. We associate a binary (0 / 1) indicator variable h_i for each $q_i \in Q$ to indicate if the bundle q_i is present or not in the minimum latency bundle. Thus, we have as many indicator variables as there are elements in Q, including the singleton bundles. We formally define the indicator variable h_i as follows:

$$h_i = \begin{cases} 1, & \text{if } q_i \in \hat{Q} \\ 0, & \text{otherwise} \end{cases}$$

Thus, the indicator variable h_i has a value of 1 for those bundles included in the solution MLCBACC, $\hat{Q}$ and a value of 0 for those bundles not included in the solution. In addition, we have the constraint that at least one container is included for each M_i. As earlier, let S_i = $\{S_{i1}, S_{i2}, S_{i3}, ...S_{in_i}\}$ denote the set of containers for microservice M_i. For each S_{ij}, we can associate it to one or more $q_k \in Q$. For each M_i, let H_i denote the set of indicator variables h_p that correspond to bundles which include some member S_{ij} of S_i for microservice M_i. Evidently, this information can be derived from the bundle information. Our constraint is to ensure at least one member of H_i for each M_i is included in our minimum latency solution.

Further, we need to enforce the compatibility constraints between service containers. Consider p_k: $S_{ij} \rightarrow \{S_{(i+1)m}, S_{(i+1)n}, \ldots\}$ as earlier. Let $\hat{H}_a^k$ denote the set of indicator variables that correspond to bundles which include the *antecedent* of p_k, i.e. S_{ij}, where k denotes the cardinality of the set $\{S_{(i+1)m}, S_{(i+1)n}, \ldots\}$. Let $\hat{H}_{(i+1)m}$ denote the set of indicator variables that correspond to bundles which include $S_{(i+1)m}$, let $\hat{H}_{(i+1)n}$ denote the set of indicator variables that correspond to bundles which include $S_{(i+1)n}$ and so forth. Let $\hat{H}_c^k$ be the union of the indicator variable sets for the *consequents* of p_k, i.e. $S_{(i+1)m}$, $S_{(i+1)n}$, ... If S_{ij} is included, at least one among $S_{(i+1)m}$, $S_{(i+1)n}$ ... has to be taken. Thus, if one of the indicator variables in $\hat{H}_a^k$ is set to 1, at least one in $\hat{H}_c^k$ will also have to be set to 1. Let r denote the total number of bundles in the repository including singleton bundles. Combining the above, we have the ILP:

$$Minimize : \sum_{i=1}^{r} h_i * latency(h_i)$$

$$Subject\ to\ : \sum_{h_p \in H_i} h_p \geq 1\ ,\ \forall\ H_i \qquad (1)$$

Additionally, for each compatibility constraint $p_k \in P$, following the discussion as above, for each h_x in $\hat{H}_a^k$, we have:

$$h_x \leq \sum_{h_q \in H_c^k} h_q$$

The above can be combined as follows:

$$\sum_{h_x \in H_a^k} h_x \leq \left(|H_a^k| \times \sum_{h_q \in H_c^k} h_q \right) \ , \ \forall \ p_k \in P \quad (2)$$

Constraint 1 ensures at least one container is included for each microservice M_i. Constraint 2 generates the compatibility constraints. $|H_a^k|$ denotes the number of elements in the set H_a^k. The above set of equations are then fed into an ILP solver. The solution generated by the ILP solver represents the solution to MLCBACC. As the number of microservices and containers grows, the number of possible bundle offerings and compatibility constraints grows as well and we have a large number of indicator variables. The solution space that the ILP solver has to handle grows as well, and more often than not, the ILP solver faces hurdles to scale to larger problem instances. However, if the ILP solver is able to generate a solution, it is always guaranteed to produce the optimal one. In this paper, we present another optimal solution approach on top of ILP to scale it further.

3.5 Using Abstraction Refinement on ILP

In this discussion, we present an abstraction refinement based solution strategy that applies the ILP on a smaller problem space, and generates the same optimal solution. Our procedure comprises 3 main steps, as below.

- *Abstraction:* For each microservice M_i, we create an abstract microservice $\hat{M}_i$. We create a set of abstract bundles and assign their latencies. Further, we introduce compatibility constraints between the abstract microservices.
- *Solution Generation and Validation:* We use the baseline ILP as described in Section 3.4 on the set of abstract bundles to generate the optimal selection solution. In the following discussion, we refer to such a solution as an *abstract solution.* We then check if the abstract solution mapped back to the original repository meets the compatibility constraints. If it does, i.e, we find a corresponding solution in the original repository, then a valid solution (which is optimal as well as we prove later) has been obtained from the abstraction and thus we terminate. We term the corresponding solution in the original repository as a *concrete solution.* However, if it violates any constraint, we remove the spurious solution by refining the abstraction.

- *Refinement:* Refinement is carried out by splitting the abstract microservices in case of spurious solutions.

We first formalize the notion of abstraction and then describe the detailed methodology of our ILP + Abstraction Refinement Framework.

Definition 3.5 [**Concrete Bundle:**]
Given an MLCBACC problem L, we refer to each $q \in Q$ as a concrete bundle *of Q.*

Example 3.5.1. Each bundle summarized in Table 3.3 is a concrete bundle for the container repository in Figure 3.2. ■

Definition 3.6 [**Microservice Level Partition:**]
Let S_i denote the set of containers for microservice M_i. A microservice level partition of S, denoted $\tilde{S}$, is one in which each S_i is partitioned into subsets $\sigma_{i1}, \ldots \sigma_{ik}$, collectively referred to as σ_i. For any element $s \in \sigma_i$ corresponding to all partitions of S_i, $[s]$ denotes the subset σ_{ij} such that $s \in \sigma_{ij}$, i.e, $[s]$ denotes the subset to which s belongs.

Example 3.5.2. Table 3.4 summarizes a microservice level partition for the repository in Figure 3.2. Each microservice comprises two partitions, one partition comprising the minimum latency container, while the other partition comprises the remaining containers. Thus, for the container S_{12}, $[S_{12}] = \sigma_{11}$ and for S_{13}, $[S_{13}] = \sigma_{12}$. ■

Definition 3.7 [**Abstract MLCBACC Problem:**]
Given an MLCBACC problem $L = \langle S, Q, W, P \rangle$, and a microservice level partition $\tilde{S}$, we define the abstract MLCBACC problem $\hat{L} = \langle \hat{S}, \hat{Q}, W, \hat{P} \rangle$:

- $\hat{S} = \tilde{S}$, $\hat{Q} = \{[q] \mid q \in Q\}$ *with* $[q] = \{[s] \mid s \in q\}$.
- *latency*$([q])$ = *min*$\{$*latency*$(q') \mid [q'] = [q]\}$, *where each* $[q] \in \hat{Q}$ *is called an abstract bundle. We retain the minimum latency bundle in* $\hat{L}$.
- *for each M_i, we define an abstract microservice $\hat{M}_i = \bigcup [s], \ \forall [s] \in M_i$.*

Microservice	**Partition and Constituents**
M_1	$\sigma_{11} = \{S_{12}\}$ $\sigma_{12} = \{S_{11}, S_{13}, S_{14}\}$
M_2	$\sigma_{21} = \{S_{24}\}$ $\sigma_{22} = \{S_{21}, S_{22}, S_{23}\}$
M_3	$\sigma_{31} = \{S_{31}\}$ $\sigma_{32} = \{S_{32}, S_{33}, S_{34}\}$
M_4	$\sigma_{41} = \{S_{43}\}$ $\sigma_{42} = \{S_{41}, S_{42}\}$
M_5	$\sigma_{51} = \{S_{53}\}$ $\sigma_{52} = \{S_{51}, S_{52}\}$

Table 3.4: Task Level Partition of the Repository shown in Figure 3.2

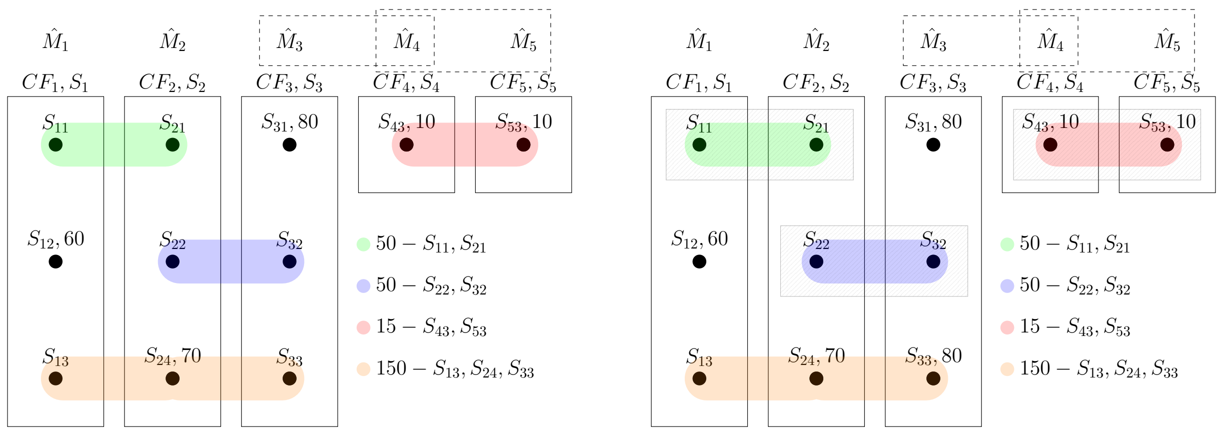
Definition 3.8 [**Abstract Compatibility Constraint**:]
For every constraint $S_{ij} \rightarrow \{S_{(i+1)1}, \ldots, S_{(i+1)k}\}$ *in* P*, consider the abstract microservice corresponding to* $[S_{ij}]$ *as* $\hat{M}_{S_{ij}}$ *and the abstract microservice corresponding to the* $(i+1)^{th}$ *microservice as* $\hat{M}_{i+1}$*.* $\hat{M}_{S_{ij}} \implies \hat{M_{i+1}}$ *is an abstract compatibility constraint in* $\hat{P}$*.*

Note that in the above definition, we need to consider only $\hat{M_{i+1}}$, since we consider compatibility constraints between adjacent microservices only and $S_{(i+1)1}$, $S_{(i+1)2} \ldots S_{(i+1)k}$ are all containers of microservice M_{i+1}. In the following subsection, we explain the detailed construction of the Abstract MLCBACC problem and the subsequent refinements as and when necessary.

3.5.1 Initial Abstraction Generation

We create a microservice level partition for each M_i, such that one partition comprises the container with minimum latency and the other partition comprises the remaining containers. For each microservice M_i, we create an abstract microservice $\hat{M_i}$. We include only the partition comprising the minimum latency container in the abstract microservice since other containers can be part of a realizing MLCBACC solution only if it is offered as a bundle as noted in Observation 3.3.1. Thus, such containers are represented as abstract bundles. We create the set of abstract bundles $\hat{Q}$ on the abstract microservice set for each $\hat{M_i}$. Further, for each $\hat{M_i}$, we create a singleton bundle $\hat{q}_i$ and assign it the minimum singleton bundle latency among elements in S_i. Lines 3-11 of Algorithm 3, which summarize the initial abstraction generation approach, perform this assignment. We examine the other concrete bundles to create further abstract bundles. For each concrete bundle q_i, we have the containers included, and can therefore, extract the microservices involved in them. For each 2-element bundle q_p, we proceed as follows. Let M_i, M_j be the microservices involved in q_p. We create an abstract bundle $\hat{q}_{ij}$, involving the microservices $\hat{M_i}$ and $\hat{M_j}$. We assign the latency of $\hat{q}_{ij}$ as the minimum among all 2-element bundles that involve M_i, M_j. We repeat the same with all 2-element concrete bundles and create the corresponding abstract bundles involving 2 abstract microservices. We now examine each 3-element concrete bundle, extract the set of microservices M_i, M_j, M_k involved, create an abstract bundle $\hat{q}_{ijk}$, look for other bundles involving the same triplet, and assign $\hat{q}_{ijk}$ the minimum among the 3-element bundle latencies involving M_i, M_j, M_k. We process all concrete bundles similarly. Lines 12-21 generate these bundles, and these are stored in a hash table. The hash table is a mapping from an abstract bundle to the latency associated with that bundle. Thus, initially, the hash table comprises an entry corresponding to each bundle generated in the abstraction process and the latency assigned to the bundle.

Let us consider two consecutive microservices M_i and M_{i+1}. If any S_{ij} in M_i has at least one compatibility constraint with containers in M_{i+1}, we introduce a compatibility constraint between $\hat{M}_i$ and $\hat{M}_{i+1}$. Lines 22-25 generate all such constraints. Concrete compatibility constraints between containers are thus abstracted out to create constraints between the abstract microservices. This gives us the complete abstract microservice set with the abstract bundles and latencies, and we proceed to the next step.



(a) Initial Abstraction (b) Initial Abstraction Solution

FIGURE 3.3: Initial Abstraction

Example 3.5.3. For the repository in Figure 3.2, we create 5 abstract microservices $\hat{M}_1$, $\hat{M}_2$, $\hat{M}_3$, $\hat{M}_4$ and $\hat{M}_5$. We assign $\hat{q}_1$ the latency of S_{12} since this is the minimum latency singleton bundle among all singleton bundles for containers of M_1. The process is repeated for all other microservices. We now consider the other bundles. There is only one bundle involving containers from microservices M_1 and M_2, the bundle between S_{11} and S_{21} whose latency is 50ms. Thus we add a bundle $\hat{q}_{12}$ involving $(\hat{M}_1, \hat{M}_2)$. As another example, consider the three element bundles involving containers for M_1, M_2 and M_3. There are two bundles involving these, namely, $\{S_{13}, S_{24}, S_{33}\}$ with latency 150ms and $\{S_{14}, S_{24}, S_{34}\}$ whose latency is 160ms. We take the minimum latency bundle among them $\{S_{13}, S_{24}, S_{34}\}$ and assign this latency to the abstract bundle $\hat{q}_{123}$. Further, there are two explicit compatibility constraints involving M_3 and M_4: one between S_{32} and S_{41} and another between S_{32} and S_{42}. In addition, there are implicit constraints. We introduce an explicit compatibility constraint between $\hat{M}_3$ and $\hat{M}_4$. Figure 3.3a shows the corresponding abstract microservice set with the representative containers whose latencies are chosen, the bundles and the explicit compatibility constraints. ■

3.5.2 Solution Generation

We now solve for the minimum latency solution on the generated space of abstract microservices using the baseline ILP as earlier in Section 3.4. The objective now is to identify a set of

Algorithm 3: Initial Abstraction Generation

```
   Input  : $L = \langle S, Q, W, P \rangle$                          ▷ Original MLCBACC
   Output: $\hat{L} = \langle \hat{S}, \hat{Q}, W, \hat{P} \rangle$                          ▷ Abstract MLCBACC
1  $\hat{M} \leftarrow NULL$
2                                                              ▷ Generate Singletons
3  foreach $M_i \in W.M$ do
4      |                                  ▷ W.M implies microservices in Workflow W
5      |  $minindex \leftarrow 1$
6      |  $minlatency \leftarrow NULL$
7      |  for $j$ from $(minindex + 1)$ to $|M_i|$ do
8      |      |  if $S_{ij} < minlatency$ then
9      |      |      |  $minindex \leftarrow j$
10     |      |      |  $minlatency = latency(S_{ij})$
11     |  $\hat{Q}_i \leftarrow S_{i\{minindex\}}$ , $\hat{Q}_i.latency \leftarrow minlatency$
12     |  $\hat{M}_i \leftarrow \hat{Q}_i$
13                                                              ▷ Generate Bundles
14 $absedges \leftarrow NULL$
15         ▷ $absedges$ is a hash from $\hat{q}_i$ to $([q], latency)$ for each $i$-element bundle, $\forall 1 \leq i \leq n$
16 foreach bundle $q \in Q$ do
17     |  $\hat{q}_i \leftarrow$ set of microservices $\in M$ corresponding to $q$
18     |  if $\hat{q}_i \notin absedges$ then
19     |      |  absedges$[\hat{q}_i] \leftarrow [q], latency$
20     |  else
21     |      |  if $absedges[\hat{q}_i].latency > q.latency$ then
22     |      |      |  absedges$[\hat{q}_i] \leftarrow [q], latency$
23                                              ▷ Generate Compatibility Constraints
24 foreach $p \in P$ do
25     |  $(\hat{M}_{ind}) \leftarrow$ set of indices in $\hat{M}$ corresponding to $p$
26     |  add compatibility constraint between abstract microservices corresponding to $(\hat{M}_{ind})$
```

abstract bundles from the set $\hat{Q}$ such that the latency is minimum, each abstract microservice is chosen, and compatibility constraints between the abstract microservices are honored. Once we obtain an abstract solution, we reproduce the same on the concrete containers, by mapping back the abstract $\hat{q}_i$s chosen to the concrete microservices, and then choosing the containers corresponding to the concrete containers for each such microservice. This is necessary since the solution has to be generated from the concrete container space and not the abstract container space and such a solution must also conform to concrete compatibility constraints.

We choose the minimum latency singleton bundles for each abstract microservice, and also the minimum latency among composite bundles comprising multiple microservices. It is therefore guaranteed that the minimum latency solution obtained from the abstract space is indeed the minimum latency solution in the concrete space, as formally stated later. If the solution meets all the compatibility constraints as well, we terminate claiming this is the minimum latency solution possible. However, it may as well so happen that we have a compatibility constraint violation in the corresponding concrete solution. Such a solution can exist because we introduce a compatibility constraint between abstract microservices $\hat{M}_i$, $\hat{M}_{i+1}$ whenever there exists a compatibility constraint between some member in S_i with some members in S_{i+1} of the microservices. When the abstract microservice $\hat{M}_i$ is mapped back to M_i and the container corresponding to its minimum latency singleton is chosen (call it S_{im}) and the abstract microservice $\hat{M}_{i+1}$ is mapped back to M_{i+1} and the container corresponding to its minimum latency singleton is chosen (call it $S_{(i+1)n}$), it may happen that S_{im} is not compatible with $S_{(i+1)n}$, but with some other member $S_{(i+1)q}$, $q \neq n$. The actual compatibility constraints have been abstracted away by our abstraction and this leads to a violation. We need to refine the abstraction in such cases. Algorithm 4 summarizes the solution generation and refinement process.

Example 3.5.4. In Figure 3.3a, the compatibility constraints between S_{32} and $\{S_{41}, S_{42}\}$ in the original repository are no longer present as S_{41} and S_{42} are not part of the abstraction. Their compatibility constraint has been replaced by a constraint between $\hat{M}_3$ and $\hat{M}_4$. ■

3.5.3 Solution Validation

Let us consider a solution ψ obtained using the ILP on the abstraction (Line 3). This solution comprises at least one container for each microservice M_i in the workflow. To ensure that ψ is a valid solution, we need to ensure that the compatibility constraints are satisfied. We first define a compatibility constraint violation.

Definition 3.9 [**Compatibility Constraint Violation:**]
For a constraint $p_k \in P$ of the form $S_{ij} \rightarrow \{S_{(i+1)m}\ ,\ S_{(i+1)n}\ ,\ \ldots\}$, if $S_{ij} \in \bigcup Q'$, then a compatibility constraint violation occurs if none among the set $S_{(i+1)m}$, $S_{(i+1)n}$, $\ldots \notin \bigcup Q'$.

We iterate through each compatibility constraint $S_{ij} \rightarrow \{S_{(i+1)m}, S_{(i+1)n} \ldots\}$ in the concrete space and check for satisfaction. If S_{ij} is present in ψ we check whether at least one among $S_{(i+1)m}$, $S_{(i+1)n} \ldots$ is also present in ψ or not. If not, a violation is detected, necessitating a refinement. However, if all constraints are indeed satisfied, we terminate returning the valid solution. Lines 4-15 perform this validation and return ψ if a valid solution is indeed obtained.

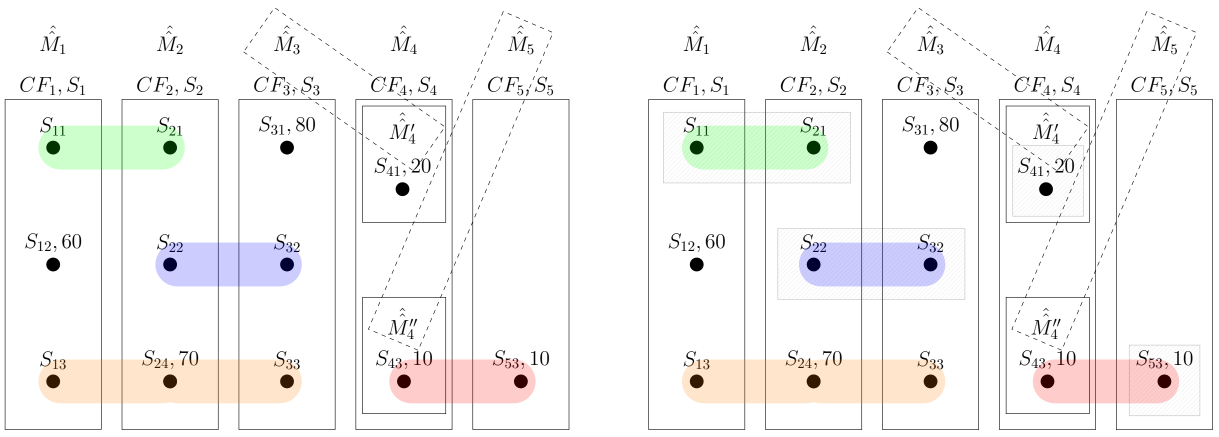
Example 3.5.5. Figure 3.3b shows the solution generated on the initial abstraction. The containers involved in the bundles are shown with filled dashed rectangles. The solution corresponds to $\{\hat{q_{12}}, \hat{q_{23}}, \hat{q_{45}}\}$ as highlighted in Table 3.5. The identified containers are S_{11} for microservice M_1, S_{21} and S_{22} for microservice M_2, S_{32} for microservice M_3, S_{43} for microservice M_4 and S_{53} for microservice M_5. This solution has a minimum latency of 115ms. However, the compatibility constraint for S_{32} is violated since it can work only with S_{41} or S_{42} neither of which is identified in ψ. This solution was generated because we introduced a compatibility constraint between $\hat{M}_3$ and $\hat{M}_4$ in lieu of the compatibility constraint between S_{32} and $\{S_{41}, S_{42}\}$. ■

Algorithm 4: Solution Generation and Refinement

```
   Input  : $\hat{L} = \langle \hat{S}, \hat{Q}, W, \hat{P} \rangle$                    ▷ Abstract MLCBACC
   Output: Minimum latency Solution for $\hat{M}$
1  $valid = false$
2  while $valid == false$ do
3      $\psi \leftarrow$ solve $\hat{M}$ using baseline ILP
4      $msol \leftarrow NULL$
5      foreach $\hat{q} \in \hat{M}_{opt}$ do
6          $msol[\hat{q}] \leftarrow$ concrete $q$ corresponding to $\hat{q}$
7      $sat \leftarrow true$
8      foreach $M_i \in W$ do
9          $S_{im} \leftarrow$ concrete $s$ in $msol$ for $M_i$
10         $S_{(i+1)n} \leftarrow$ concrete $s$ in $msol$ for $M_{i+1}$
11         foreach $s$ compatible with $S_{im}$ do
12             if at least one of $s$ is not present in $\psi$ then
13                 $sat \leftarrow false$
14     if $sat == true$ then
15         $valid = true$                                            ▷ Valid Solution
16     else
17         foreach compatibility constraint $S_{ij} \rightarrow S_{(i+1)m}$ that is violated do
18             $\hat{M}_i, \hat{M}_{i+1} \leftarrow$ abstract microservices corresponding to violating $s$
19             split $\hat{M}_{i+1}$ into $\hat{M}'_{i+1}$, $\hat{M}''_{i+1}$
20             $\hat{M}'_{i+1} \leftarrow$ containers compatible with $S_{ij}$
21             $\hat{M}''_{i+1} \leftarrow$ containers not compatible with $S_{ij}$
22             $\hat{q} \leftarrow \hat{q} \cup [s]$ corresponding to $\hat{M}'_{i+1}$
23         foreach $\hat{M}'_{i+1}$ where constraints are violated do
24             $\hat{q} \leftarrow \hat{q} \cup [s]$ corresponding to bundles for $\hat{M}'_{i+1}$ or $\hat{M}''_{i+1}$      ▷ Expose Bundles
25 return $\psi$
```

3.5.4 Refining The Abstraction

Consider a solution ψ obtained on the abstraction. Additionally, consider the compatibility constraint $S_{ij} \rightarrow \{S_{(i+1)m}, S_{(i+1)n}, \ldots\}$ for which a violation occurred in ψ. The abstract microservices corresponding to S_{ij} and $\{S_{(i+1)m}, S_{(i+1)n}, \ldots\}$ are $\hat{M}_i$ and $\hat{M}_{i+1}$ respectively. We split $\hat{M}_{i+1}$ into $\hat{M}'_{i+1}$ and $\hat{M}''_{i+1}$. $\hat{M}'_{i+1}$ includes all containers in M_{i+1} which are compatible



(a) Refined Abstraction (b) Refined Abstraction Solution

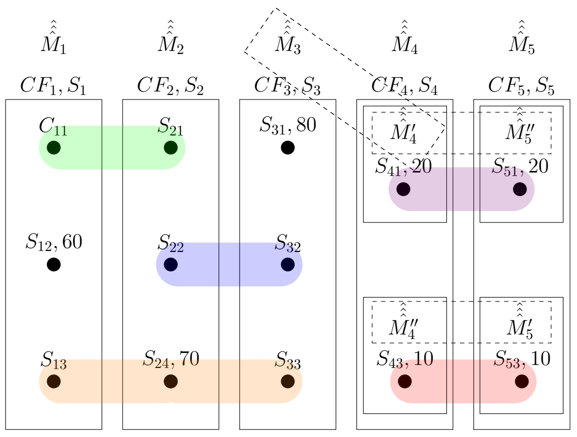
FIGURE 3.4: Refined Abstraction

with S_{ij}, and $\hat{\hat{M}}''_{i+1}$ includes the rest of the containers in M_{i+1}. Splitting the microservices results in a new microservice level partition of the container repository, for those microservices where a compatibility constraint violation occurred. The abstract bundles for this refinement are constructed as earlier. We create a singleton bundle corresponding to $\hat{\hat{M}}'_{i+1}$, assign it the minimum singleton bundle latency among the containers inside, and we do the same for $\hat{\hat{M}}''_{i+1}$ as well. Further, new concrete bundles are exposed as a result of this refinement between the containers in the other abstract microservices and these two, and we create them as earlier. Also, compatibility constraints are marked between the abstract microservices as earlier. Lines 17 - 24 refine the abstraction updating bundles $\hat{q}$ and the constraints. Lines 23-24 consider all violating constraints and update q considering the new abstract microservices created as a result of the refinement process to expose further bundles.

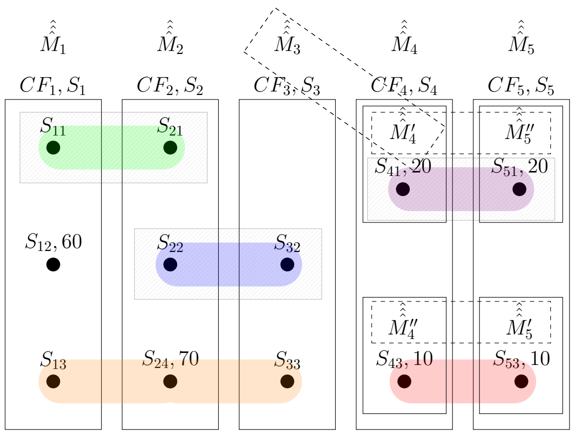
Example 3.5.6. To cater to the violation $S_{32} \rightarrow \{S_{41}, S_{42}\}$, we split $\hat{M}_4$ into $\hat{\hat{M}}'_4$ and $\hat{\hat{M}}''_4$, as in Figure 3.4a. In this case, there are no more bundles to reconsider. The new abstraction generated after refinement is depicted in Figure 3.4a. ■

We proceed to solve the minimum latency bundle ILP for this abstract set of microservices with the compatibility constraints thus obtained. Once a solution is obtained, we validate whether it meets all concrete compatibility constraints. If it does, we declare it as the desired solution. However, if it violates any constraint, we refine as earlier. This process is repeated until we find a solution that is valid (i.e., when all the compatibility constraints are satisfied), or no solution is obtained, in which case we conclude that there is no solution satisfying the concrete compatibility constraints.

Example 3.5.7. The new minimum latency solution on the refined abstraction is shown in Figure 3.4b. The identified containers are highlighted with filled dashed rectangles. This solution corresponds to $\{\hat{\hat{q}}_5, \hat{\hat{q}}_{12}, \hat{\hat{q}}_{23}, \hat{\hat{q}}'_4\}$ and is highlighted in Table 3.5. The latency of the solution



(a) Final Abstraction



(b) Final Abstraction Solution

FIGURE 3.5: Final Abstraction

Initial Abstraction	
$\hat{q}_1$	S_{12}, 60ms
$\hat{q}_2$	S_{24}, 70ms
$\hat{q}_3$	S_{31}, 80ms
$\hat{q}_4$	S_{43}, 10ms
$\hat{q}_5$	S_{53}, 10ms
$\hat{q}_{12}$	S_{11}, S_{21}, 50ms
$\hat{q}_{23}$	S_{22}, S_{32}, 50ms
$\hat{q}_{45}$	S_{43}, S_{53}, 15ms
$\hat{q}_{123}$	S_{13}, S_{24}, S_{33}, 150ms

Refined Abstraction	
$\hat{\hat{q}}_1$	S_{12}, 60ms
$\hat{\hat{q}}_2$	S_{24}, 70ms
$\hat{\hat{q}}_3$	S_{31}, 80ms
$\hat{\hat{q}}_4$	S_{43}, 10ms
$\hat{\hat{q}}_5$	S_{53}, 10ms
$\hat{\hat{q}}_{12}$	S_{11}, S_{21}, 50ms
$\hat{\hat{q}}_{23}$	S_{22}, S_{32}, 50ms
$\hat{\hat{q}}_{45}$	S_{43}, S_{53}, 15ms
$\hat{\hat{q}}_{123}$	S_{13}, S_{24}, S_{33}, 150ms
$\hat{\hat{q}}'_4$	S_{41}, 20ms

Final Abstraction	
$\hat{\hat{\hat{q}}}_1$	S_{12}, 60ms
$\hat{\hat{\hat{q}}}_2$	S_{24}, 70ms
$\hat{\hat{\hat{q}}}_3$	S_{31}, 80ms
$\hat{\hat{\hat{q}}}_4$	S_{43}, 10ms
$\hat{\hat{\hat{q}}}_5$	S_{53}, 10ms
$\hat{\hat{\hat{q}}}_{12}$	S_{11}, S_{21}, 50ms
$\hat{\hat{\hat{q}}}_{23}$	S_{22}, S_{32}, 50ms
$\hat{\hat{\hat{q}}}_{45}$	S_{43}, S_{53}, 15ms
$\hat{\hat{\hat{q}}}_{123}$	S_{13}, S_{24}, S_{33}, 150ms
$\hat{\hat{\hat{q}}}'_4$	S_{41}, 20ms
$\hat{\hat{\hat{q}}}'_5$	S_{51}, 20ms
$\hat{\hat{\hat{q}}}'_{45}$	S_{41}, S_{51}, 35ms

TABLE 3.5: Bundles in the Stages of the Abstraction Refinement Process

is 130ms. This solution now violates the compatibility constraint between S_{41} and S_{51}. We refine the abstraction again, and introduce $\hat{\hat{\hat{M}}}'_5$ and $\hat{\hat{\hat{M}}}''_5$ and assign latencies and constraints accordingly. We now reconsider the bundles between $\hat{\hat{M}}'_4$ and $\hat{\hat{\hat{M}}}''_5$ and retain the minimum latency bundle which corresponds to the bundle between S_{41} and S_{51} having a latency of 35ms. The resulting abstraction is shown in Figure 3.5a. The solution on this abstraction is shown in Figure 3.5b which corresponds to $\{\hat{\hat{\hat{q}}}_{12}, \hat{\hat{\hat{q}}}_{23}, \hat{\hat{\hat{q}}}'_{45}\}$, as in Table 7, with latency 135ms which satisfies all compatibility constraints. We terminate with this solution. ■

We continue iterating until we obtain a valid solution. The worst case arises when a realizing solution is not obtained on an abstraction of the container repository. In such a scenario, all the containers of the repository are involved in computing the realizing solution. As a consequence, our approach incurs additional running time in such scenarios leading to a performance worse than the baseline ILP. However, we fare much better on an average as show in Section 3.6.

3.5.5 Proof of Correctness

We now discuss the proof of correctness and termination of our framework.

Lemma 3.2. *Given an MLCBACC problem $L = \langle S, Q, W, P \rangle$, and a microservice level partition $\tilde{S}$, let $\hat{L}$ be the abstract MLCBACC problem $\langle \hat{S}, \hat{Q}, W, \hat{P} \rangle$ with respect to $\tilde{S}$. If there is a compatible solution Q' in L, there is also a realizing, compatible solution $\hat{Q}' \in \hat{L}$. Further, $latency(\hat{Q}') \leq latency(Q')$.*

Proof: Given a realizing and compatible solution $Q' \subseteq Q$ for L, define $\hat{Q}' = \{[q] \mid q \in Q'\}$, where $[q] = \{[s] \mid s \in q\}$. Since Q' is a realizing solution in L, for every M_i, there is a $s_i \in S_i$ and a bundle $q \in Q'$ such that $s_i \in q$. By construction, $[s_i] \in \hat{S}_i$, and $[q] \in \hat{Q}'$. Therefore, $\hat{Q}'$ is realizing in $\hat{L}$. Now, suppose there is a constraint $\hat{M}_{[s]} \implies \hat{M}_k$ in $\hat{P}$, the set of abstract compatibility constraints, and assume $[s] \in \bigcup \hat{Q}'$, where $\hat{M}_{[s]}$ is the abstract microservice corresponding to $[s]$. Let $S_{ij} \in [s]$ be a container that is in $\bigcup Q$. By construction of the constraints, there is a microservice M_k such that $S_{ij} \to \{S_{(i+1)1}, \ldots, S_{(i+1)k}\}$ is a constraint in P. Since S_{ij} is in the concrete solution that must be compatible with P, $M_k \in \bigcup Q'$. This proves that $\bigcup \hat{Q}'$ is compatible with $\hat{P}$. Lastly, by construction, $latency([q]) \leq latency(q')$ for all $q' \in [q]$, since the latency of $[q]$ is assigned to the minimum latency bundle q_m such that $[q] = [q_m]$. Therefore, $latency(\hat{Q}') \leq latency(Q')$. ♦

An immediate corollary of the previous lemma is as follows.

Corollary 3.3. *If there exists a minimum latency solution Q' for MLCBACC, then the minimum latency solution for any microservice-level abstraction of the problem has latency less than that of Q'.* ♦

Lemma 3.4. *The abstraction refinement algorithm terminates and outputs a realizing, compatible and minimum latency solution of the MLCBACC problem if one exists, and terminates with no solution if none exists.*

Proof: Starting from the initial abstraction, at each abstraction level, we attempt to solve the abstract MLCBACC problem. If there is no solution, we terminate with the assertion that no solution exists for the concrete MLCBACC problem as well, as discussed in Lemma 3.2. If there is a solution for the abstract problem, we check if the concrete witness is also a solution for the concrete MLCBACC problem. If yes, then by Corollary 3.3, the concrete witness is the minimum latency concrete solution. Therefore, we terminate with the concrete witness as the final solution. If the concrete witness is spurious, then the abstract problem is refined. Note that the refinement is strict i.e. the number of partitions in at least one task strictly increases. The new abstract problem is subjected to the above steps. Therefore, either we find a result (positive or negative) at an abstraction level or eventually end up in the concrete MLCBACC problem where we terminate with the result (positive or negative). ⧫

3.6 Results and Discussion

We carry out experiments on latency values generated from the WS-DREAM dataset. We perform a number of experiments to demonstrate the impact of variance in number of microservices, number of containers per microservice, the degree of bundles in the repository, the composition of each bundle and the number of compatibility constraints in the repository. In order to demonstrate the impact on scalability, we gradually increase the value of the different parameters such as the number of microservices, containers, etc. and study the comparative performance of different algorithms. All experiments are carried out with Python 3.7 using Gurobi [108] as the ILP solver on an Intel Core i5 Processor with 16GB of RAM. We set the maximum memory limit for Python to $4GB$. We use Gurobi since it has a warm start feature which allows an already computed solution for an ILP to be used for any subsequent modifications to the model in terms of variables or constraints [108].

3.6.1 Experimental Setup

For each experiment, we allocate QoS values to the containers randomly from the WS-DREAM dataset. Then, for each container, we assign latencies to containers by normalizing all allotted QoS values to containers between 0 and 1. We then assign a latency in the range of 30ms to 100ms with lower latencies being assigned to containers having high QoS values. Using this technique, a container with a high QoS value has a low latency. In all the experiments, we generate the workflows as explained later in individual experiments. To evaluate the effectiveness of the approach, we perform two sets of experiments:

1. keeping number of containers, bundles, constraints, microservices small to simulate small scale scenarios

2. simulation of large scale scenarios by incorporating a large number of microservices comprising a large number of containers, bundles and constraints.

We explain each scenario in the following subsections.

3.6.2 Comparison With Existing Selection Models

In this section, we compare the effectiveness of our framework with the model most similar to our work in literature. The Correlation-Aware Service Pruning (CASP) [107] algorithm deals with service selection similar to service allocation but with pairwise bundles and no compatibility constraints. To compare with our framework, we set all bundles in the following scenarios to pairwise, i.e, between two containers with no compatibility constraints.

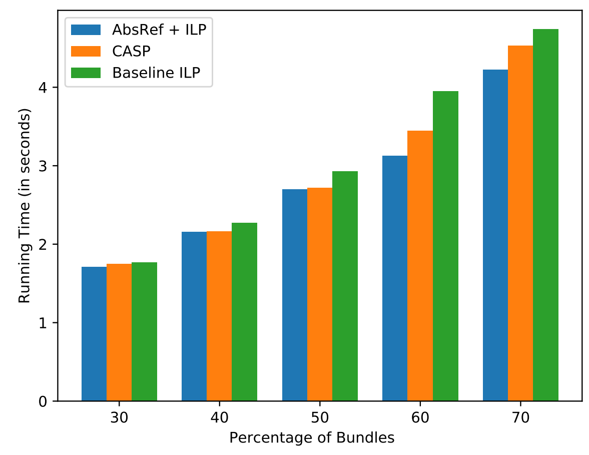
Impact of Percentage of Bundles: To evaluate the impact of percentage of bundles, we set the number of microservices to 10 as has been done in [107]. We set the number of containers for each microservice to 50. We assign to each such container a latency value as mentioned in the above discussion. We consider the total number of service containers in the repository and create random bundles involving 10 percent of such service containers. The bundle percentage is then varied as $20, 30, \ldots$, 80. As can be inferred from Figure 3.6a, as the percentage of bundles is increased, there is a general increase in the running time of the algorithms. This is because with an increase in the number of bundles, the number of indicator variables corresponding to both singletons and bundles increases for both the baseline ILP and the Abstraction Refinement + ILP framework. However, the Abstraction Refinement + ILP framework effectively prunes the bundles by retaining only the minimum cost n-bundle as explained in Section 3.5 for the tasks corresponding to each n-bundle. Thus it fares better than the baseline ILP and CASP, both of which explore the search space microservice wise retaining the best bundled pairs explored in each iteration leading to a far greater exploration of the search space.

Impact of Number of Containers: We now set the value of the bundle containers percentage to 30 with respect to the total number of containers. The values of the other parameters are set as in the above experiment. We vary the number of containers for each microservice as $\{100, 200, \ldots, 500\}$. The results are shown in Figure 3.6b. There is a general increase in the running time of both the algorithms when the number of containers per microservice is increased. With an increase in the number of containers per microservice, the exploration search space increases, thereby leading to an overall increase in running time. Even when the number of containers is increased, our framework performs much better than CASP and the naive baseline ILP while offering maximum speed-up when the number of containers is high.

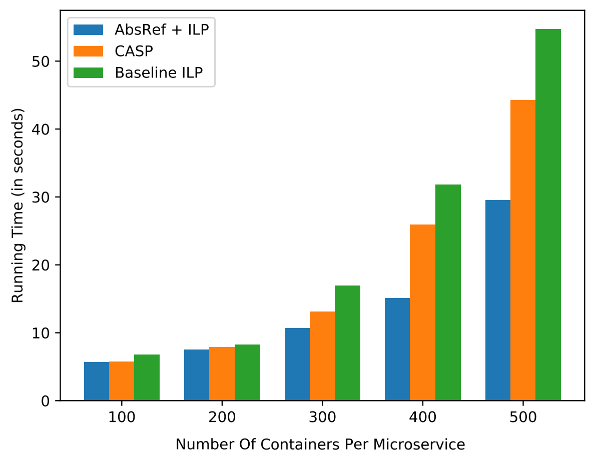
Impact of Constant Number of Bundles: In this case, we vary the number of containers but keep a fixed number of bundles. We set the number of bundles fixed at 100. The values of the other parameters are set as in the above experiment. We vary the number of containers for each task as $\{100, 200, \ldots, 500\}$. Figure 3.6c shows the results obtained. It is interesting to note that Abstraction-Refinement + ILP obtains a significant speed-up over CASP and the baseline ILP in such scenarios. Additionally, with an increase in number of containers, the time taken by our framework does not increase as much as the time incurred by CASP and the baseline ILP. This is because our framework effectively prunes the search space retaining only the best bundles. With only an increase in containers which do not participate in bundles, the pruning becomes even more effective amongst singleton containers.

Impact of Number of Microservices: In order to evaluate the impact of the number of microservices in the workflow on the algorithm, we vary the number of microservices as $10, 20, \ldots, 50$ while keeping the bundle percentage at 30 with each microservice comprising 50 containers. As the number of microservices increases, there is a general increase in running times as observed in Figure 3.6d. Varying the number of microservices implies an overall variance in the number of containers. With an increase in the number of microservices, there is a general increase in the running time of the algorithms. This is in concordance with the previous experimental scenario as well where we varied the number of containers per microservice. Our framework provides a speed-up in these scenarios as well.

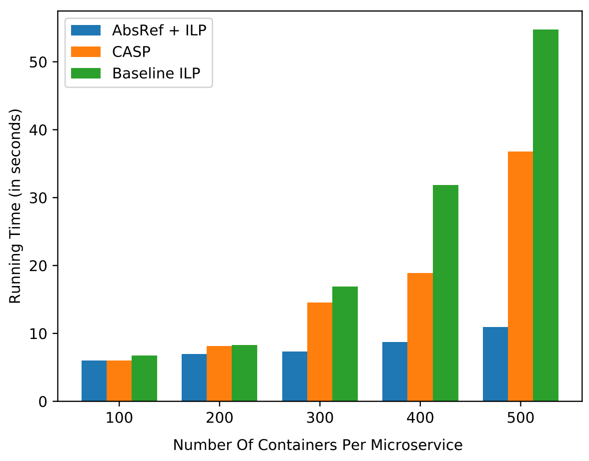
Large Scale Scenarios: In order to demonstrate the scalability of our framework, we perform similar experiments as described above: i) we keep the number of microservices fixed at 100 with 200 containers per microservice with the percentage of bundles varied as $\{30, 40 \ldots 70\}$ ii) we vary the number of containers per microservice as $\{100, 200, \ldots 500\}$ while keeping the number of microservices fixed at 100 and percentage of bundles at 30 iii) we vary the number of microservices as $\{1000, 2000, 3000\}$ to simulate large scale scientific workflows. Figures 3.7a and 3.7b demonstrate the results of varying the percentage of bundles and containers respectively. Table 3.7 lists the impact of a large number of microservices. In this case, our framework performs much better than the baseline ILP in comparison to the experiments carried out on the small scale scenarios. When the number of microservices is varied in the large scale scenario, both CASP and the baseline ILP run out of memory once the number of microservices is increased to 2000. Our framework is able to generate results even in such scenarios since the Abstraction Refinement + ILP approach effectively retains only a partial view of the container repository, refining as and when required.



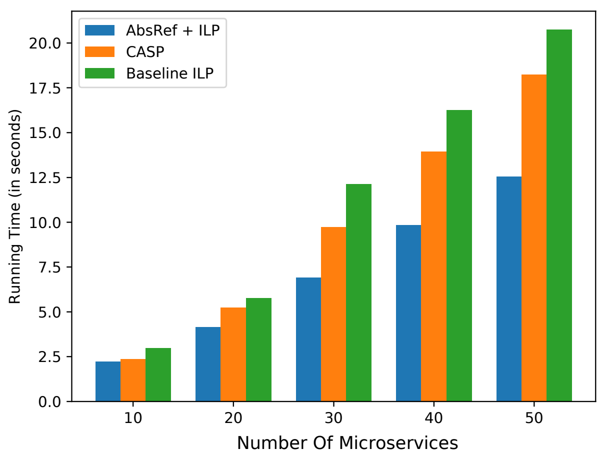
(a) Varying Percentage of Bundles



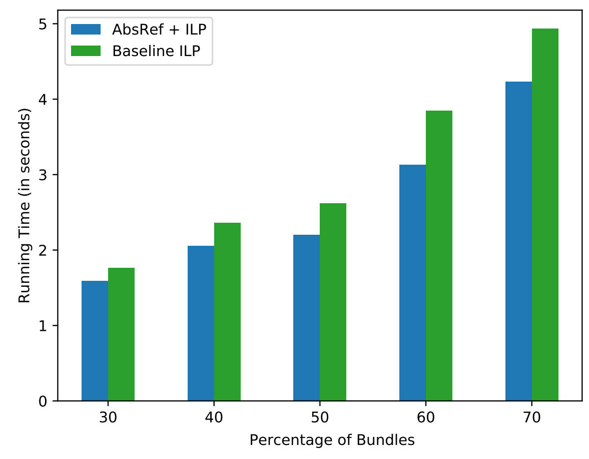
(b) Varying Number of Containers



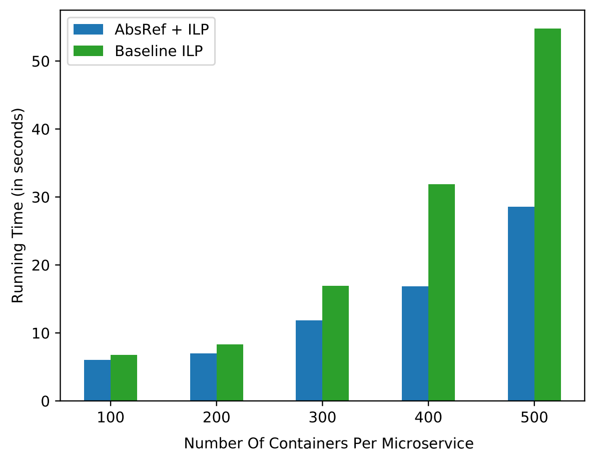
(c) Varying Number of Containers - Fixed Bundles



(d) Varying Number of Microservices

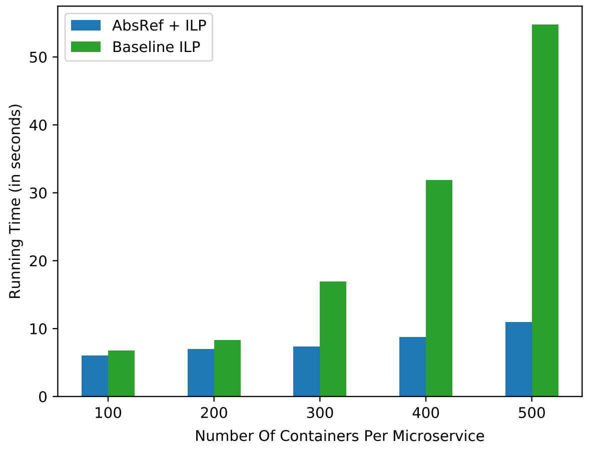


(e) Varying Percentage of Package Bundles

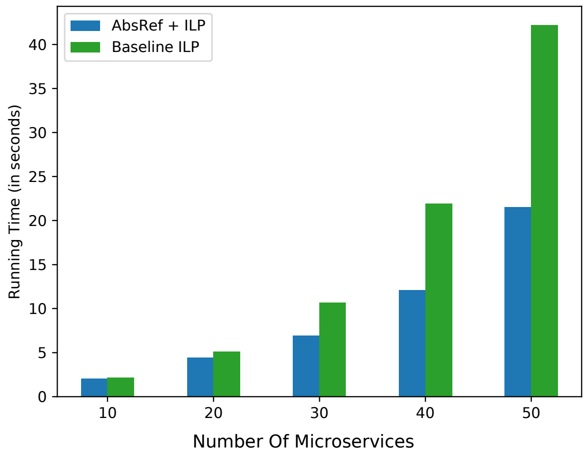


(f) Varying Number of Containers per Microservice

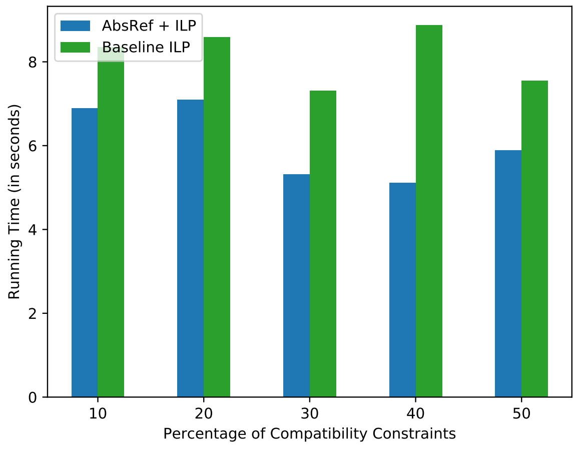
FIGURE 3.6: Performance Comparison for Small Scale Scenarios



(g) Varying Number of Containers - Fixed Bundles



(h) Varying Number of Microservices



(i) Varying Number of Compatibility Constraints

FIGURE 3.6: Performance Comparison for Small Scale Scenarios

3.6.3 Effectiveness of our ILP + Abstraction Refinement Framework

In this section, we evaluate the effectiveness of the proposed model with compatibility constraints and generic bundles (not necessarily pairwise).

Impact of Percentage of Bundles, Number of Containers, Constant Number of Bundles and Number of Microservices: Retaining all the other parameters as above, we repeat the same experiments but with bundles incorporating multiple containers, i.e, more than 2 containers per bundle, with compatibility constraints fixed at 30%. The results in Figure 3.6e, 3.6f, 3.6g, 3.6h show that there is no significant difference in all of the scenarios described for small scale scenarios in comparison with existing selection models. Thus, the presence of bundles

%Bundles	Refinements
30	4
40	11
50	7
60	8
70	15

(A) Figure 3.6e

Containers	Refinements
100	17
200	22
300	15
400	19
500	28

(B) Figure 3.6f

Containers	Refinements
100	8
200	11
300	7
400	11
500	14

(C) Figure 3.6g

Microservices	Refinements
10	8
20	14
30	28
40	24
50	31

(D) Figure 3.6h

% Constraints	Refinements
10	8
20	10
30	5
40	5
50	7

(E) Figure 3.6i

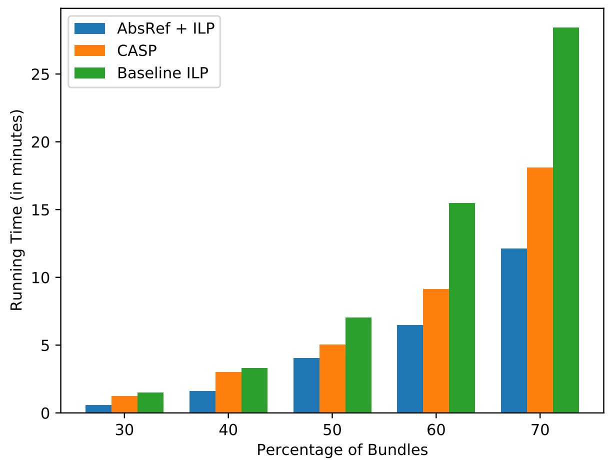
TABLE 3.6: Number of Refinements for corresponding Small Scale Scenarios in Figure 3.6

which span across multiple microservices instead of just pairwise bundles, does not affect the performance of our proposed framework.

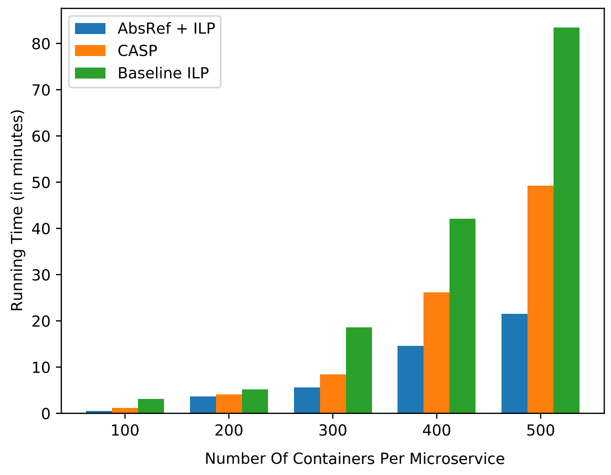
We now discuss the effect of varying some parameters which are unique to the bundle model incorporating multiple containers and compatibility constraints.

Impact of Composition of Bundles: We set the number of microservices as 200 with each microservice having 200 containers. Additionally, we set the percentage of bundles to 30 and percentage of constraints to 30 both with respect to the total number of service containers. We then vary the length of bundles, i.e, the number of containers involved in bundles, as $\{3, 5, \dots 11\}$ as shown in Figure 3.7c. There is no definite pattern (increasing / decreasing) as the length of the bundles are varied. This is because there is no definite correlation between the length of the bundles and the optimal solutions to an instance of MLCBACC which rather depends on how these bundles are distributed amongst the microservices and the degree of their compatibility.

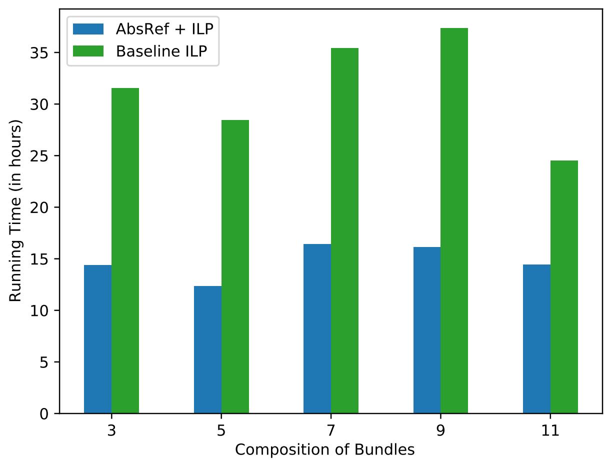
Impact of Compatibility Constraints: To demonstrate the impact of compatibility constraints, the percentage of total number of containers in the repository which include compatibility constraints is varied as $\{10, 20, 30, \dots, 50\}$. It is interesting to note that when we vary the number of compatibility constraints, there is no direct correlation observed between the percentage of compatibility constraints and the running times of the algorithms as is observed in Figure 3.6i. Such a scenario occurs because spurious compatibility constraints between abstract microservices are generated only when they are not part of the minimum latency bundle retained in the initial abstraction. If the minimum latency bundle is indeed compatible with the corresponding subsequent bundles in the workflow, spurious solutions can be avoided hence resulting in lower running times. As noted in Figure 3.6i, when the percentage of compatibility constraints is increased from 30 to 40, there is a decrease in the running time of the Abstraction Refinement



(a) Varying Percentage of Bundles



(b) Varying Containers Per Microservice



(c) Varying Composition of Bundles

FIGURE 3.7: Performance Comparison for Large Scale Scenarios

+ ILP approach. However, when the percentage is further increased to 50, there is an increase in the running time. This shows that the distribution of bundles along with the corresponding compatibility constraints play a crucial role in determining the running times of the algorithms. The same holds for the baseline ILP. Additionally, we vary the percentage of compatibility constraints in $\{10, 20, 30\}$ for large scale scenarios as listed in Table 3.8. The baseline ILP incurs a memory out error in all the large scale scenarios due to the large number of microservices, containers and percentage of containers involved in bundles and compatibility constraints. On the other hand, our framework is able to generate results even for such scenarios.

Number of Refinements Required: Table 3.6 summarizes the number of refinements required for the different small scale scenarios demonstrated in Figure 3.6 while Table 3.9 lists the number

Running Time (in Hours)			
Microservices	AbsRef + ILP	CASP	ILP
1000	0.40	2.15	3.40
2000	1.58	MemoryOut	MemoryOut
3000	3.35	MemoryOut	MemoryOut

Table 3.7: Performance Comparison for Varying Microservices

Running Time (in Hours)		
Constraints %	AbsRef + ILP	ILP
10	1.55	MemoryOut
20	4.40	MemoryOut
30	3.58	MemoryOut

Table 3.8: Performance Comparison for Varying Constraints

Bundles		Constraints		Microservices	
Bundles	Rounds	%Constraints	Rounds	%Tasks	Rounds
3	12	10	19	1000	42
5	18	20	38	2000	31
7	15	30	32	3000	38

Table 3.9: Number of Refinements

of refinements required for large scale scenarios. As can be inferred from Table 3.9, there is no direct correlation between the various parameters and the number of refinements since there is no definite pattern observed. Such a scenario occurs since the number of refinements depends on the correlation between the minimum latency bundles which are used to construct the initial abstraction and their compatibility with the containers retained in the abstraction for the subsequent microservices. Since refinements only occur for incompatibilities, the nature of distribution of bundles and their association plays the determining role in deciding how many refinements are needed. However, note that when we vary the number of compatibility constraints while keeping other parameters constant, there is a direct correlation between the running time and the number of refinements. With an increase in the number of refinements, the running time increases. This is expected since more bundles are added to successive refined abstractions. In scenarios where we set compatibility constraints to empty, no refinements are required since a refinement only occurs upon violation of such constraints and hence are not summarized in the tables.

3.7 Conclusion

In this chapter, we present an abstraction refinement approach to expedite the minimum latency container service selection problem. We provide a hardness proof, and follow it up with an ILP based approach for solving the same. Finally, we present a proposal to expedite the ILP with abstraction refinement to improve performance. We consider static scenarios, wherein we cater to the minimum latency container bundle selection problem where containers are pre-deployed on MEC servers and the locations of users are known apriori. In the next chapter, we consider a dynamic allocation policy where the service placement configuration as well as the user allocations adapt to user mobility.

Chapter 4

Proactive Microservice Placement and Allocation

4.1 Introduction

In the previous chapter, we designed a microservice allocation policy for scenarios with a predefined service placement configuration and known user locations. To overcome the limitations of such a static approach, in this chapter, we consider the impact of user mobility and design a dynamic, joint microservice placement and allocation policy. In recent years, several dynamic placement and allocation policies for monolithic applications considering different scenarios and optimization metrics for application service provisioning in the MEC context have been proposed in literature [3, 4, 51, 97]. However, most of these solutions need to be re-examined today through a different lens, considering the recent paradigm shift in the application provisioning model, from a monolithic service architecture to the micro-service deployment model, that is being increasingly adopted across the service industry by service providers like Amazon, Netflix [48]. The problem of service placement and allocation in the microservice context in MEC is much more complex than the one for their monolithic counterparts, considering the

This work is published as:

- Kaustabha Ray, Ansuman Banerjee, and Nanjangud C. Narendra. "Proactive Microservice Placement and Migration for Mobile Edge Computing", In Proceedings of IEEE/ACM Symposium on Edge Computing, pp. 28-41, 2020.

inter-dependencies that need to be accounted for, while deploying a solution. Additionally, for mobility-aware service placement and allocation, state migration also needs to be considered. Indeed, in recent literature, only a small handful of proposals [109, 110, 111], to the best of our knowledge, have focused on the microservice model in the MEC context. However, these approaches neither take into consideration the non-trivial latencies involved with microservice containers nor do they take into account user mobility patterns. We consider both in this chapter in addition to the factors considered by those existing methods. This is the main context in MEC that we attempt to address in this chapter.

Traditional placement and allocation policies focus on *reactive* placement, i.e., microservice placement after a service invocation request originates. Our main proposal in this chapter is a *proactive* microservice placement and migration approach by prefetching microservices considering the workflow dependency structure in the application microservice workflow. This aids to abate service deployment latencies. Proactively prefetching microservices is a complex task due to: i) the large configuration space of the mobility of devices coupled with microservice interdependencies; ii) the unpredictability of edge servers as an operating environment due to the dynamic and on demand nature; and iii) the stochasticity of user service requests while having a myriad of mobility and service invocation patterns.

In this chapter as well, for simplicity and ease of explanation, we focus on applications with workflows defined by a linear sequence of microservices. To the best of our knowledge, this is the first work that exploits the microservice dependency task structure to prefetch and pre-provision microservices to better meet latency requirements in MEC. We use a Markov Decision Process (MDP) with rewards to model proactive microservice placement and migration. Further, since the rewards corresponding to each state of the MDP are unknown, we use Reinforcement Learning (RL) to demonstrate how to learn the unknown rewards to effectively deploy and migrate services. In particular, to make effective use of the MDP, we use the Dyna-Q [34] algorithm, which is a combination of model free and model based RL. Additionally, to cater to different service request traffic patterns, we design a heuristic to adapt to varying traffic loads. We present experimental results of our algorithm in practical scenarios driven by real-world mobility traces of taxis in San Francisco [97] and timing characteristics obtained from the DeathStarBench microservice benchmark suite [48]. Our analysis reveals an average 28% improvement in latency obtained using our proactive approach over the traditional reactive one, for some state-of-the-art MEC microservice benchmark models.

The rest of this chapter is organized as follows. Section 4.2 presents an example to be used in the rest of this chapter. Section 4.3 describes the problem formulation. Section 4.4 describes the formal model. Section 4.5 describes our RL-based approach. Section 4.6 describes our implementation along with experimental details. Section 4.7 concludes the chapter.

4.2 Motivating Example

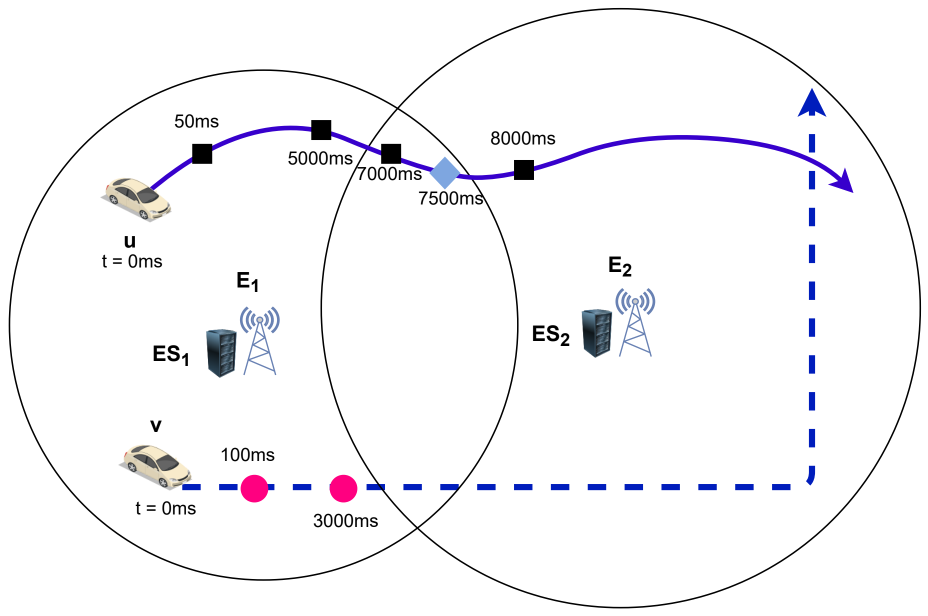


Figure 4.1: Microservice Invocations by Vehicular Users

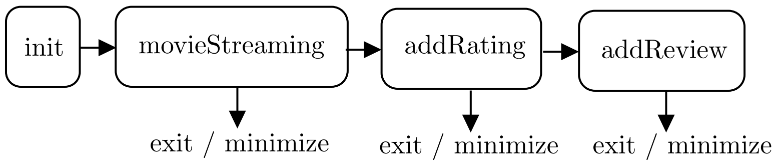


Figure 4.2: Movie Streaming Application Microservice

In this section, we present a motivating example to explain the problem context addressed in this chapter. Consider two vehicles u and v following the trajectories shown in Figure 4.1. The passengers of the vehicles access several applications using their smartphones, with each application modeled as an almost linear microservice workflow (with special exit / minimise nodes but no branching in control flow). We select a movie streaming application as a representative use case. A user can either access the microservices in the linear sequence or choose to exit / minimise the application. In the minimized state, microservice containers corresponding to the application are retained on the server while a container is removed if it has no active users. The workflow of the application, depicted in terms of its constituent interdependent microservices, is shown in Figure 4.2. The microservices hosted as service containers are deployed by service providers on edge servers with service areas associated with them, as depicted by circles around servers E_1 and E_2 in Figure 4.1.

We now explain in detail the latencies associated with the microservice containers assumed in this chapter based on the provisioning model discussed in Chapter 2. When a user invokes a

Time t	User Action	Server-Service State
$0ms$		No Services Deployed
$50ms$	$u \rightarrow$ movieStreaming	initialize movieStreaming
$75ms$		$ES_1 \rightarrow$ movieStreaming
$100ms$	$v \rightarrow$ movieStreaming	$ES_1 \rightarrow$ movieStreaming initialize new task for v
$110ms$		$ES_1 \rightarrow$ movieStreaming, v_{task}
$3000ms$	v exits movieStreaming	$ES_1 \rightarrow$ movieStreaming
$5000ms$	$u \rightarrow$ addRating	initialize addRating
$5025ms$		$ES_1 \rightarrow$ addRating
$7000ms$	u minimizes addRating	$ES_1 \rightarrow$ addRating
$8000ms$	$u \rightarrow$ addReview	initialize addReview
$8025ms$		$ES_2 \rightarrow$ addReview

Table 4.1: On-demand Placement of Microservices

microservice, the container corresponding to the microservice has to be deployed on an edge server if the container is not already present. Additionally, the corresponding service registry has to be updated on a container orchestration system to reflect the deployment state of the microservices. On the other hand, if the container corresponding to the microservice already exists on the edge server, a new task is spawned out of the existing container. The tasks of deploying containers and creating new tasks incur non-negligible latencies. Prefetched microservices, if not used, have no state migration cost. On the other hand, a state-aware migration has to be performed when an user actively using a microservice moves out of the service area of the server where it is hosted and the local computation state has to be sent to the server from where he is served next. For the sake of simplicity, in the following discussion, we assume it takes $25ms$ to initialize a container, $10ms$ to create a new task in an already existing container and $30ms$ to perform a state-aware migration of a container from one server to another. It may be noted that the timing values used here are just representative ones used for illustrating our problem context. We work with real world microservice timings in our experiments (Section 4.6). In accordance to our objective of proactively placing microservices, we explain in the following subsections how prefetching and proactive deployment can help mitigate some of these latencies, compared to an on-demand service placement scheme wherein service containers are provisioned only after the corresponding microservice is invoked and the container deployed for the first time.

4.2.1 On-Demand Microservice Placement

In an on-demand placement scheme, the microservices are deployed only when a user invokes the service. Microservice invocations are depicted by black rectangles and shaded circles on the trajectories of u and v respectively in Figure 4.1. At time $t = 50ms$, u invokes the “movieStreaming” service. Since ES_1, the nearest server, does not yet host the “movieStreaming” service, the corresponding container is deployed (maybe downloaded from the cloud or nearby servers), initialized and the registry is updated. The process takes a total time of $25ms$. At $t = 100ms$, v invokes the “movieStreaming” service. Since the corresponding container is already deployed on ES_1, only a new task is created incurring an initialization time of $10ms$. At time $t = 5000ms$, u invokes the “addRating” service and incurs an assumed initialization time of $25ms$. At $t = 7000ms$, u minimizes the application on his mobile. Let us assume the “addRating” service is not utilized henceforth. At $t = 8000ms$, the user relaunches the application but instead uses the “addReview” service which requires an initialization latency of $25ms$ at ES_2. Thus, the total initialization latency incurred in an on-demand scheme is $25 + 25 + 25 = 75ms$, which adds to the overall latency experienced. Table 4.1 shows the sequence of events.

4.2.2 Proactive Microservice Prefetching and Migration

To mitigate the latencies incurred when deploying services, we propose to proactively prefetch the services, considering the microservice dependency structure. Such an approach allows microservices expected to be utilized in the near future to be prefetched and deployed on the MEC server while simultaneously catering to the previously invoked service. To cater to mobility, proactively migrating already deployed services also needs to be examined as we explain later.

Proactively Prefetching Microservices

Consider the following deployment strategy. Initially, when u invokes the “movieStreaming” service, both “addRating” and “addReview” are prefetched to server ES_1. Thus, at $t = 135ms$, all three service containers have been initialized on ES_1. At $t = 5000ms$, when u invokes the “addRating” service, it no longer incurs the initialization latency of $25ms$. At $t = 8000ms$, u invokes the “addReview” service, however, it is no longer in the coverage area of ES_1. Thus, “addReview” which was initialized at ES_1, needs to be migrated to ES_2. However, since “addReview” was not used, it can either be re-initialized (if it was not already deployed at ES_1) or a new task created (if it was already deployed at ES_2) incurring a total latency of

Time t	User Action	Server-Service State
$0s$		No Services Deployed
$50ms$	$u \rightarrow$ movieStreaming	initialize movieStreaming
$75ms$		$ES_1 \rightarrow$ movieStreaming
$100ms$		$ES_1 \rightarrow$ movieStreaming, addRating
$100ms$	$v \rightarrow$ movieStreaming	$ES_1 \rightarrow$ movieStreaming, addRating initialize new task for v
$110ms$		$ES_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$135ms$		$ES_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$3000ms$	v exits movieStreaming	$ES_1 \rightarrow$ movieStreaming, addRating, addReview
$5000ms$	$u \rightarrow$ addRating	$ES_1 \rightarrow$ addRating, addReview
$7000ms$	u minimizes addRating	$ES_1 \rightarrow$ addRating, addReview
$8000ms$	$u \rightarrow$ addReview	state-aware migrate addReview
$8025ms$		$ES_2 \rightarrow$ addReview

TABLE 4.2: Proactive Placement of Microservices

$25 + 25 = 50ms$. Interleaving service prefetching and execution thus leads to a reduction in initialization latencies. However, the additional latency of $25ms$ incurred while re-initializing the “addReview” service was due to the mobility of u from ES_1’s service zone to that of ES_2. As such, a service placement scheme has to be revisited owing to user mobility. Table 4.2 summarizes the timeline of events using prefetching.

Proactive Prefetching and Migration

Let us assume that at $t = 7000ms$, instead of minimizing the application, u continues utilizing the “addRating” service till $t = 8000ms$. Since u traverses service zones while utilizing a service, the service has to be re-deployed once u is in $E2$’s service area. In such scenarios, for the “addRating” service, a state-aware migration has to be performed from ES_1 to ES_2. Additionally, since the “addReview” service had been proactively deployed on ES_1, it has to be migrated as well. Let us assume the state-aware migration is initialized at $t = 7500ms$ depicted by the light blue diamond on u’s trajectory. The migration is completed at $t = 7555ms$ since it takes $30ms$ to migrate the “addRating” service and $25ms$ to re-initialize “addReview”. The events are summarized in Table 4.3. The total initialization latency experienced by u in this case is $25ms$. However, additional *interleaved* migration latencies for “addReview” and “addRating” are incurred which are *not* perceived by the user. Note that, since the migration is performed

Time t	User Action	Server-Service State
$0s$		No Services Deployed
$50ms$	$u \rightarrow$ movieStreaming	initialize movieStreaming
$75ms$		$ES_1 \rightarrow$ movieStreaming
$100ms$		$ES_1 \rightarrow$ movieStreaming, addRating
$100ms$	$v \rightarrow$ movieStreaming	$ES_1 \rightarrow$ movieStreaming, addRating initialize new task for v
$110ms$		$ES_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$135ms$		$ES_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$3000ms$	v exits movieStreaming	$ES_1 \rightarrow$ movieStreaming, addRating, addReview
$5000ms$	$u \rightarrow$ addRating	$ES_1 \rightarrow$ addRating, addReview
$7500ms$	$u \rightarrow$ addRating	migrate addRating, addReview
$7555ms$	$u \rightarrow$ addRating	$ES_2 \rightarrow$ addRating, addReview
$8000ms$	$u \rightarrow$ addReview	$ES_2 \rightarrow$ addReview

Table 4.3: Proactive Placement + Migration of Microservices

proactively, at $t = 8000ms$, u did not have to wait to use the "addReview" microservice. The total speed-up obtained over the on-Demand placement scheme is thus 66%. For real benchmarks, depending on the sizes of the containers corresponding to the microservices and their deployment times, container initialization times can often be in the order of seconds or more, unlike milliseconds as assumed here in the representative use case. For such cases, the speed-up achieved by us can significantly impact user-perceived latencies.

The example above shows the trade-off in latency overhead using proactive prefetching versus reactive on-demand provisioning with migration of the microservices. The challenge is in determining for a given microservice, how many successor microservices to deploy proactively, and more importantly, the target edge servers to deploy them as the user moves and accesses these enroute. An overtly conservative strategy may always proactively prefetch the containers of all successor microservices, whenever any microservice is deployed. However, this may at times turn out to be wasteful in terms of resources needlessly blocked on the edge server by the prefetched containers, if these microservices are actually not invoked at that location. On the other extreme, a fully reactive policy does not help as well since such a policy would lead to initialization latency overheads for each microservice invocation. The challenge is in being able to predict the user service invocation pattern as a function of a user's mobility so that better prefetching can be carried out. Our objective here is to learn and synthesize the optimal proactive prefetch, deployment and migration schedule, given a microservice workflow.

4.3 Formal Model

In this section, we formally describe the proactive placement and migration problem and formulate an MDP model.

4.3.1 Problem Definition

The MEC system comprises a set of edge sites $E = \{E_1, E_2, ..., E_s\}$. In this work, we assume each edge site is associated with a single edge server. For each edge site, the respective edge servers are denoted as $ES = \{ES_1, ES_2, ..., ES_p\}$, where each ES_i is associated with a service radius r_i. We have a set of users $U = \{u_1, u_2, \ldots u_q\}$ and a set of applications $A = \{A_1, A_2, \ldots A_r\}$. An application $a \in A$ comprises a linear workflow of microservices $M = \{M_1, M_2, ..., M_n\}$ with special exit nodes. The order denotes the order of invocation of microservices for a given application workflow. We use a model similar to [112] where we do not consider a back-end cloud, instead consider only a set of edge servers. A location is defined as the latitude and longitude coordinates of the entity under consideration. Servers have fixed locations while users are free to move and their coordinates vary over time. We consider a discrete time model, as in [97].

Let us consider a user $u \in U$ where $u(t)$ denotes the user's current location at time slot t. We denote the set of active microservices associated with $u(t)$ as $h(t)$ and the corresponding location as $l(t)$. We assume that the set $h(t)$ can only be co-located at a single $l(t)$, i.e., all microservices in $h(t)$ will be placed by our scheme on a single edge server, as discussed later. We also assume that all latencies for deploying containers, instantiating tasks and migrating containers are strictly additive. As discussed later, the policy agent designed by our RL approach governs the placement and migration of microservices on the edge servers. At each time-slot, it observes $u(t), h(t)$ and $l(t)$, and decides on placing/migrating the relevant services $h(t+1)$, so that the user experiences the best latency values. At the beginning of each time slot, our policy agent can choose from one of the following options:

- Proactively Placing Microservices: At any location $u(t)$, when u invokes a microservice $M_i \in S$ whose successor microservices are $\{M_{i+1}, M_{i+2}, ..., M_n\}$, the agent selects the nearest server $ES_i \in E$ to deploy s_i along with j successor microservices, i.e., $\{M_{i+1}, M_{i+2}, ..., M_{i+j}\}$, where $0 \leq j \leq n - i$, and updates $h(t+1) = h(t) \cup s_i \cup M_{i+1} \cup M_{i+2} \cup ... \cup M_{i+j}$. It incurs a deployment cost $c(r)$, where c is a non-decreasing function of r, the resource requirement of the microservices to be deployed. We relate $c(r)$ with the MDP reward function as explained in Section 4.5.

- Microservice Migration: When u moves away from the service zone of the server on which $h(t)$ was deployed, the agent re-deploys the microservice if not already deployed at the edge server nearest to the user's new location and additionally performs a state-aware migration to transfer the user data associated with the microservice currently in use. Other microservices which are not currently in use are re-initialized. In this case, $h(t) = h(t+1)$, but $l(t) \neq l(t+1)$. Performing such a migration for active services incurs a cost $m(r)$, where m is a non-decreasing function of r whereas re-initialization incurs cost $c(r)$. We assume stateful migration of microservices where relocating containers between servers incurs data movement latencies.

Our objective is to determine for each time slot t, the actions of the policy agent for each $u \in U$ such that the latency incurred due to container deployment and task creation is minimized. For the sake of simplicity and ease of illustration, we first present the problem model for a single user accessing a single application. We relax these requirements later in Section 4.5 where we build on this to cater to multiple users accessing multiple services simultaneously. In the following, we use an MDP to formally model prefetching and migration.

4.4 Formal Model

Formally, we define our policy agent for proactive placement and migration as an MDP below.

Definition 4.1 [**Proactive Microservice Placement and Migration MDP:**]
The proactive placement and migration MDP is a 7-tuple $\mathcal{M}$ = (S, Λ, P, AP, L, R, init) where

- *S is a finite set of states, each state being represented by a vector $\langle service, distance\rangle$.*
- *Λ is a set of actions representing all possibilities of proactively prefetching microservices.*
- *$P : S \times \Lambda \times S \to [0,1]$ is the transition probability function such that for all states $s \in S$ and actions $\lambda \in \Lambda$: $\sum_{s \in S} P(s, \lambda, s') = 1$.*
- *AP is a set of Atomic Propositions corresponding to microservices prefetched and distance (as elaborated in the following discussion).*
- *$L : S \to 2^{AP}$ is an AP labelling function, labelling states with the APs.*
- *$R : S \to \mathbb{R}$ is the reward associated with each state $s \in S$.*
- *init is the initial state.*

In the following subsections, we discuss the state representation and transition representation of the MDP in detail.

4.4.1 State Representation of MDP

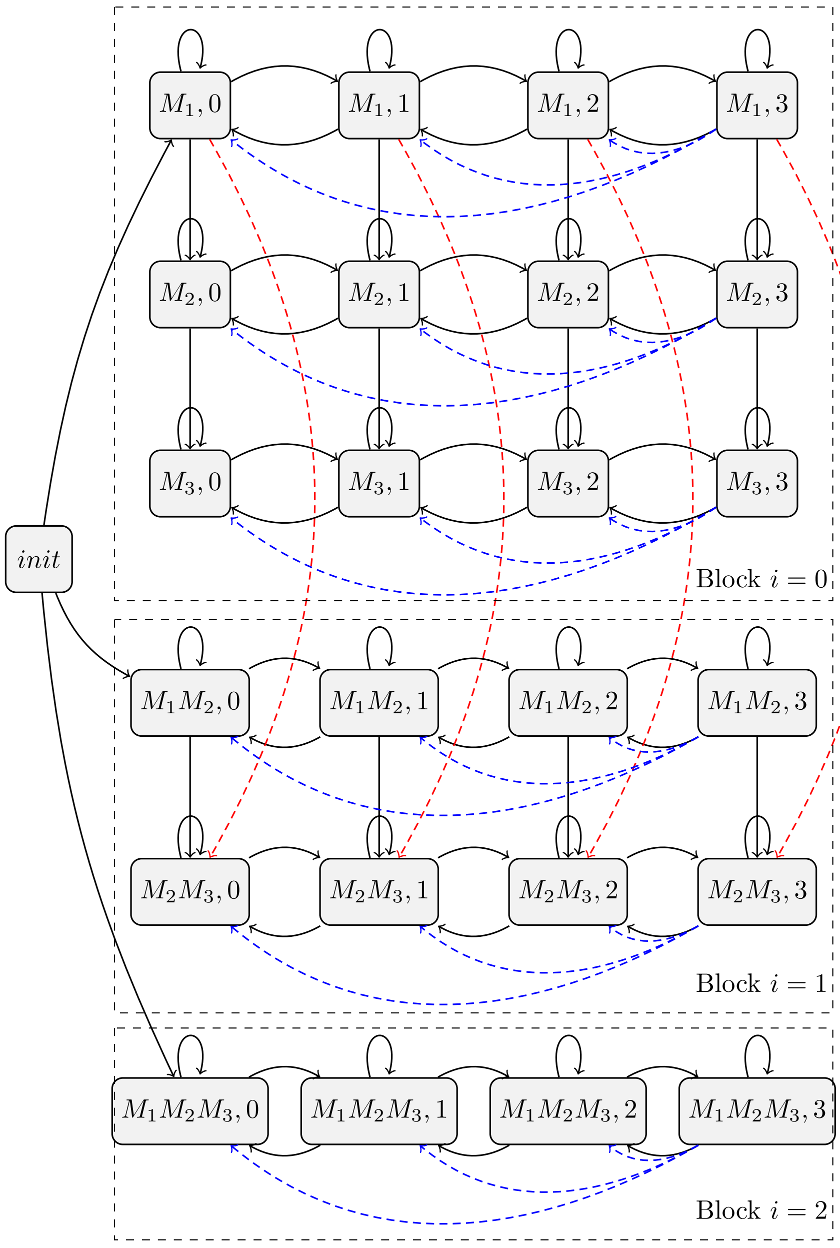


FIGURE 4.3: Proactive Service Placement and Migration MDP

Each state of the MDP for a user u is represented by a vector $\langle service, distance \rangle$. In each state, *service* represents $h(t)$, while *distance* represents the distance between the location $u(t)$ of u and the location of $h(t)$, i.e., $l(t)$. We represent the distance as an abstract measure similar to [97]. In the MDP, *distance* is a mapping from a concrete measure such as the Euclidean or Manhattan distance to the abstract measure. The server ES_i corresponding to $l(t)$ is associated with a maximum service area depicted by the radius r_i from its location. Hence, there is an upper bound on the *distance* representation in the MDP. The upper bound denotes the distance between the location of the server and a coordinate located on the circumference induced by the service radius r_i of the server ES_i. Further, since each server can have a different service radius, the upper bound distance is normalized in the range $[0, k]$ where k is a user defined parameter. However, since such an interval is continuous, the interval $[0, k]$ is discretized at intervals of 1. Hence, all possible values of distances are $0, 1, \ldots, k$. The concrete distance from $u(t)$ to $l(t)$ is thus mapped to the discretized interval distance set as follows: distance measures between $u(t)$ and $l(t)$ in the continuous interval $[0, 1)$ are mapped to $k = 0$, distances in the continuous interval $[1, 2)$ are mapped to $k = 1$ and so forth, where $[0, 1)$ denotes the continuous interval inclusive of the lower bound 0 and exclusive of 1. As such, $distance = k$ denotes the scenarios when the distance between $u(t)$ and $l(t)$ exceeds k. The $\langle service, distance\rangle$ vector thus, uniquely identifies the microservices which have been proactively prefetched and the distance between a user u and the server $ES_i \in ES$ where the prefetched microservices are hosted. The MDP structure embodies all possibilities for prefetching discussed earlier, considering a given microservice workflow. In the following, we use our example application with 3 main constituent microservices to illustrate the MDP construction for user u.

Example 4.4.1. Figure 4.3 depicts the MDP for the Movie Streaming Application accessed by user u in Section 4.2. The application comprises 3 microservices "movieStreaming", "addRating" and "addReview", represented as M_1, M_2 and M_3 respectively. The state space of the MDP represents all possible scenarios of microservice deployments and the corresponding location of the user u within an edge site. The transitions represent all possible control flows between consecutive microservices as well as all subsequent prefetching possibilities. The "movieStreaming" microservice is initialized on server $ES_1 \in E$ upon invocation of the application by u. We consider the Euclidean distance measure as an illustration. The MDP assumes $k = 3$. Let us suppose the Euclidean distance between the location of u and ES_1 evaluates to a value between 0 and 1. The state $\langle M_1, 0\rangle$ denotes such a scenario. Thus, the distance identifier of a state vector abstractly represents a concrete distance interval. Similarly, $\langle M_1, 1\rangle$ is the scenario when the distance between u and ES_1 is between 1 and 2. Since $k = 3$, the state $\langle M_1, 3\rangle$ denotes the scenario when u moves to a location when the distance between u and ES_1 exceeds 3. Such states correspond to only a single service being deployed at a discrete time-point. The state $\langle (M_1, M_2, M_3), 0\rangle$, on the other hand, exhibits all three microservices being deployed on a server at a distance between 0 and 1 from u with M_2 and M_3 being prefetched in addition to M_1. ■

4.4.2 Proactively Prefetching Microservices

The MDP can be viewed as comprising several blocks. Each block corresponds to prefetching i ($0 \leq i < n$) services corresponding to the linear workflow, where n is the number of microservices in the application. The case $i = 0$ corresponds to reactive deployment, where only upon invocation, the respective service is initialized. The case $i = n - 1$, on the other hand, corresponds to an overly conservative strategy where all services comprising the workflow are pre-fetched upon application initialization. When the value of i ranges between 1 and $n - 1$, i consecutive services are prefetched.

Example 4.4.2. The MDP constructed in Figure 4.3 comprises 3 blocks, each depicted with a dashed rectangle. The first block corresponds to reactive microservice deployment while the remaining blocks represent prefetched deployment. The state $\langle (M_1, M_2), 0\rangle$, within the block $i = 1$ denotes the scenario where microservices M_1 and M_2 are prefetched while the state $\langle (M_1, M_2, M_3), 0\rangle$ within the block $i = 2$ denotes the scenario where the microservices M_1 , M_2 and M_3 are prefetched. ■

4.4.3 Transition Representation of MDP

Transitions from the *init* state denote the number of microservices to initially prefetch with one transition to each block.

Example 4.4.3. The *init* state has outgoing transitions to the states $\langle M_1, 0\rangle$, $\langle (M_1, M_2), 0\rangle$ and $\langle (M_1, M_2, M_3), 0\rangle$. Each such state denotes the number of microservices proactively prefetched. When the application is invoked using M_1, a transition from *init* to $\langle M_1, 0\rangle$ denotes the scenario where only the microservice M_1 is deployed on the server with no additional pre-fetching. Similary, the transition to $\langle (M_1, M_2), 0\rangle$ denotes the scenario where M_2 is pre-fetched on the server along with M_1. ■

Other transitions occur when the state of u changes and can be broadly classified into two types: transitions within a block and transitions between blocks.

Intra-Block Transitions: Transitions within a block occur only when a user moves from one location to another or when there is a transfer of control from one microservice to its subsequent microservice in the application workflow.

Example 4.4.4. Let us suppose the initial Euclidean distance between u and ES_1 was between 0 and 1. Such a scenario is denoted in the MDP by the state $\langle M_1, 0\rangle$. Along the course of u's path, let us suppose the Euclidean distance measure at some time-point exceeds 1. This change in u's location is represented by the transition to $\langle M_1, 0\rangle$. Continuing along its trajectory, as long as the Euclidean distance between u and ES_1 lies between 0 and 1, it remains in the state $\langle M_1, 0\rangle$ denoted by the self transition. The transition from $\langle M_1, 0\rangle$ to $\langle M_2, 0\rangle$ denotes the transfer of flow of control in the microservices workflow from M_1 to M_2 while the distance between u and the server where M_2 is deployed remains within 1. Note that however, transitions denoting changes to both distance and service invocation trajectory can not happen. For example, there is no transition from $\langle M_1, 0\rangle$ to $\langle M_2, 1\rangle$. This is because prefetching services is carried out with respect to the current relative locations of the server and the user. A simultaneous change in both is only accounted for by first updating the location followed by the service invocation. ■

Inter-Block Transitions: Transitions between blocks represent the possibility of proactively prefetching variable number of microservices. Consider a state S of the MDP where the *service* component of the state vector comprises $(M_m, \ldots, M_n)$. In the event of the transfer of flow of control of microservices from M_m to M_{m+1}, outgoing transitions from S portray choices of the number of proactively prefetched microservices by transitions to all states in the MDP whose identifier begins with M_{m+1}. Such transitions are represented in Figure 4.3 by red curved dashed lines.

Example 4.4.5. The outgoing transition from state $\langle M_1, 0\rangle$ to state $\langle (M_2, M_3), 0\rangle$ of block $i = 1$ denotes the situation when u experiences a flow of control transfer from M_1 to M_2, and both M_2 and its successor M_3 are prefetched to the server. Note that the other choice of deploying M_2 only is already covered in block $i = 0$. Thus, the transition from M_1, M_2 in block $i = 1$ to M_2 block $i = 0$ depicts the scenario when u experiences the same flow of control from M_1 to M_2, but the agent decides not to proactively fetch any other service. However, note that transitions such as those from $\langle M_1, 0\rangle$ to $\langle (M_1, M_2, M_3), 0\rangle$ are not possible since the latter depicts prefetching all three services upon invocation of M_1 while the former denotes M_1's deployment with no other service prefetched. Also note that transitions such as those from $\langle (M_2, M_3), 0\rangle$ to $\langle M_3, 0\rangle$ are not possible. In state $\langle (M_2, M_3), 0\rangle$, both M_2 and M_3 have already been prefetched. Such a transition would only depict a flow of control from M_2 to M_3 which does not necessitate a further prefetching decision. ■

4.4.4 Migration of Microservices

Migrations occur when u moves from one service zone to another. In such a scenario, if the new service zone has only one server associated with it, the microservices are migrated to that server. Otherwise, if multiple choices of servers are available, the nearest server is selected. We consider server capacity constraints as explained later in Section 4.5.2. When u crosses the boundaries of service zones, a migration is triggered. Such migrations are represented in the MDP by transitions from states whose distance vector component is k to states whose distance vector component is $\{0, 1, \ldots, k-1\}$. We assume that the MEC servers are distributed such that each area is in the coverage of at least one MEC server, hence a target server always exists.

Example 4.4.6. Let us assume that along u's trajectory, at some time-point, the Euclidean distance between u and ES_1 is between 2 and 3 denoted by the state $\langle M_1, 2\rangle$. When u moves further away from the server, exceeding ES_1's service radius, the state of the MDP is updated to $\langle M_1, 3\rangle$. M_1 is then migrated to the nearest server, say ES_2. In such a scenario, the Euclidean distance between ES_2 and u is re-calculated and the corresponding new abstract distance is represented in the MDP by blue dashed transitions to $\langle M_1, 0\rangle, \langle M_1, 1\rangle, \langle M_1, 2\rangle$ according to the re-calculated distance and mapped appropriately using the abstract distance representation. ■

Additionally, since users can exit the application at any stage, we add an extra state, *exit* to the MDP demarcating that the user has exited the application. From all states excluding the *init* state, transitions are drawn to this state signifying the event of an application closure. The *exit* state and the corresponding transitions are not shown in Figure 4.3 for brevity. Further, we do not require any explicit encoding to denote the minimized state of an application since prefetched services are retained on MEC servers while minimized and evicted only upon application exit.

The MDP built above embodies the underlying solution space for our problem context, accounting for all prefetch and deployment possibilities. While on one hand, the states represent the different service user deployments, the transitions represent the corresponding possibilities, induced by user movement and possible service invocations. Once the MDP is built, we now proceed to determine the rewards associated with our MDP, based on the movement and service invocation patterns of users. This is helpful for deciding the proactive prefetching strategy, i.e. for which microservice, how many successors to prefetch at which location and deploy on which edge server. We formulate this problem as a Reinforcement Learning (RL) problem where the agent explores interactions with the environment to learn the rewards for the best strategy.

4.5 Reinforcement Learning Solution

We use the Dyna-Q [34] RL algorithm, a combination of model based RL and Q-Learning. The Dyna-Q Algorithm is summarized in Algorithm 5 and explained in detail in Chapter 2. Q-learning essentially estimates the optimal Q-function, Q, by its sample averages. The Q-Table is initialized with the state space of the MDP described earlier and the possible proactive deployment choices. In this chapter, we consider the standard simple ϵ-greedy action selection method: at any decision step i, with probability ϵ, Q-learning chooses a random action to *improve* its knowledge of the application, whereas, with probability $1-\epsilon$, it chooses the action greedily by *exploiting* its knowledge about the application, i.e., $\lambda = \text{argmax}_\lambda\ Q(s,\lambda)$. Such actions correspond to the different choices of proactive microservice deployment. Dyna-Q then proceeds by simulating the real-time experiences of proactively deploying microservices where ζ is the learning rate.

Algorithm 5: Dyna-Q

```
1  Initialize $Q(S,\Lambda)$ and $Model(S,\Lambda)$, $\forall s \in S, \forall \lambda \in \Lambda$
2  while true do
3      $s \leftarrow$ observe the application state
4      $\lambda \leftarrow \epsilon$-greedy$(s,q)$
5      Observe the next state $s'$ and the reward obtained
6      $Q(s,\lambda) \leftarrow Q(s,\lambda) + \zeta\,[r + \gamma\ \text{argmax}_\lambda[Q(s',\lambda)] - Q(s,\lambda)]$
7      Model $(s,\lambda) \leftarrow r, s'$
8      for $i = 0 \ldots n$ do
9          $s \leftarrow$ random state previously observed
10         $\lambda \leftarrow$ random action previously taken in $s$
11         $r, s' \leftarrow Model(s,\lambda)$
12         $Q(s,\lambda) \leftarrow Q(s,\lambda) + \zeta\,[r + \gamma\ \text{argmax}_\lambda[Q(s',\lambda)] - Q(s,\lambda)]$
```

In our context, in a real-world scenario, multiple users access multiple applications simultaneously. Each application A_i is thus associated with its own MDP $\mathcal{M}_i$ as described in Section 4.4.1. Each $\mathcal{M}_i$ has a corresponding reward r_i whose initial value is set to 0. When a user u_j invokes an application A_i, the corresponding MDP $\mathcal{M}_i$ is assigned to the user. This is used with u_j and $\mathcal{M}_i$ to execute the required prefetching / migration actions. When the agent executes an action, it receives rewards from the environment. Thus, rewards are assigned whenever there exists a transition to denote state change. The reward function denoted by R is a weighted combination of resources consumed by prefetched microservices actually utilized and prefetched microservices not invoked by the user.

$$R = \sum_{\mu \in \mu_{used}} [\mu * c(\mu_{resources})] - \sum_{\mu \in \mu_{unused}} [\mu * c(\mu_{resources})] \tag{4.1}$$

μ_{used} and μ_{unused} are sets of indicator variables representing the set of prefetched services which have been invoked and not invoked respectively while $c(\mu_{resources})$ represents the resource costs of microservice μ according to a cost function $c(r)$. This reward is calculated whenever the user invokes a microservice in the application workflow following which the Q-values are updated. Additionally, the Dyna-Q algorithm simulates previous real-world interactions (Lines 8-12). A positive reward is assigned for services which are prefetched and utilized by the user while a negative reward is assigned to unused services. Such a reward function embodies migration decisions as well. Negative rewards signal the agent to lean towards prefetching a lower number of services thereby reducing migration costs while advocating a reduction in the number of unnecessary migrations. As such, for rewards corresponding to migrations, the same reward function is used, with the migration cost function $m(r)$ instead of $c(r)$.

The formulation allows us to characterize user service invocation patterns as a function of mobility using a distance based MDP which is space efficient as well. However, it neither effectively quantifies the prefetch policy influenced by the network load characteristics nor does it cater to capacity constraints of servers. To characterize traffic load distribution and capacity constraints, we propose a heuristic in the following discussion.

4.5.1 Catering to traffic variation

The number of users invoking different applications may actually vary over time. In such a scenario, the number of users can play a crucial role in determining how the RL agent is trained. If an agent receives a high reward for an action in a low service request traffic load environment, i.e., when the number of application users is low, it may end up receiving a low reward for the same action in a high service request traffic environment since a single application proactively prefetching multiple microservices can lead to clogging of resources for other users depending on the arrival of microservice invocations. Such variance in rewards can confuse the RL agent [113]. To overcome this difficulty, instead of assigning a single MDP $\mathcal{M}_i$ to A_i, we associate three MDPs $\mathcal{M}_i^{low}$, $\mathcal{M}_i^{med}$ and $\mathcal{M}_i^{high}$ to each A_i denoting low, medium and high traffic loads respectively. When u_j invokes A_i, depending on the current load distribution in the k-hop neighbourhood of u_j, the corresponding MDP is assigned to u_j. Such a mechanism described in Algorithm 6 allows us to separately characterize traffic workloads and enables the RL agents to effectively use the exploitation phases to learn traffic-aware policies. Note that we utilize a centralized controller where the MDP and DynaQ-Tabulation can be stored on disk and instantiated upon the invocation of an application corresponding to a user.

4.5.2 Catering to Capacity Constraints

The MDP discussed in Section 4.4.1, only comprises information pertaining to the location of users and services currently in operation. We do not encode information pertaining to capacity of servers or request-server bindings in the MDP. In a realistic setting, MEC servers do not possess infinite storage or computing capabilities, and therefore, capacity constraints need to

Algorithm 6: Proactive Prefetching and Migration

```
Input   : $low, medium, l, servers$
1  Initialize K-D Tree with $servers$
2  foreach $timepoint\ t$ do
3      foreach $user\ u$ do
4          $curr_{loc} \leftarrow$ updated location of $u$
5          $server_l \leftarrow$ location of server assigned to $u$
6          $d \leftarrow$ distance between $server_l$ and $curr_{loc}$
7          if $d \geq server_l.coverage$ then
8              $newserver \leftarrow$ query K-D tree for location of nearest server to $u$
9              $server_l \leftarrow newserver$
10             $nd \leftarrow$ distance between $newserver$ and $u$
11             update MDP state according to $nd$ and migrate microservices greedily
12         else
13             $nd \leftarrow$ new distance between $server_l$ and $u$
14             update MDP state according to $nd$
15         if current application/service status is different from previous time-slot then
16             if $u$ has invoked $A_i$ then
17                 $ld \leftarrow$ load in $l$-hop neighbourhood of $u$
18                 if $ld \leq low$ then
19                     $u.\mathcal{M} \leftarrow \mathcal{M}_i^{low}$
20                 else if $ld \leq med$ then
21                     $u.\mathcal{M} \leftarrow \mathcal{M}_i^{med}$
22                 else
23                     $u.\mathcal{M} \leftarrow \mathcal{M}_i^{high}$
24             $\lambda \leftarrow$ action selected with Dyna-Q for $u$
25             if $\lambda$ exceeds server capacities then
26                 $M_i, M_j, \ldots M_n \leftarrow$ prefetched services
27                 foreach $M \in M_i, M_j, \ldots M_n$ in order do
28                     allocate $M$
29                     if allocate $M$ is $NULL$ then
30                         $M_i, M_j, \ldots M_k \leftarrow$ services allocated successfully
31             set MDP to state corresponding to $M_i, M_j, \ldots M_k$ successfully allocated
32             update $Q$ with reward for action $\lambda$ using Equation 2 for $u$'s associated MDP
33             upon application exit, update corresponding MDP for $A_i$ with highest rewards
```

be adhered to. We address such capacity constraints heuristically without the requirement of having to encode any additional information in the MDP states. Initially, the MDP agent selects an action corresponding to deploying $M_1, M_2, \ldots M_n$ microservices. If all such services can be deployed without violating capacity constraints, the agent proceeds to deploying all the microservices $M_1, M_2, \ldots M_n$. On the other hand, with capacity constraints not withholding, the algorithm greedily deploys the containers pertaining to the prefetched services. This deployment is carried out in sequence of the prefetched microservices of the workflow until the capacity constraints are exhausted resulting in a subset of the prefetched microservices being deployed. It then updates the MDP state corresponding to the services deployed greedily. Allocation upon migration is carried out in a similar manner.

Algorithm 6 initializes a K-D Tree [114] with the location of the MEC servers (Line 1). This allows efficient queries to locate the nearest server for users. It proceeds to update the locations of the users in the current slot (Line 4) and identifies users which have moved out of the service zones of the servers to which they were assigned. For users which have indeed moved out, the nearest server in the current location is identified by querying into the K-D tree (Line 8). Accordingly, the MDP is updated to the value of the normalized distance from the currently assigned server and the updated location of the user (Lines 7-14). It proceeds to check the service usage status of the users in the current time slot (Line 15) and calls the Dyna-Q algorithm. Depending on the action selected by the Dyna-Q algorithm and the subsequent greedy placement (Lines 25-30), the state of the MDP and the Q-values are updated with the generated rewards (Line 31-32). In the scenarios that the prefetched microservices can not be accommodated on the server, the algorithm greedily selects the set of microservices that can be accommodated in the workflow sequence of the application, as explained above. Once it exhausts the server capacities (Line 29), it proceeds to set the MDP state to reflect the microservices successfully allocated greedily (Line 31). Once a user exits the application, the MDP is updated accordingly (Line 32).

4.6 Experimental Evaluation

We perform extensive experiments to show the efficacy of our approach, and compare its performance against a) on-demand placement, and b) MCAPP-IM [109], an algorithm for placement of applications with multiple components. In the following, we describe our experimental setup and the results.

4.6.1 Experimental Setup

In the following, we describe how we generate the problem instances for our simulation experiments and describe the experimental setup.

4.6.1.1 MEC Server Locations and User Trajectories

We consider a discrete time slotted system in which the locations of users in the network may change from one time slot to another. We use the San Francisco taxi dataset and the 'Existing Commercial Wireless Telecommunication Services Facilities in the San Francisco' dataset with the setup described in Section 2.6.2. We assume k as 3 for our experiments as well unless specified otherwise. The generated coverage areas of all servers are normalized in $[0, 3]$ when translated to the MDP representation.

4.6.1.2 Service Invocations

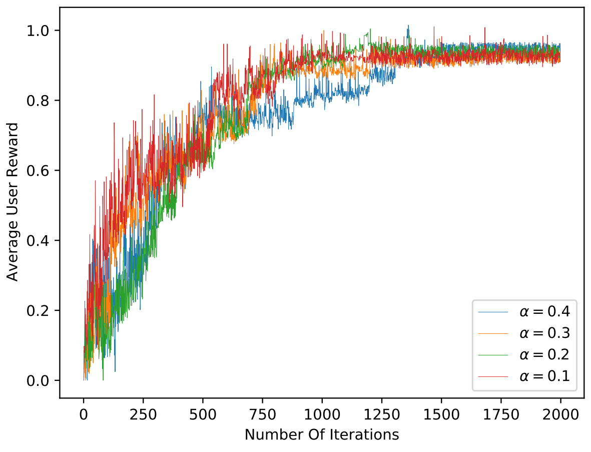
We use microservices from the 'Media Microservices' application of the 'DeathStarBench' benchmark suite [48]. We use the size of the container of each microservice as its representative resource requirement. We fetch the corresponding containers from Docker Hub [49] initially. We then note the starting times of each containerized microservice by invoking a fresh docker container start after stopping all running containers. We use these times as the deployment times of containers on MEC servers. Further, we generate running times in the range ($1/3 \times$ starting-time $\pm \lambda$) to simulate representative times of creating tasks from already existing containers. We use these values as representative times since the DeathStarBench benchmark uses a composition driven approach where multiple microservices are used to execute a task. In the DeathStarBench benchmark, each task comprises a composite workflow unlike our model where we assume a single container being associated with a task. 1/3 is chosen since creation of tasks involves lower computation times as compared to deploying dedicated containers and λ is assigned a random value between 0 and 0.05 to simulate random runtime deviations. To simulate migration times, we add these task creation times to the container deployment time. These values are used as $c(r)$ and $m(r)$ for the reward function. To each taxi trajectory obtained from the dataset, we assign such service invocations randomly at different discrete time slots. We assign the invocations considering the distance between the location of the taxi in the previous time slot and the current time slot. If the distance exceeds a threshold value, we treat such a slot as a fresh invocation of an application, since we assume that the applications exit when the user leaves the area or closes the application. Additionally, while extracting the trajectories from the original taxi dataset, we only use a portion of the San Francisco area. As a consequence, there are several time slots, where the location of the taxis incur abrupt changes in latitude and longitude coordinates. We treat such changes in coordinates also as fresh invocations of an application. In the event that the distance does not exceed this threshold city area, we randomly generate invocation/minimization decisions along the linear workflow structure of the application. Upon invocation of the last microservice in the workflow, a fresh invocation of a random application is considered.

4.6.2 Results and Discussion

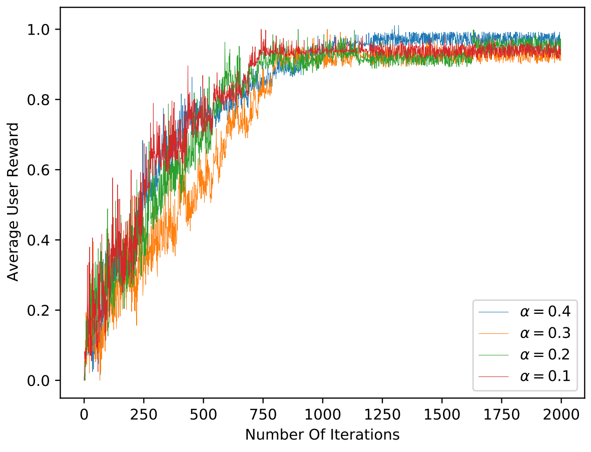
We compare the performance of our approach with that of on-demand reactive service provisioning to demonstrate the impact of proactively prefetching services on user experienced latency. Additionally, we compare with the MCAPP-IM algorithm [109]. The MCAPP-IM algorithm formulates the problem as an online bipartite matching problem supplemented with a greedy local search technique between application components and edge servers. However, they do not consider proactively deploying any of the application components. MCAPP-IM runs at each time slot and the result of the matching thus obtained is the service-server binding. The MCAPP-IM algorithm does not consider coverage area zones of MEC servers. All experiments are performed in Python 3.7 with the K-D tree implementation of the SciPy library on an Intel Core i5 8250U processor and 16 GB of RAM. We set the value of the exploration parameter ϵ of Dyna-Q to 0.2.

Varying the Number of Users

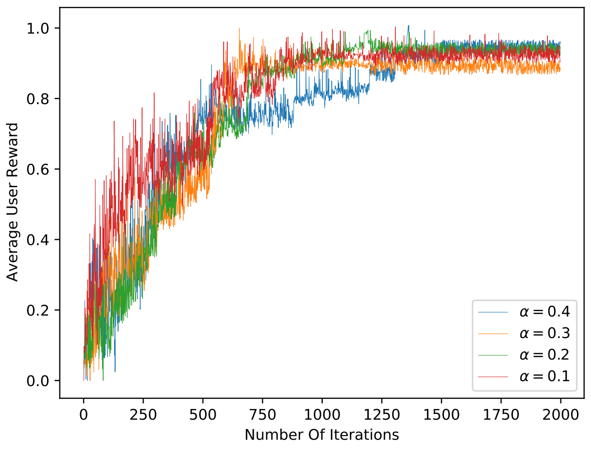
We vary the total number of users from 50 to 300 at an interval of 50. For each scenario, we additionally vary the learning rate ζ as $\{0.1, 0.2, 0.3, 0.4\}$. In Figure 4.4, we plot the average reward of the policy agent against the number of iterations where each iteration refers to one discrete time-slot where the average reward value is normalized between 0 and 1. As can be inferred from the figure, different learning rates produce a variation in the rewards accumulated by our agent. With a higher learning rate, a wider variation in reward accumulation is observed in general, since a higher learning rate corresponds to a greater weightage in Q-value updation in each iteration. It is however interesting to note that for a high number of users, specifically in Figure 4.4f, the deviations obtained are rather minor as compared to others. This is due to the fact that during high request traffic environment, the MEC servers are resource constrained and hence lean towards more reactive deployments, justifying our design objective.



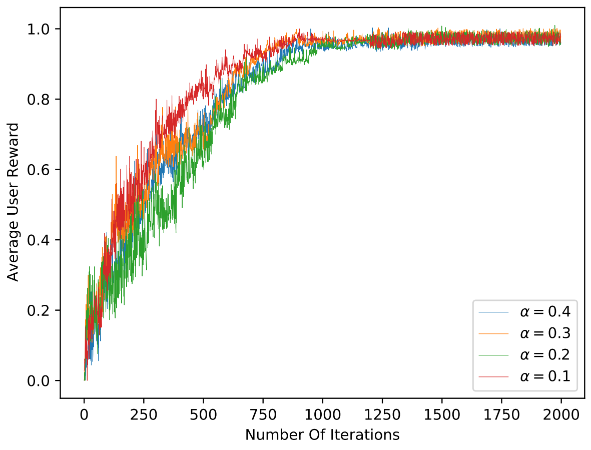
(a) Number of Users = 50



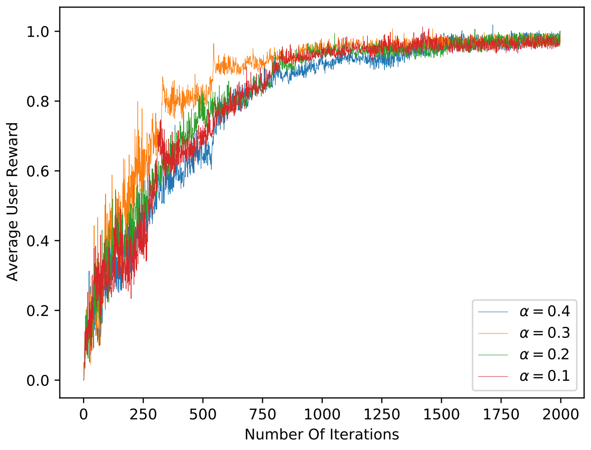
(b) Number of Users = 100



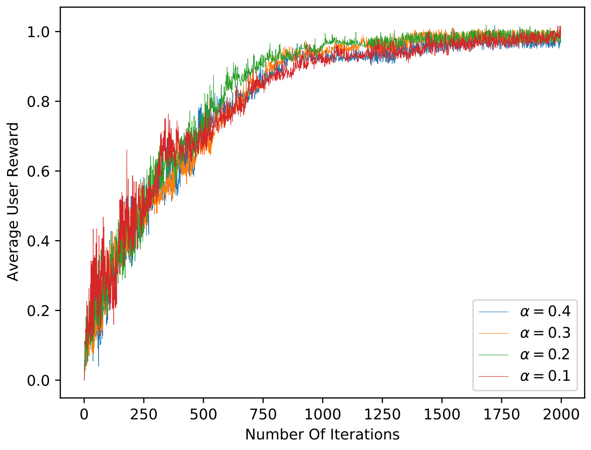
(c) Number of Users = 150



(d) Number of Users = 200

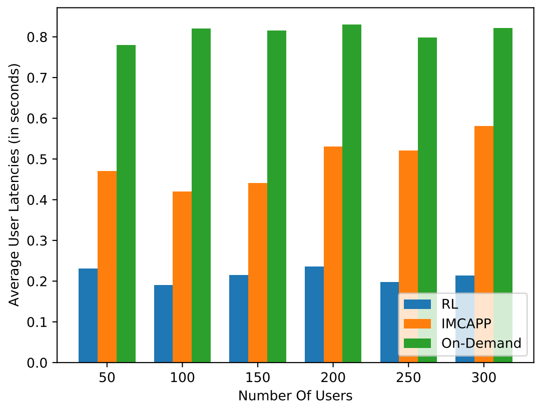


(e) Number of Users = 250

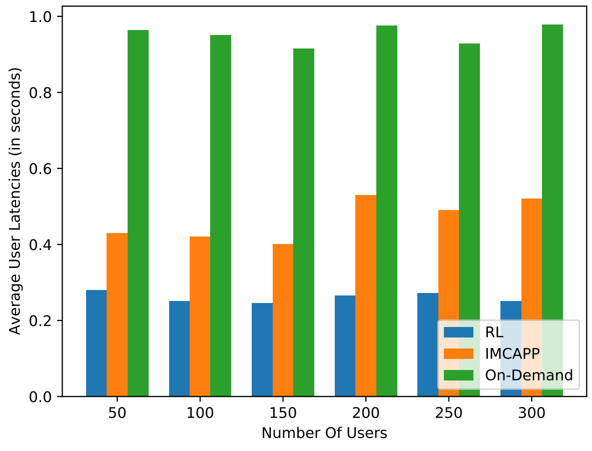


(f) Number of Users = 300

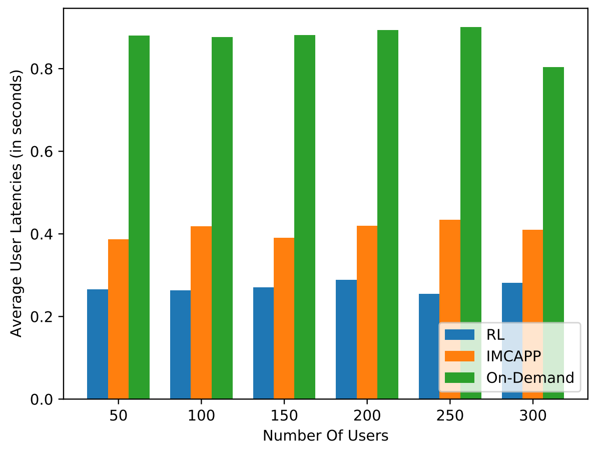
FIGURE 4.4: Average Reward Accumulation with Variable Number of Users



(a) $n = 4$



(b) $n = 8$

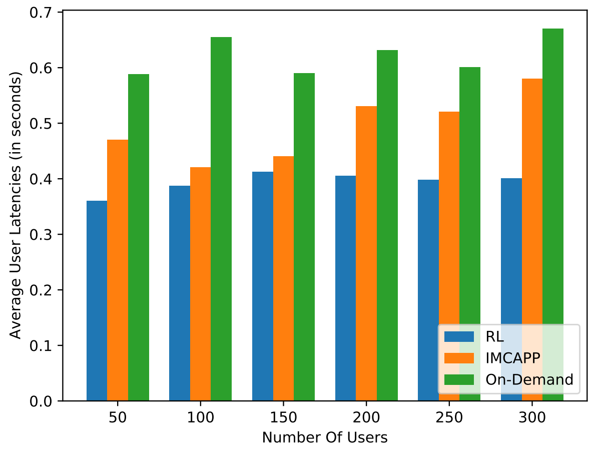


(c) $n = 12$

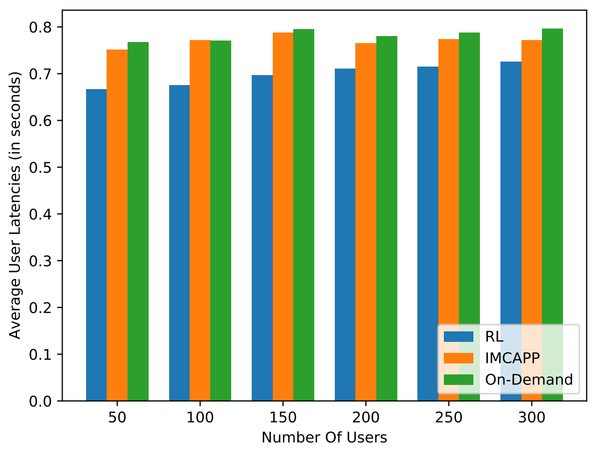
FIGURE 4.5: Average User Latency with Varying Number of Application Microservices

Varying the Number of Microservices

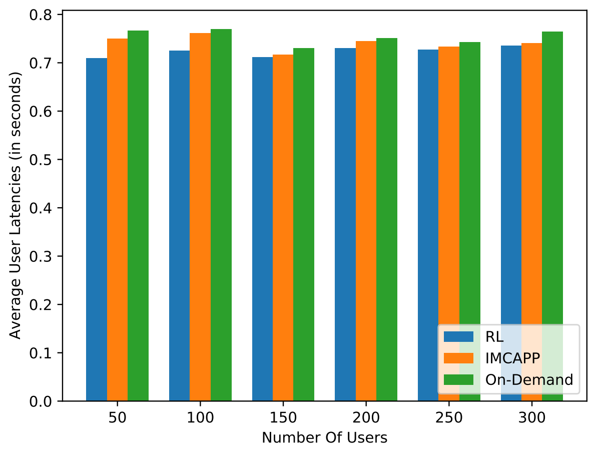
In Figure 4.5, we plot the average user latencies as we vary the number of microservices in the application workflows. We consider representative applications involving $4, 8$ and 12 microservices (denoted as n in the figure), from the 'DeathStarBench' benchmark suite [48]. We measure the performance of the algorithms as we vary applications with the aforementioned number of microservices. There is no definite increase/decrease pattern with latencies as the number of users are increased. This is expected since with adequate availability of server resources, a higher number of users does not necessarily lead to greater contention. Proactively deploying microservices leads to an overall benefit of the latency perceived by the user in all scenarios as observed. Since MCAPP-IM does not involve server coverage areas, it incurs a lower latency



(a) Server Resources = 130%



(b) Server Resources = 100%



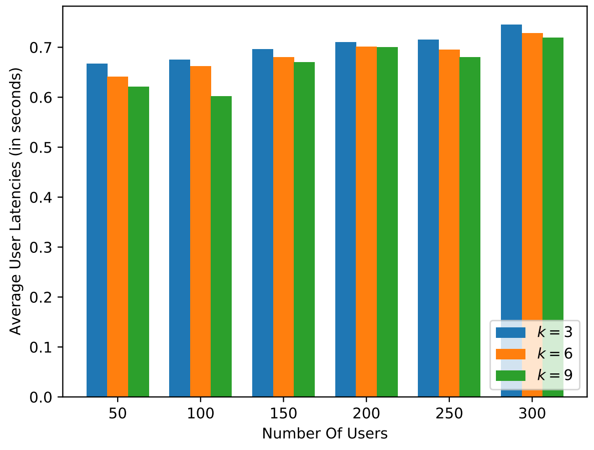
(c) Server Resources = 65%

FIGURE 4.6: Average User latencies for Varying Request Traffic Distributions

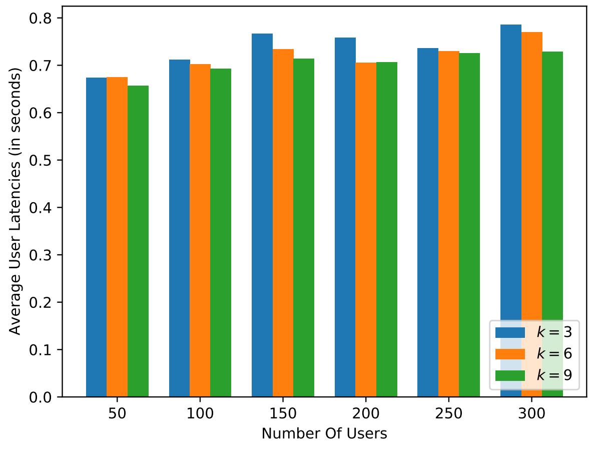
as compared to the On-Demand scheme. However, our RL based algorithm being specifically catered to proactive deployment, is able to outperform MCAPP-IM in terms of average latency experienced by users.

Varying Server Resource Availability

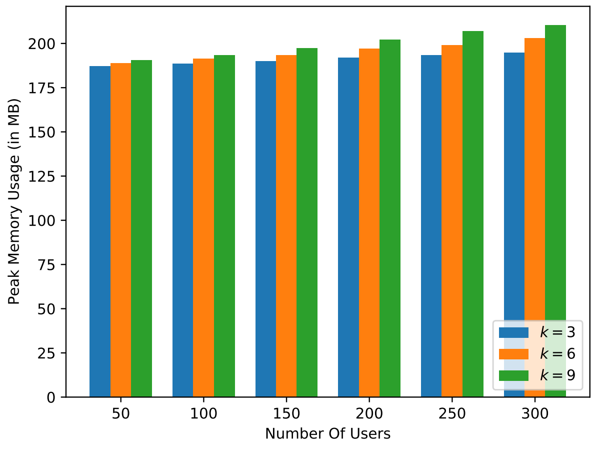
We next analyze the effect of the service request traffic load on the performance of the various algorithms. We vary the number of available MEC servers while keeping the number of users fixed to simulate availability of server resources. We consider scenarios where we fix the total server resources at 130%, 100% and 65% of the required total resource consumption of the users. A server resource percentage of 100% denotes the scenario in which the resource availability of



(a) Varying k for $n = 4$



(b) Varying k for $n = 8$



(c) Memory Usage for Varying k

FIGURE 4.7: Effect of Varying k on Latency and Memory Usage

the server can cater to exactly the number of users fixed. In Figure 4.6, we plot the average latencies of the three algorithms in each scenario. With high availability of server resources, our algorithm obtains lower average user latencies as compared to MCAPP-IM. The average improvement over MCAPP-IM is around 44%. However, as the resource availability of servers is decreased, with high traffic loads, the benefits of proactive deployment are far lower, at an average of 11%, as observed in Figures 4.6a, 4.6b and 4.6c. This is because in such a resource constrained environment, the agent favors lesser proactive prefetching of microservices. Such a scenario, for an application which comprises 3 microservices, as in the example in Section 4.2, would correspond to the MDP in Section 4.4.1 executing most of the transitions in Block $i = 0$ prefetching a small number of microservices. This supports our intuition and justifies our design objectives as well.

Varying the Parameter k

We now analyze the effect of k on the performance of our RL approach. k is a user defined parameter which specifies the discretization constant for coverage areas of each server. We experiment with values of $k = \{3, 6, 9\}$. As the value of k is increased, the size of the MDP increases. As a result, the memory consumption increases which is validated in Figure 4.7c. Additionally, we vary k in the same range for another application which comprises 8 microservices instead of 4. Figure 4.7b depicts that there is no definite increase/decrease pattern with respect to varying n and the number of users as observed previously when we study the impact of n on latencies. It is however interesting to note that, as the value of k is increased, there is an improvement in latency for several scenarios in Figures 4.7a and 4.7b. Increasing the value of k, allows us to represent the discretized coverage areas of servers more precisely and hence, the agent can make decisions more accurately. Thus, varying k can have an impact on the overall latency incurred. However, larger values of k incur a greater cost of representation in memory thus presenting a trade-off. Further, in Figure 4.7a, only a marginal improvement is observed when k is varied for the scenario with 200 users. Such scenarios can indeed happen if the dataset does not incorporate the entire action space of the agent thereby rendering some actions unexplored.

4.7 Conclusion

In this chapter, we propose a learning based mechanism for proactive deployment of microservices on edge servers considering microservice application structures. For the sake of simplicity, we consider a linear workflow microservice, examples of which are abundant in practice. Even for such simple workflow structures, the proactive placement strategy and its benefits have not been addressed in literature, to the best of our knowledge. The linear structure helps us contain the possibilities we need to examine in the solution space, and helps us build the foundation of our learning based solution framework. Experimental results on real datasets are encouraging, and demonstrate the latency improvements that our scheme leads to. In the next chapter, we propose an auto-scaling policy for MEC to aid load balancing and fault tolerance in scenarios where each edge site is associated with multiple edge servers.

Chapter 5

Horizontal Auto-Scaling for Applications

5.1 Introduction

In an MEC system, application services are associated with thresholds on maximum latency, which if exceeded, results in aggravated Quality-of-Experience (QoE) to end-users [3, 13]. As users move about, multiple service requests from such users provisioned by a single edge server can lead to high resource contention culminating in latency threshold violations. To mitigate these effects, dynamic auto-scaling policies are advocated which seamlessly adapt to request load variations by automatically provisioning and de-provisioning resources [10, 12, 13]. As discussed earlier, when a service invocation request is triggered, the service request can either be allocated by spawning a new thread in an *already existing container* or provisioned by spawning a new container instance on a *different edge server*. Spawning a new thread in an already existing container can lead to added resource contention subsequently resulting in latency threshold violations. Spawning a new container instance at another server, on the other hand, can reduce contention amongst resources but is accompanied with the cost of utilizing additional resources

This work is published as:

- Kaustabha Ray and Ansuman Banerjee, "Horizontal Auto-Scaling for Multi-Access Edge Computing Using Safe Reinforcement Learning", In ACM Transactions on Embedded Computing Systems 20, no. 6 (2021): 1-33.

such as memory and CPU there. Deciding when to spawn a new container instance is therefore a challenging task. Additionally, multiple heterogeneous applications running simultaneously on an edge server can further add to the complication where some applications are predominantly CPU affine while some other applications make heavy use of the GPU. Our main objective is thus to design a horizontal auto-scaling policy in the MEC context which automatically determines when to *retain/add/remove* container instances of an application to ensure that the probability of adherence to application latency requirements is maximized or equivalently, the probability of users incurring latency violations is minimized.

Since we deal with the problem of minimizing the probability of incurring latency violations specified by application specific latency thresholds, conventional optimization techniques do not apply directly. Additionally, the latencies incurred by the users vary at runtime depending on several aspects such as the heterogeneous nature of applications deployed on the same server, concurrent service invocations, resource contention and so forth. As a consequence, analytically modeling such uncertain characteristics is a difficult task [88]. Conventional optimization approaches are either static wherein they rely on such detailed analytical models or incorporate uncertainty by assuming some distributions of the parameters involved. Learning based auto-scaling policies, on the other hand, overcome this difficulty without the requirement of modeling individual system characteristics and automatically adjusting to dynamic variability in latencies as a function of both the network latency and system characteristics [12]. Recently, Reinforcement Learning (RL) based auto-scaling policies have been demonstrated as an effective tool to automatically adjust containers in Cloud Computing environments [12].

RL based auto-scaling policies explore the choices of adding, removing or retaining container instances with respect to service request invocation variations. Such a strategy is effective in automatically learning which auto-scaling decision to execute with respect to the service request invocation variability. However, unrestrained exploration of RL based policies [89] often leads to violations in user-perceived latencies. To circumvent latency violations, our main proposal in this chapter is to develop a Safe-RL based auto-scaling policy which automatically learns when to execute the appropriate auto-scaling decisions while ensuring that the probability of latency violations is minimized. A Safe-RL policy [89, 115, 116] employs latency specifications in Temporal Logic to tailor the training process. Whenever the RL agent executes a decision leading to a latency violation, the reward function in conjunction with the specification ensures that in subsequent exploration cycles, such decisions are executed with low probability. The main highlights of this chapter are as follows:

- We use a Markov Decision Process (MDP) to model the MEC environment.
- We demonstrate how latency requirements can be formally specified in LTL.

- We develop a quantitative reward formulation to characterize user incurred latencies.
- We prove the convergence of our proposed approach.
- We use the Q-Learning algorithm to learn the rewards associated with the MDP.
- We present experimental results of our algorithm in practical scenarios driven on a test-bed setup with multiple heterogeneous real-world benchmark applications.

The rest of this chapter is organized as follows. Section 5.2 discusses the modeling and generation of safe auto-scaling strategies. Section 5.3 details the obtained results. Section 5.4 concludes the chapter.

5.2 Detailed Methodology

In an MEC environment, users invoke service requests simultaneously for a myriad of different application types. Since multiple applications are deployed on each MEC server, the resulting contention arising from accessing shared resources can have a critical impact on the user-perceived latency. Horizontal auto-scaling serves as a viable mechanism to alleviate resource contention by automatically provisioning/de-provisioning application container instances to enhance load balancing. The horizontal auto-scaling policy governs when to *add/remove/retain* container instances of an application at a particular edge-site. For multiple applications, the challenge is in determining the required number of container instances to ensure adherence to pre-specified application specific latency requirements as a function of the service invocation traffic and the resulting latencies arising out of shared resource contention. For a particular application, an overtly aggressive strategy can spawn a large number of container instances leading to blockage of resources which could have proved beneficial if allocated to other applications. On the other hand, a restrained strategy may lean towards spawning lower number of containers leading to higher latencies, ultimately culminating in latency threshold violations. Additionally, in the MEC environment, users located within a particular edge site can also access edge servers associated with other edge sites with an additional access latency. In certain scenarios, such as for non-safety critical applications, provisioning service requests at other edge sites where container instances of the application are already deployed can also prove to be beneficial instead of deploying a new container instance. In such scenarios, the challenge is in determining whether to spawn a new container instance locally or determine whether container instances already deployed at other distant edge sites can ensure latency conformance. We propose a Safe RL based approach to generate horizontal auto-scaling policies to maximize the probability of adherence to latency requirements. Our Safe RL based framework has the following salient features:

- We use a distributed approach where each edge site is associated with an auto-scaling agent for each application.
- We use an MDP to model the MEC environment and the latency experienced by users in the MEC environment while concurrently representing non-deterministic auto-scaling decision making.
- We demonstrate the composition of LTL based latency specifications with the MDP model of the MEC environment, resulting in another MDP with two types of states, safe and unsafe, with safe states defined in accordance with the LTL specification.
- We quantitatively characterize each safe state in the composed MDP to design a reward function which acts as a guide, tailoring the training process of the RL agent to ensure adherence to the LTL specifications.
- We use Safe-Q-Learning [89] to automatically synthesize the horizontal auto-scaling policy.
- We prove that the composed state space with the quantitative characterization of safe states preserves all properties of standard Safe-RL methodologies ensuring convergence.

We first formally define the horizontal auto-scaling strategy synthesis problem and then discuss each of these aspects in detail in the following sub-sections.

5.2.1 Problem Formulation and Assumptions

We consider the following setup in this work:

- An MEC system comprising n MEC service provider sites, $E = \{E_1, E_2, \dots E_n\}$, where each site is represented by its latitude and longitude coordinates.
- Each site E_j has an associated service zone radius E_j^r.
- Each site $E_j \in E$ has a set of associated servers $ES_j = \{ES_{j1}, ES_{j2}, \dots ES_{jm}\}$ distributed over its service zone.
- We assume edge sites do not share edge servers, i.e., $ES_x \cap ES_y$ is empty, where $1 \leq x, y \leq n, x \neq y$.

Notation	**Description**
E	set of edge sites
ES_j	set of edge servers associated with edge site E_j where $ES_j = \{ES_{j1}, ES_{j2}, \dots ES_{jm}\}$
ES_{jp}	pth server of site E_j where $1 \leq p \leq m$
E_j^r	service zone radius of edge site E_j
E_j^{χ}	edge sites within χ-hop neighbourhood of E_j
A	set of application services where $a_k \in A$
a_k	an application $\in A$
U	set of all users associated with E
t	length of discrete time slot
$c_t^{a_k}$	number of containers of a_k in slot t
$U_t^{a_k}$	set of users accessing a_k in slot t
$l_t^{a_k}$	average latency of $U_t^{a_k}$ in slot t
$L_{interval}$	latency discretization interval
$L_{max}^{a_k}$	maximum threshold latency of a_k
Λ	set of auto-scaling decisions : {retain, upscale, downscale}
λ	a particular auto-scaling decision $\in AS$
M	MDP of auto-scaling policy
ϕ	LTL formula
A_ϕ	LDBA for an LTL formula
M'	composition of MDP M and LDBA A_ϕ
$R'_B(s')$	reward for state s'
$\Gamma'_B(s')$	discount factor for state s'
ρ	path of an MDP
$Ret_t(\rho)$	return of path ρ

Table 5.1: Table of Notations

- Application services $A = \{A_1, A_2, \dots A_p\}$ are deployed across MEC sites E. We assume all microservices of a particular application are deployed together at an edge server. Henceforth in this chapter, we refer to an application container as the set of all microservices associated with the application.
- An application $a_k \in A$ is deemed as deployed at an MEC site $E_j \in E$ only if an application instance of a_k is deployed on at least one of the servers associated with E_j, i.e, $ES_{jp} \in \{ES_{j1}, ES_{j2}, \dots ES_{jm}\}$ where $1 \leq p \leq m$.
- A server $ES_{jp} \in ES_j, 1 \leq p \leq m$ can deploy at most a single container instance of an application.
- A set of users $U = \{u_1, u_2, \dots, u_q\}$ access application services with each user's location specified by the latitude and longitude coordinates used to determine which users are located within which edge site's coverage zones.
- A user $u_j \in U$ can access an application service $a_k \in A$ deployed on $E_j \in E$ only if u_j is located inside E_j^r.

- We assume a discrete time model as in [63] where time is discretized into slots of duration t units.
- All user requests routed to an application container instance of a_k deployed on a particular edge server ES_{jp} are assumed to be homogeneous. By homogeneous computation demands, we mean that the user requests comprise tasks with homogeneous resource requirements, for example, object recognition tasks with identical image and resolution. However, our framework caters to multiple applications simultaneously deployed on an edge server as well with homogeneity assumed only within an application. For example, an edge server can host both image and video processing applications simultaneously.
- We assume availability of broadcast messages between edge sites to update the status of container deployments and assume reliable delivery with negligible delivery time.

Let $c_t^{a_k}$ denote the number of container instances of $a_k \in A$ deployed at an edge site E_j in time slot t. Note that $1 \leq c_t^{a_k} \leq |ES_j|$ since each server can deploy at most a single container instance of a_k. At $E_j \in E$, $U_t^{a_k} \subseteq U$ denotes the set of users accessing an application $a_k \in A$ in a particular time slot t. Let $l_t^{a_k}$ denote the average latency experienced by $U_t^{a_k}$ while $L_{max}^{a_k}$ denotes the maximum threshold latency requirement of $a_k \in A$. We now define latency below.

Definition 5.1 [**Latency:**]
For each user $u \in U_t^{a_k}$, we define latency as the turn-around time from the time point at which the service request is initiated to the time point at which the output of the service invocation is available to the user. We consider this latency as the sum of access and computation latencies as in Chapter 3.

A horizontal auto-scaling policy for an application a_k executes one of the following three actions in each time step t:

- retain the number of container instances $c_t^{a_k}$.
- increase $c_t^{a_k}$ by 1.
- decrease $c_t^{a_k}$ by 1.

We now define the horizontal auto-scaling policy synthesis problem formally:

Definition 5.2 [**Horizontal Auto-Scaling Synthesis Problem:**]
At each discrete time-step t, the horizontal auto-scaling policy synthesis problem determines which of the three auto-scaling decisions to execute such that the probability of users incurring latency violations, i.e., $l_t^{a_k} > L_{max}^{a_k}$ is minimized.

In the following sub-section, we describe our auto-scaling management architecture.

5.2.2 Horizontal Auto-Scaling Management Architecture

We use a distributed approach where each edge site is associated with an auto-scaling agent for each application. The auto-scaling agent stores the number of container instances of the application currently deployed in the edge site. Further, the auto-scaling agent also stores the number of container instances of the application deployed at other edge sites within a χ-hop neighbourhood defined as in the following.

Definition 5.3 [**χ-hop Neighbourhood:**]
Two edge sites E_i and E_j are defined as 1-hop neighbours if their coverage areas intersect. For an edge site E_j, E_j^1 denotes the set of edge sites which are 1-hop neighbours of E_j. E_j^2 denotes the set of edge sites which are 2-hop neighbours of E_j defined as: $E_j^2 = \{\forall E_i \in E$ where $E_i \notin E_j^1$ and $E_i \neq E_j, \exists\ E_k \in E_j^1 | E_k$ and E_i are 1-hop neighbours$\}$ Thus, E_j^χ is defined inductively as the χ-hop neighbourhood.

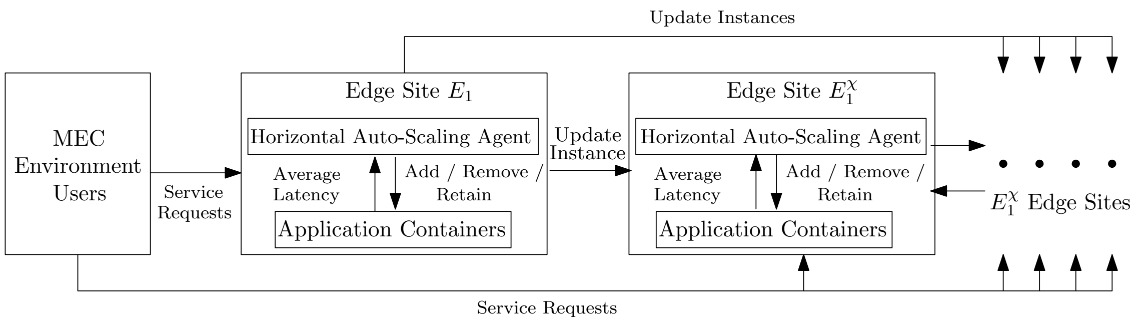


FIGURE 5.1: System Architecture for edge site E_1, the same setup is replicated for $E_2, \ldots E_n$

The auto-scaling agent monitors the latencies experienced by the users located within the edge site for the duration of a discrete time slot. It then decides whether to add/remove/retain container instances within that edge site. Upon deciding on the number of container instances, the auto-scaling agent for the edge site broadcasts the newly updated number of container instances across all χ-hop neighbouring edge sites. Such a setup is replicated across all edge sites. Each auto-scaling agent updates the number of container instances only within the edge site and upon updation, a broadcast message is sent to all χ-hop neighbourhood edge sites. Therefore no additional synchronization is required between the auto-scaling agents. Figure 5.1 summarizes the details of our proposed auto-scaling management architecture. The diagram depicts the overall data flow for the edge site E_1. E_1^χ denotes the set of edge sites which are within χ-hop distance of E_1. Once the agent decides on the number of container instances, it sends a broadcast message to all E_1^χ edge sites to update the deployment status. The setup is replicated across all edge sites $E_1, E_2, \dots E_n$. In the next section, we discuss our formal model of the auto-scaling policy at each edge site.

5.2.3 Formal MDP Model of an Auto-Scaling Policy

In this section, we first explain the philosophy behind our MDP. Figure 5.2 represents the MDP representation for an application a_k at an edge site E_j.

Definition 5.4 [**Auto-Scaling MDP:**]
The auto-scaling MDP $\mathcal{M}$ is a 6-tuple (S, Λ, P, AP, L, R) defined as:

- *S is a finite set of states. Vector $\langle noOfContainers, neighbourhoodCount, averageLatency\rangle$ represents each state $s \in S$.*
- *Λ is the set of auto-scaling actions, i.e., retain the number of container instances / increase the number of container instances by one / decrease the number of container instances by one.*
- *P is a finite set of probabilistic transitions where each transition takes M from one state $s_1 \in S$ to another $s_2 \in S$ on an action $\lambda \in \Lambda$.*
- *AP is a set of Atomic Propositions corresponding to $noOfContainers$, $neighbourhoodCount$, $averageLatency$.*
- *A labelling function $L : S \rightarrow 2^{AP}$ defines the labels associated with each state $\langle noOfContainers,\ neighbourhoodCount,\ averageLatency\rangle$.*
- *R is a reward function. The MDP in a state $s_1 \in S$, on action $\lambda \in \Lambda$ transitions to a state $s_2 \in S$ and generates a reward defined by $R(s_1, \lambda, s_2)$.*

We utilize a model-free approach where the probabilities are not used for our approach. Hence all such transitions are either deterministic or non-deterministic. Transitions can broadly be classified into three types: transitions depicting the non-deterministic nature of latencies, non-deterministic choices of auto-scaling decisions and deterministic transitions representing updation of the number of container instances in the χ-hop neighbourhood. We now explain each component of the MDP M in detail below.

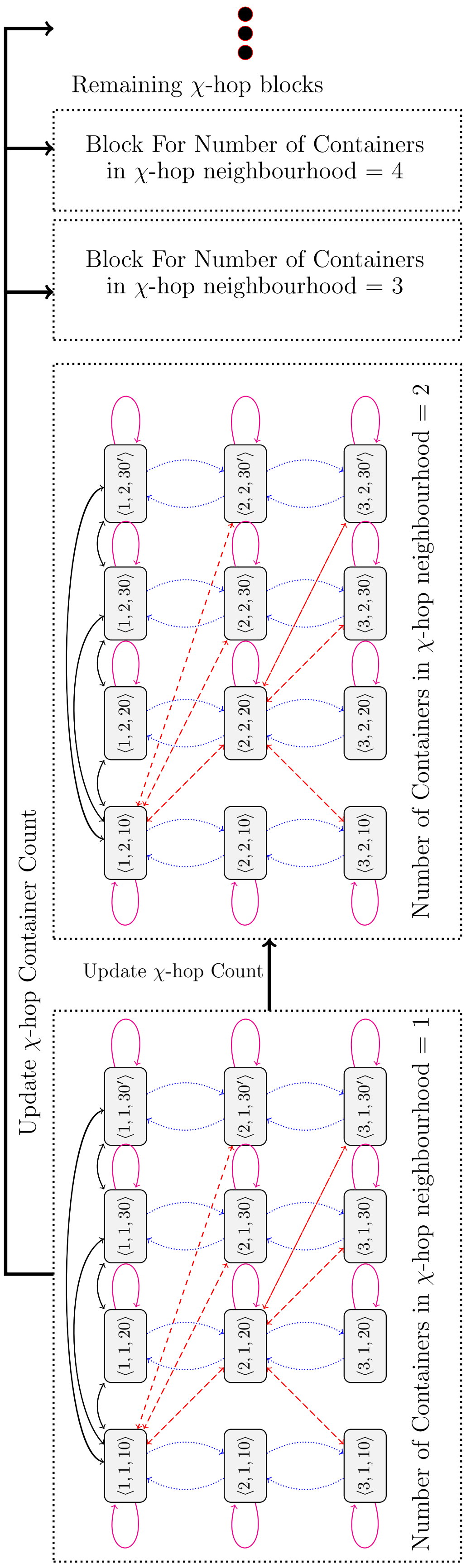


FIGURE 5.2: MDP Representation of Auto-Scaling Policy. All transitions have not been depicted for brevity.

5.2.3.1 State Representation of the MDP

We follow a state representation similar to [12], whilst incorporating inter-edge site communication. In each state, $noOfContainers$ represents $c_t^{a_k}$, the number of servers on which the application a_k is deployed at E_j, $neighbourhoodCount$ represents the number of edge sites in the χ-hop neighbourhood where container instances of a_k are deployed while $averageLatency$ represents $l_t^{a_k}$, the average latency of all users accessing application a_k in time slot t. The rationale behind representing only the aggregated count of the number of edge sites where container instances of a_k are deployed is to keep the state space small, and not encounter state space explosion, since representing them individually may take more space. The latency representation is discretized into equal sized sub-intervals in the closed interval $[0, L_{max}^{a_k}]$. Additionally, the state $\langle noOfContainers, neighbourhoodCount, {L_{max}^{a_k}}' \rangle$ depicts the scenario where the average latency of $U_t^{a_k}$ exceeds the threshold $L_{max}^{a_k}$. Latency is thus represented as a discretized measure [63] where the size of each discretized interval is denoted by $L_{interval}$. The concrete continuous valued latencies are mapped to the discretized interval latency set as follows: latencies between 0 and $L_{interval}$ in the continuous interval $[0, L_{interval})$ are mapped to the state where the $averageLatency$ component is $1 \times L_{interval}$, latencies in the continuous interval $[L_{interval}, 2 \times L_{interval})$ are mapped to $2 \times L_{interval}$ and so forth, where $[0, L_{interval})$ denotes the continuous interval inclusive of the lower bound 0 and exclusive of $L_{interval}$. Latencies which exceed the threshold $L_{max}^{a_k}$ are mapped to ${L_{max}^{a_k}}'$.

Example 5.2.1. In Figure 5.2, we consider the representation of an auto-scaling policy for an application where $L_{max}^{a_k} = 30ms$ with sub-intervals $L_{interval}$ as $10ms$ with number of edge servers $m = 3$ at the edge site. Each dotted rectangular block corresponds to the number of edge sites within the χ-hop neighbourhood where a_k is deployed. The state $\langle 1, 1, 10 \rangle$ denotes the scenario when the average latencies of users accessing the application service a_k at E_j evaluates to a value in the interval $[0, 10)ms$, when the application service is deployed on a single server and the number of edge sites in the χ-hop neighbourhood where the application is deployed is also 1. Thus, the latency identifier of a state vector is a discrete intervalized representation of continuous latency values. Our work builds on a similar discretized interval representation as in [12, 63]. Similarly, $\langle 1, 1, 20 \rangle$ is the scenario when the average latency is between $[10, 20)ms$. The state $\langle 1, 1, 30' \rangle$ denotes the scenario when the average latency exceeds 30ms. Such states correspond to the application deployed only on a single server. The state $\langle 2, 1, 10 \rangle$, on the other hand, exhibits the application being deployed on two servers within the edge site. The MDP captures all possible container deployment scenarios along with all discretized intervals. ■

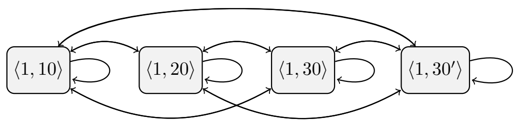


FIGURE 5.3: Latency Transitions

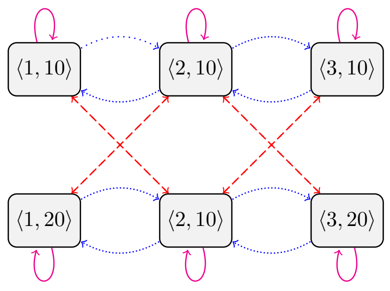


FIGURE 5.4: Auto-Scaling Transitions

5.2.3.2 Transition Representation of MDP

Transitions can broadly be classified into three types: transitions representing updation of the number of container instances in the χ-hop neighbourhood, transitions representing the non-deterministic nature of latencies incurred by users and transitions representing the non-deterministic choices of auto-scaling decisions. Each transition occurs only at each discrete time step.

Update Neighbourhood Count Transitions: At each discrete time step, an edge site receives broadcast messages from all edge sites in the χ-hop neighbourhood. The state of the MDP is then updated to reflect the aggregated count of the number of edge sites in the χ-hop neighbourhood where the application a_k is deployed. Such transitions are represented by "Update χ-hop Container Count" in Figure 5.2 but not explicitly depicted in the figure for brevity.

Example 5.2.2. Update χ-hop Container Count transitions exist between all states between the blocks represented by dotted rectangles in Figure 5.2. A transition between $\langle 1, 1, 10\rangle$ to $\langle 1, 2, 10\rangle$ denotes the scenario when the number of edge sites where the container instances are deployed in the χ-hop neighbourhood has changed from 1 to 2 while the number of container instances deployed within the edge site remains 1 and there is no change in $averageLatency$. Similarly, a transition from $\langle 1, 1, 10\rangle$ to $\langle 2, 2, 10\rangle$ depicts changes in container counts both within the edge site as well as the neighbourhood. ■

Note that our model enables the representation of scenarios where communication between edge sites is not permitted. In such scenarios, the $neighbourhoodCount$ Atomic Proposition is set to $NULL$. In the subsequent discussions, for ease of explanation, we elucidate our methodology in scenarios where $neighbourhoodCount$ is set to $NULL$. We use the shorthand representation $\langle noOfContainers, averageLatency\rangle$ to denote such scenarios. We use this shorthand notation henceforth. We explain later how our methodology generalizes to scenarios where communication within a χ-hop neighbourhood is also taken into consideration.

Latency Transitions: Latency transitions represent the non-deterministic nature of servers where average latencies vary over time. Latency transitions are depicted by Black Solid Curves in Figure 5.2. Since we use a discrete time model, the average latencies are calculated at each discrete time step and the state in the MDP updated correspondingly. In each state, the $noOfContainers$ component denotes a particular container deployment configuration. For a particular container deployment configuration, latency transitions are included to all other states for the same container deployment configuration, i.e, for two states $s_1 : \langle s_1\_noOfContainers, s_1\_averageLatency\rangle$, $s_2 : \langle s_2\_noOfContainers, s_2\_averageLatency\rangle$, latency transitions are included when $s_1\_noOfContainers = s_2\_noOfContainers$.

Example 5.2.3. Figure 5.3 depicts all possible latency transitions for $noOfContainers = 1$. From state $\langle 1, 10\rangle$, the transitions to states $\langle 1, 10\rangle$, $\langle 1, 20\rangle$, $\langle 1, 30\rangle$ and $\langle 1, 30'\rangle$ represent changes in average latencies experienced by users during a discrete time step. The self-transition at $\langle 1, 10\rangle$ denotes no change in average Latency. Such transitions are included at all states where $noOfContainers = 1$, since latency transitions only depict changes in average latencies while retaining the same container deployment. Note that no latency transition is included from $\langle 1, 10\rangle$ to $\langle 2, 10\rangle$ since such a transition depicts a change in the container deployment configuration while the $averageLatency$ component is unaffected. We depict only some of the latency transitions for $noOfContainers = 2$ *and* 3 in Figure 5.2. Other latency transitions are implicitly present but omitted in Figure 5.2 for brevity. ■

Latency transitions effectively characterize fluctuations in average latencies over time due to variations in service invocation and the resulting contention to access resources. Such transitions thus aid in executing auto-scaling transitions as a function of average latency. We discuss auto-scaling transitions next.

Auto-Scaling Transitions: In each state, auto-scaling transitions correspond to the three possible choices of auto-scaling decisions, denoted by Λ, at each discrete time step:

- retain the same number of container instances deployed.
- increase the number of container instances by 1.
- decrease the number of container instances by 1.

Auto-scaling transitions are depicted by blue dotted curves in Figure 5.2. Such transitions are identical for all the states apart from the ones labelled with $noOfContainers$ component as 1 since in such states container instances cannot be removed. Such states have only two possible auto-scaling choices, i.e., to retain or add a container instance. Additionally, for states labelled with $noOfContainers$ as the maximum number of edge servers, i.e. $|ES_j|$, only retain and remove container decisions are present. For states in which the latency component is $L^{a_k}_{max}$, the remove container action is not represented since a latency violation trivially implies increasing container instances. As a result of executing an upscaling decision where the number of containers is increased by 1, the average latency experienced by users may decrease. Such scenarios are depicted by changes to both the elements of the $\langle noOfContainers, averageLatency\rangle$ vector.

Example 5.2.4. Figure 5.4 depicts auto-scaling decisions on a fragment of the MDP in Figure 5.2 for $averageLatency = 10$ and $averageLatency = 20$. The transition from $\langle 1, 10\rangle$ to itself depicts the decision to retain the number of container instances same in successive time steps while there is no change in average latency. Such transitions are depicted by magenta self-loops in states. The transition from $\langle 1, 10\rangle$ to $\langle 2, 10\rangle$ similarly depicts the scenario where the number of container instances is increased by 1. A decrement in the number of container instances is depicted by the transition from $\langle 2, 10\rangle$ to $\langle 1, 10\rangle$. Note that in such scenarios there is no change in average latency. Such transitions are depicted by blue dotted curves. Note that only one container instance is added/removed at any discrete time step and hence transitions such as $\langle 1, 10\rangle$ to $\langle 3, 10\rangle$ are absent. The transition from $\langle 1, 20\rangle$ to $\langle 2, 10\rangle$ depicts the scenario where the number of containers is increased by 1 and as a consequence, the average latency has decreased from the interval [10, 20) to [0, 10). Changes in both components are depicted by red dashed lines. ■

5.2.3.3 Updation Semantics

At a particular state $s_1 : \langle s_1\_noOfContainers, s_1\_averageLatency\rangle$, one of the three possible choices of auto-scaling decisions, $\lambda \in \Lambda$ is executed. The average latency $l^{a_k}_t$ is then observed for the duration of a discrete time-step. The state of the MDP is updated to $s_2 : \langle s_2\_noOfContainers, s_2\_averageLatency\rangle$, where $s_2\_noOfContainers$ denotes the number of containers as a result of executing the auto-scaling decision and $s_2\_averageLatency$ denotes the subsequently observed average latency in the next discrete time-step. On transition from s_1 to s_2, a reward is generated according to a reward function R which is utilized by RL agents to automatically learn which auto-scaling decisions to execute in each state of the MDP. We discuss the role of the reward function in the RL strategy synthesis process in Section 5.2.4.

The MDP built above encompasses the underlying solution space for our problem context, accounting for all non-deterministic auto-scaling decisions and the resulting non-deterministic average latencies from execution of the auto-scaling actions. Note that although the MDP encompasses the entire solution space, the rewards for executing the auto-scaling decisions from the various MDP states are initially unknown. In order to determine these unknown rewards characterizing the effectiveness of the auto-scaling decisions in each MDP state, we formulate an RL problem where the RL agent learns the rewards by utilizing the MDP as the state space.

5.2.4 Safe-RL Strategy Synthesis

Standard RL approaches using the MDP solution space as described above do not provide any guarantees on the auto-scaling policy synthesized with respect to latency requirement thresholds [89]. In a latency driven MEC environment, adherence to critical latency requirements is of utmost importance. To ensure adherence to such pre-defined latency measures, we use Safe-RL which ensures conformance of the synthesized strategy to latency threshold specifications defined in LTL. The LTL specification is translated to a Limit Deterministic Büchi Automata (LDBA) [89] which acts as a guide to the learning process. The reward function is then defined on the composition of the MDP encompassing the solution space and the LDBA corresponding to the LTL specification. Standard Safe-RL strategies, however, define rewards distinguishing only between safe and unsafe states. Such a distinction, however, does not fully characterize the latency-driven MEC environment. We define a reward function quantitatively distinguishing each safe state using the *averageLatency* component of the state. We discuss our approach in detail in the following sub-sections.

5.2.4.1 Representation of Latency Specifications in LTL

We utilize the discretization of latency characteristics as described in the MDP construction to specify latency requirements in LTL. We first formally define LTL formulae and the satisfaction relation of an LTL formula on an MDP:

Definition 5.5 [**Linear Temporal Logic:**]
LTL formulae [37] over a given set of atomic propositions AP are syntactically defined as:

$$\phi : true \mid \psi \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 \, U \phi_2$$

where $\psi \in AP$, and the operators X and U are called next and until, respectively. Using the until operator we define two further temporal modalities: (1) eventually, $F\phi = true \; U \; \phi$; and (2) always, $G\phi = \neg F \neg \phi$.

Definition 5.6 [**MDP Path:**]
A path of an MDP $\mathcal{M} : (S, \Lambda, P, AP, L, R)$ is an infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots$ with $s_i \in S$ such that for all $i \geq 0$, there exists $\lambda \in \Lambda$ where a transition from state s_i to $s_i + 1$

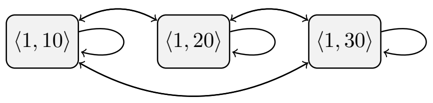


FIGURE 5.5: Fragment of MDP illustrating LTL satisfaction

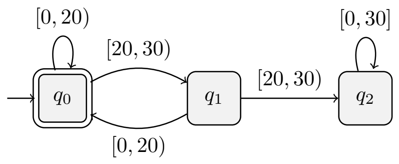


FIGURE 5.6: LDBA for $G(latency = 30 => X(latency = 20))$

due to action λ is present. We use $\sigma[i]$ to denote the state s_i, as well as $\sigma[: i]$ and $\sigma[i+1 :]$ to denote the prefix $s_0 s_1 \ldots s_i$ and the suffix $s_{i+1}s_{i+2}\ldots$ of the path, respectively.

Definition 5.7 [**LTL Satisfaction on MDP:**]
The satisfaction of an LTL formula ϕ for a path ρ of an MDP denoted by $\rho \models \phi$ is thus defined as: if the first state s_0 of ρ is labelled with ψ, i.e., $\psi \in L(s_0)$, then $\rho \models \phi$; a path ρ satisfies $X\phi$ if $\rho[1 :]$ satisfies the formula ϕ; and finally,

$$\rho \models \phi_1 U \phi_2, \textit{if } \exists i, \rho[i] \models \phi_2 \textit{ and } \forall j < i, \rho[j] \models \phi_1 \tag{5.1}$$

Thus, satisfaction of an LTL formula on an MDP is based on the paths of an MDP.

Satisfaction of an LTL formula on an MDP path aids in synthesizing policies which conform to the LTL specification.

The G operator specifies a particular condition to be *true* in all states of all paths of an MDP. Hence the G operator is utilized to specify safety constraints. We specify latency requirements using the G operator to ensure all states of the MDP satisfy the latency requirement. Consider the latency characteristics of a_k specified in LTL in Equation 5.2 using the always (G) operator.

$$G(\ averageLatency = [w, x) => X(averageLatency = [y, z)\ ) \tag{5.2}$$

The specification states that at any discrete time step if the MDP is in a state where the average latency lies in the interval $[w, x)$, in the next time step, the auto-scaling policy should ensure that the MDP is in a state where the average latency lies in the interval $[y, z)$. Such specifications dictate proactively undertaking auto-scaling decisions to ensure that latency thresholds are not violated. Note that such a specification is representative of a single latency requirement specification. However, any latency characteristic expressed in LTL can be utilized. In order to incorporate LTL specifications as a guide in the RL process, the LTL formula is converted to a corresponding LDBA [89].

Example 5.2.5. Consider the MDP in Figure 5.2 where the latency threshold $L^{a_k}_{max} = 30ms$ and $k = 10ms$. The LTL specification G ($averageLatency = [20, 30) => X(averageLatency = [0, 20)$) indicates that if the average latency experienced by the users of a_k lies in the interval [20, 30) at any discrete time step, in the next time step, the average latency must remain in the interval [0, 20). Consider a fragment of the MDP M as shown in Figure 5.5. Let us consider a path comprising the sequence of states $\rho = \langle 1, 10\rangle$, $\langle 1, 20\rangle$, $\langle 1, 10\rangle$, $\langle 1, 30\rangle$ starting from the state $\langle 1, 10\rangle$. Such a path satisfies the LTL formula $\phi : X\ averageLatency = [10, 20)$, i.e., $\rho \models X\ averageLatency = [10, 20)$, since the second state in the path is labelled with the Atomic Proposition corresponding to $averageLatency = [10, 20)$. Additionally, let us consider the path $\rho = \langle 1, 10\rangle$, $\langle 1, 10\rangle$, $\langle 1, 10\rangle$, $\langle 1, 20\rangle$. Such a path starting from the state $\langle 1, 10\rangle$ satisfies the LTL formula $\phi : averageLatency = [0, 10)\ U\ averageLatency = [10, 20)$ since all states are labelled with $averageLatency = [0, 10)$ until a state labelled with $averageLatency = [10, 20)$ is encountered. ■

5.2.4.2 LTL to Limit-Deterministic Büchi Automaton (LDBA)

Each LTL specification can be represented by an equivalent LDBA. Satisfaction of an LTL formula can then be evaluated on the paths of an LDBA derived from the corresponding LTL formula [89]. We first formally define an LDBA below.

Definition 5.8 [**Limit Deterministic Büchi Automaton:**]
An LDBA is a 5-tuple $A = (Q, \Sigma, \delta, q_0, B)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ is a (partial) transition function, $q_0 \in Q$ is an initial state, and B is a set of accepting states, such that (i) δ is total except for the ϵ-moves, i.e., $|\delta(q, v)| = 1, \forall q \in Q, v \in \Sigma$; and (ii) the set of states Q can be divided into two partitions, Q_I and Q_A such that $Q_I \cup Q_A = Q$, which satisfies:

- *ϵ transitions are not allowed within Q_A, i.e, $\delta(q, \epsilon) = \Phi, \forall q \in Q_A$, where Φ denotes the empty set.*
- *the outgoing transitions from Q_A stay within Q_A, i.e., $\forall q \in Q_A, v \in \Sigma, \delta(q, v) \subseteq Q_A$.*
- *the Büchi accepting states are in Q_A, i.e., $B \subseteq Q_A$.*

An infinite path ρ is accepted by the LDBA if it satisfies the Büchi acceptance condition, i.e., $inf(\rho) \cap B \neq \Phi$, where $inf(\rho)$ denotes the set of states visited by ρ infinitely many times. Thus, an infinite path is accepted if it visits a Büchi accepting state infinitely often.

Example 5.2.6. Figure 5.6 depicts the LDBA corresponding to the LTL latency specification $G\ (\ averageLatency = [20, 30)\ => X(averageLatency = [0, 20)\ )$. The LDBA comprises three states q_0, q_1 and q_2 with state q_0 as an accepting state defined by the Büchi accepting condition. The self transition at q_0 depicts the scenario that the average latency is in the interval $[0, 20)ms$ while the transition from q_0 to q_1 is executed when the average latency lies in the interval $[20, 30)ms$. There are two outgoing transitions from q_1, one to q_0 and another to q_2. The transition from q_1 to q_0 is executed when the average latency incurred is in the interval $[0, 20)ms$ while the transition from q_1 to q_2 is executed when the average latency incurred is in the interval $[20, 30)ms$. The state q_2 is a trap state, from which there is only one self-transition. State q_2 corresponds to a violation in the LTL latency specification where two successive discrete time steps incur average latencies in the interval $[20, 30)$. The transition from q_1 to q_0, on the other hand, signifies an adherence to the LTL specification and hence results in a transition to a Büchi accepting state. Any path which visits the state q_0 infinitely many times, such as $\{q_0, q_1, q_0, q_1, q_0, q_1, q_0 \ldots\}$ with q_0 being visited repeatedly henceforth satisfies the Büchi acceptance criterion. In this case $inf(\rho) = q_0, q_1$ and since $\{q_0, q_1\} \cap B = q_0$, the path is accepting. An infinite path such as $\{q_0, q_1, q_2, q_2, q_2, q_2, \ldots\}$ with q_2 henceforth is not accepted as an infinitely accepting criterion. ■

The state of the LDBA is updated in each discrete time slot according to the *averageLatency* observed in the duration of the time slot. The sequence of such states denote the path traced out by the MEC system. If the path visits the Büchi accepting states infinitely often, the MEC system is deemed to be safe. Thus, the LDBA serves as an effective safe representation of the behaviour of an MEC system with respect to the LTL property. In order to incorporate the LTL specification as a guide to the RL policy synthesis, we utilize the composition (also referred to as product) of the LDBA corresponding to the LTL specification and the MDP corresponding to the auto-scaling policy. This is discussed in the following.

5.2.4.3 Composition of LTL and MDP

The composition of the LDBA and the MDP results in another MDP which serves as a global view of the environment augmented with the LTL property of interest. We now define the composition of MDP and LDBA formally below.

Definition 5.9 [**Composition of MDP and LDBA:**]
Given an LTL formula ϕ representing a latency specification, and an MDP M, the composed (or product) MDP is constructed with an LDBA A_ϕ derived from the LTL formula ϕ [89]. A product MDP of an MDP $M = (S, \Lambda, T, AP, L, R)$ and an LDBA $A_\phi = (Q, \Sigma, \delta, q_0, B)$ is defined as $M' = (S', T', AP', L', R', B')$ where, $S' = S \times Q$ is the set of states where $\times$ is the Cartesian

product operator, $B' = \{\langle s, q\rangle \in S' \mid q \in B\}$ *is the set of accepting states and* T *is the transition function represented as:*

$$T'(\langle s_1, q_1\rangle, \lambda, \langle s_2, q_2\rangle) = \begin{cases} T(s_1, s_2) & if\ q_2 = \delta(q_1, L(s_1)) \\ absent & otherwise \end{cases} \tag{5.3}$$

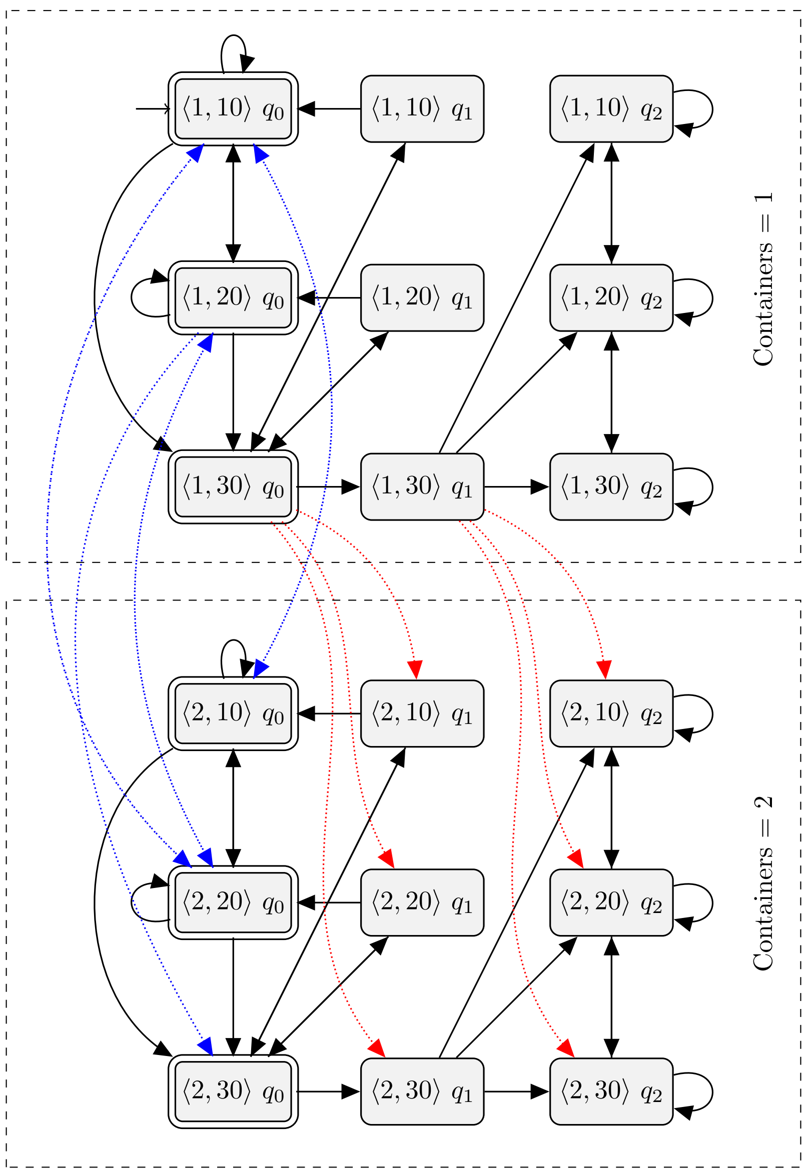
A path ρ *of the product MDP* M' *satisfies the Büchi condition* ϕ_B *if* $inf(\rho) \cap B' \neq \Phi$, *i.e., the path* ρ *visits the set of accepting states of the LDBA* B *infinitely often.*

For the sake of simplicity and ease of explanation, we illustrate the product construction on a subset of the MDP described in Figure 5.2. Figure 5.8 depicts the composition on a subset of the MDP M and the LDBA corresponding to the LTL formula G ($latency = [20, 30)$ => $X(latency = [0, 20)$). Table 5.2 denotes the transitions δ for the LDBA according to which the corresponding transitions are computed using Equation 5.3. The transitions are computed in accordance with the latency APs only. The product yields a global synchronized representation of both the MDP and the LDBA. Transitions in M' are executed whenever the agent executes one of the three auto-scaling decisions. Additionally, in M', the state of the LDBA is also simultaneously updated. Thus, when an auto-scaling action is executed, the next MDP state is determined by the transitions in M while the LDBA simultaneously makes a transition by consuming the label of the current MDP state. We now discuss each component of the product MDP in detail.

Product MDP Semantics: The product MDP can be viewed as comprising several blocks. Each block represents a particular container deployment configuration. The number of blocks corresponds to the number of edge servers associated with an edge site, i.e., the number of blocks = $|ES_j|$.

Example 5.2.7. The MDP in Figure 5.7 depicts two blocks denoted by dotted rectangles corresponding to deployments where the number of containers are 1 and 2 respectively. ■

Product MDP Transitions: The transitions in M' can be categorized into two types: transitions within a block and transitions between blocks. Within each block, transitions represent the behaviour of the LDBA in accordance with latency characteristics when the auto-scaling policy retains the container deployment configuration. Transitions across blocks represent the behaviour of the LDBA in accordance with latency characteristics when the auto-scaling policy executes an up-scaling or down-scaling decision. We discuss each type of transition below.



$\langle 1,10\rangle\ q_0$
$\langle 1,10\rangle\ q_1$
$\langle 1,10\rangle\ q_2$
$\langle 1,20\rangle\ q_0$
$\langle 1,20\rangle\ q_1$
$\langle 1,20\rangle\ q_2$
$\langle 1,30\rangle\ q_0$
$\langle 1,30\rangle\ q_1$
$\langle 1,30\rangle\ q_2$
Containers = 1
$\langle 2,10\rangle\ q_0$
$\langle 2,10\rangle\ q_1$
$\langle 2,10\rangle\ q_2$
$\langle 2,20\rangle\ q_0$
$\langle 2,20\rangle\ q_1$
$\langle 2,20\rangle\ q_2$
$\langle 2,30\rangle\ q_0$
$\langle 2,30\rangle\ q_1$
$\langle 2,30\rangle\ q_2$
Containers = 2

FIGURE 5.7: Composition of MDP and LDBA. All Inter-Block Transitions are not depicted

$\delta(State, AP)$	State
$\delta(q_0, 10)$	q_0
$\delta(q_0, 20)$	q_0
$\delta(q_0, 30)$	q_1
$\delta(q_1, 10)$	q_0
$\delta(q_1, 20)$	q_0
$\delta(q_1, 30)$	q_2
$\delta(q_2, 10)$	q_2
$\delta(q_2, 20)$	q_2
$\delta(q_2, 30)$	q_2

TABLE 5.2: LDBA Transitions in accordance with Atomic Propositions

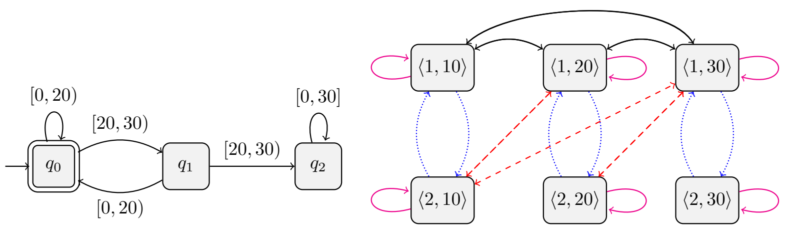


FIGURE 5.8: LDBA Used To Compute Composition and MDP Fragment Used To Demonstrate Composition

Intra-Block Transitions: Transitions within a block depict the scenarios when the auto-scaling agent decides to retain the container deployment configuration. Thus, such transitions correspond to non-deterministic latency transitions as explained in Section 5.2.3. Within each block, the states can be divided into two types, Büchi accepting and Büchi non-accepting. The Büchi accepting states correspond to states which have the LDBA component in the product as an LDBA accepting state, i.e., for any state $s : \{\langle noOfContainers, averageLatency\rangle, q_i\}$, s is accepting if $q_i \in B$. All MDP paths within a block thus represent whether the decision to retain the container deployment configuration adheres to or violates the LTL specification.

Example 5.2.8. There are two blocks in Figure 5.7, one corresponding to $noOfContainers = 1$ and another corresponding to $noOfContainers = 2$. The block which corresponds to $noOfContainers = 1$ comprises 9 states since there are three different latency valued states in the MDP M while the LDBA also comprises three states. Intra-Block transitions are depicted by black curved arrows in Figure 5.7. The transition from state $\{\langle 1, 10\rangle, q_0\}$ to $\{\langle 1, 20\rangle, q_0\}$ denotes the scenario where the MDP M makes a transition from state $\langle 1, 10\rangle$ to $\langle 1, 10\rangle$ while the LDBA A_ϕ remains in state q_0. Such a transition is possible since the LDBA A_ϕ in state q_0 upon encountering 10 as the latency atomic proposition remains in state q_0. However, in state $\{\langle 1, 30\rangle, q_0\}$, note that all outgoing transitions correspond to states where the LDBA a_ϕ makes a transition to q_1. Such transitions correspond to the LDBA's transitions from state q_0 to q_1 upon encountering the latency atomic proposition 30. All such transitions are computed according to Table 3 which outlines how the LDBA executes transitions in accordance with latency atomic propositions as encountered in the MDP M. Since q_0 is the Büchi acceptance state in the LDBA A_ϕ, all states in the product comprising of q_0, are deemed as Büchi accepting states. ■

Inter-Block Transitions: Transitions between blocks depict the scenarios when the auto-scaling agent decides to change the container configuration, i.e, add or remove a container instance and the resulting non-deterministic latency changes as a result. Thus, such transitions correspond to the auto-scaling decision transitions as explained in Section 5.2.3. Additionally, since such transitions are also constructed in accordance with the δ transition function of the LDBA, all paths of the MDP involving inter-block transitions represent whether the decision to upscale/downscale the container deployment adheres to or violates the LTL specification.

Example 5.2.9. Inter-block transitions are depicted by blue dashed curved arrows in Figure 5.7 which indicate changes in container deployment configurations. All such transitions are also computed according to Table 5.2 since the LDBA A_ϕ represents the required latency specifications without any reference to a particular container deployment configuration. Similar to the intra-block transitions, transitions from LDBA state q_0 to q_1 are executed upon encountering the latency atomic proposition 30. The transitions from the LDBA state q_1 to

q_2 are executed upon encountering two successive MDP states with latency atomic proposition 30. Thus, states which include q_2, indicate a violation in the LTL specification and hence all outgoing transitions from states with the LDBA component as q_2, remain within such non-accepting Büchi states. ■

Updation Semantics: The states of the product MDP M' are updated in a manner similar to the auto-scaling policy MDP. At a state $\{s_1 : \langle s_1\_noOfContainers, s_1\_averageLatency\rangle\ s_1\_q_x\}$, one of the three possible choices of auto-scaling decisions is executed. The average latency $l_t^{a_k}$ is then observed for the duration of a discrete time-step. The state of the MDP is updated to $\{s_2 : \langle s_2\_noOfContainers, s_2\_averageLatency\rangle\ s_2\_q_y\}$, where $s_2\_noOfContainers$ denotes the number of containers as a result of executing the auto-scaling decision and $s_2\_averageLatency$ denotes the subsequently observed average latency in the discrete time-step. Additionally, in M', the LDBA state is also updated according to the observed latency. The new state q_y of the LDBA is determined by $\delta(q_x, s_2\_averageLatency)$.

Example 5.2.10. The transition from $\{\langle 1, 10\rangle\ q_0\}$ to $\{\langle 1, 20\rangle\ q_0\}$ depicts the scenario where the number of containers is retained as 1 but the *averageLatency* component has changed to the interval [10, 20). Since the LDBA component was in state q_0, the LDBA transition is computed according to $\delta(q_0, [10, 20)) = q_0$. ■

A path $\rho = s'_1, s'_2, s'_3 \ldots$ of M' thus denotes a sequence of states depicting the auto-scaling decisions, the resulting average latency as a consequence and the corresponding LDBA transition. Each state of the path is of the form $\{\langle noOfContainers, averageLatency\rangle\ q_x\}$. The path ρ satisfies the Büchi acceptance condition only if the sequence of LDBA states q_x in ρ satisfies the Büchi acceptance condition.

Example 5.2.11. Consider the path $\rho = \{\langle 1, 10\rangle\ q_0\}, \{\langle 1, 20\rangle\ q_0\}, \{\langle 1, 10\rangle\ q_0\}$ in Figure 5.7. The path satisfies the Büchi acceptance condition since the accepting state q_0 is visited infinitely often. Additionally, let us take into consideration another path in Figure 5.7 $\rho = \{\langle 1, 20\rangle\ q_0\}, \{\langle 1, 30\rangle\ q_0\}, \{\langle 1, 20\rangle\ q_1\}, \{\langle 1, 20\rangle\ q_0\}, \{\langle 1, 20\rangle\ q_1\} \ldots$ with the path remaining in state $\{\langle 1, 20\rangle\ q_1\}$ henceforth. Such a path also satisfies the Büchi acceptance condition since the accepting state q_0 is visited infinitely often on this path. However, consider the path $\rho = \{\langle 1, 20\rangle\ q_0\}, \{\langle 1, 30\rangle\ q_0\}, \{\langle 1, 30\rangle\ q_1\}, \{\langle 1, 20\rangle\ q_2\}, \{\langle 1, 20\rangle\ q_2\} \ldots$ with the path remaining in state $\{\langle 1, 20\rangle\ q_2\}$ henceforth. Such a path does not satisfy the Büchi accepting state since only q_2 is visited infinitely often which is not a Büchi acceptance state. ■

The paths of M' are thus determined by the transition on the original MDP M simultaneously with the LDBA A_ϕ. The product MDP comprises the entire space of resulting latencies including both safe and unsafe latencies. A sequence of auto-scaling decisions is thus represented by paths of M'. During the learning process, there is a possibility that the MDP can enter an unsafe state from which it can not return to a safe state. In the subsequent RL episode, the MDP and LDBA states are reset. As a consequence, the MDP once again starts off in a safe state. The MDP obtained by the composition of M and A_ϕ thus encompasses all auto-scaling decisions, the corresponding latency observations and the corresponding compliance with the LTL specification. Note that the transitions of the product MDP are computed in accordance with the *averageLatency* atomic proposition AP of the LDBA. The AP corresponding to *neighbourhoodCount* is not present in the LDBA. As a consequence, in the generalized scenario when we also consider the active container deployments in the χ-hop neighbourhood as with the MDP in Section 5.2.3, the only difference is in the number of total states in the product MDP and the resulting transitions among those states. The methodology outlined above is utilized to compute the transitions on the generalized product MDP. Variations in latencies occurring as a result of the varying number of container deployments within the χ-hop neighbourhood is encoded via the extra states in the generalized product MDP. This composition is used as the solution space of the RL strategy synthesis method which we describe next.

5.2.4.4 Synthesizing Safe Policies

In RL, an agent automatically learns when and which auto-scaling decisions to execute via trial-and-error interactions with the MEC environment. The agent and the environment interact in discrete time steps. At the beginning of each step t, the agent observes the state of the environment represented by $s'_t \in S'$ in MDP M', and out of the three possible choices of auto-scaling decisions Λ, executes an auto-scaling decision, $\lambda \in \Lambda$. The state of M' is then updated to s'_{t+1} in accordance with the updation semantics discussed in Section 5.2.4.3. A reward $r_{t+1} = R(s'_t, \lambda, s'_{t+1})$ is returned to the agent, which servers as an indicator of the effectiveness of the agent's auto-scaling decision λ in state s'_t. The return for a particular state s', is the cumulative future discounted reward $R = \sum_{t=0}^{\infty} \gamma^t r_t$, where r_t is the immediate reward at time step t, and $\gamma \in [0, 1]$ is the discount factor that controls the influence of future rewards. A policy π for an MDP M is a function $\pi : S' \rightarrow \Lambda$. Thus, a policy is a mapping from each state of the MDP to one of the three auto-scaling decisions. The objective of the agent is to learn an optimal policy π^* that maximizes the return for all states $s' \in S'$. The reward accumulated is used by the agent to learn the optimal policy π^*.

In this paper, we use Q-Learning [34] to synthesize auto-scaling policies. Q-learning utilizes a Q-matrix initialized with the state space S' of the product MDP M'. Each state is associated with three Q-values, representing the three auto-scaling decisions Λ. We consider the simple ϵ-greedy action selection method: at any decision step t, with probability ϵ, Q-learning chooses a random auto-scaling decision to improve its knowledge of the application, whereas, with probability $1-\epsilon$, it selects the auto-scaling decision with the highest Q-value, i.e., $\lambda = \text{argmax}_{\Lambda}\ Q(s'_t, \Lambda)$. Most of the time, the ϵ-greedy policy selects the best known auto-scaling decision for a particular state, while it favors the exploration of other auto-scaling decisions with low probability. After executing the auto-scaling decisions, at the end of the time step, the environment returns a reward to the RL agent. The standard Q-Learning algorithm when used with the solution space M' as the Q-matrix does not differentiate between Büchi and non-Büchi states. In order to synthesize policies which adhere to the LTL specification, the discount factor and the reward function need to be re-formulated.

Note that M' comprises two broad categories of paths: paths which adhere to application latency requirements and visit only safe (Büchi) states and paths which includes unsafe (non-Büchi) states that exceed the threshold latency requirements. Standard RL based policies do not provide any guarantees on whether auto-scaling decisions result in paths which visit only safe (Büchi) states [89, 117]. A Safe-RL policy characterizes the return of each path to ensure that the probability of visiting safe (Büchi) states is maximized. As a consequence, Safe-RL policies ensure that the policies generated adhering to the LTL specifications minimize the probability of latency violations [89, 117]. Additionally, the return of a Safe-RL policy ensures that the policy generates auto-scaling decisions even when the latency requirements cannot be adhered to (maybe due to lack of resources). Note that traditional RL policies when incorporated with constraints corresponding to latency requirements do not consider such scenarios. Traditional RL based auto-scaling policies deal with minimizing *overall latency* as opposed to our proposed Safe-RL approach which aims at minimizing *latency violations*. We now formally define the return of paths in our Safe-RL based approach. A model free RL algorithm such as Q-Learning always generates a policy π^{ϕ} that maximizes the probability of satisfaction of an LTL specification ϕ on M' if the return of a path is defined in a specific way, as summarized below [89].

Theorem 5.1. *For a given MDP M' with $B' \subseteq S'$ the value function v^{π} for policy π and discount factor $0 < \gamma < 1$ satisfies $\lim_{\gamma \to 1^-} v^{\pi}(s') = Pr_{\pi}(s' \models GF\ (B'))$ for all states $s' \in S'$, if the return of a path is defined as*

$$Ret_t(\rho) = \sum_{i=0}^{\infty} R'_{B'}(\rho[t+i]) \prod_{j=0}^{i-1} \Gamma_{B'}(\rho[t+j]) \tag{5.4}$$

where $\prod_{j=0}^{-1} = 1$, $R'_{B'} : S' \to [0,1)$ and $\Gamma_{B'} : S' \to (0,1)$ are the reward and discount functions defined as:

$$R'_{B'}(s') = \begin{cases} 1-\gamma_{B'} & s' \in B' \\ 0 & s' \notin B' \end{cases}, \Gamma_{B'}(s') = \begin{cases} \gamma_{B'} & s' \in B' \\ \gamma & s' \notin B' \end{cases} \tag{5.5}$$

and $\gamma_{B'} = \gamma_{B'}(\gamma)$ *is a function of* γ *such that* $\lim_{\gamma \to 1^-} \frac{1-\gamma}{1-\gamma_{B'}(\gamma)} = 0$. ♦

The proof [89] that the MDP M' satisfies the LTL specification ϕ is reproduced in the supplementary Appendix A. The reward function $R'_{B'}(s')$ and the discount factor $\gamma_{B'}$ embody the characterization of the states of the product MDP M' to account for Büchi and non-Büchi accepting states. However, such a reward function and discount factor formulation provide no distinct categorization within the Büchi accepting states. In order to account for the scenario where certain Büchi accepting states are differentially prioritized, i.e, some Büchi states attract more rewards as compared to others, we define a class of functions which allows quantitative characterization of the Büchi states.

Theorem 5.2. *The class of functions* $\gamma_{B'}(\gamma) = 1 - c(1-\gamma)^\alpha$ *where* $c \in [0,1]$, $0 < \gamma < 1$ *and* $0 < \alpha < 1$ *satisfies the condition* $\lim_{\gamma \to 1^-} \frac{1-\gamma}{1-\gamma_{B'}(\gamma)} = 0$

Proof: By substituting $1-\gamma = t$ and $1 - \gamma_{B'}(\gamma) = g(t)$ it is possible to re-write the condition $\lim_{\gamma \to 1^-} \frac{1-\gamma}{1-\gamma_{B'}(\gamma)} = 0$ as $\lim_{t \to 0^+} \frac{t}{g(t)} = 0$, since as $\gamma \to 1^-$, $t \to 0^+$ for $1-\gamma = t$. Thus, the class of functions $g(t)$ which satisfies $\lim_{\gamma \to 0^+} \frac{t}{g(t)} = 0$ also satisfies $\lim_{\gamma \to 1^-} \frac{1-\gamma}{1-\gamma_{B'}(\gamma)} = 0$.

Let us consider the class of functions $g(t) = ct^\alpha$ where $0 < \alpha < 1$ and c is a constant in the closed interval $[0,1]$. For such $g(t)$, $\lim_{t \to 0^+} \frac{t}{g(t)} = \lim_{t \to 0^+} \frac{t}{ct^\alpha} = \frac{1}{c} \lim_{t \to 0^+} t^{1-\alpha}$. Since $0 < \alpha < 1$, the condition $0 < 1-\alpha < 1$ also holds. Thus, $\frac{1}{c} \lim_{t \to 0^+} t^{1-\alpha} = 0$. Hence the class of functions $g(t) = ct^\alpha$ satisfies both the above conditions. By re-substitution of $1 - g(t) = \gamma_{B'}(\gamma)$ and $1-\gamma = t$, we obtain: $1 - ct^\alpha = \gamma_{B'}(\gamma) \implies 1 - c(1-\gamma)^\alpha = \gamma_{B'}(\gamma)$. ♦

Theorem 5.1 implies that $c \in [0,1]$ can be utilized as a quantitative measure of the Büchi accepting states of the LTL property under consideration, thereby allowing effective steering of the agent's training process. Additionally, since such a class of functions satisfies the above limiting conditions, Theorem 5.1 also implies preservation of all properties of the framework outlined in [89]. We define the function $c : B' \to [0,1]$ in Equation 5.6 to characterize Büchi states where $w_{latency}$ and w_{cost} are the weights assigned to latency and cost of deploying containers respectively on edge servers.

$$c(B') = w_{latency} * 1/(averageLatency) + w_{cost} * 1/(noOfContainers) \tag{5.6}$$

We now prove that our quantitative reward function definition ensures convergence of the model-free algorithm. We first define some properties of the function to establish convergence.

Lemma 5.3. *The class of functions* $1-c(1-\gamma)^\alpha$ *satisfies the condition* $\lim_{\gamma \to 1^-} 1-c(1-\gamma)^\alpha = 1$ *where* $c \in [0,1]$, $0 < \gamma < 1$ *and* $\alpha \in (0,1)$.

Proof: Note that the class of functions $1 - c(1-\gamma)^{\alpha}$ is always continuous since $0 < \gamma < 1$ and $\alpha \in (0,1)$ is a constant while c is also a constant in the closed interval $[0,1]$. Thus, the limit for this class of functions is the value at its evaluation point, i.e., at $\gamma = 1$. Hence, $\lim_{\gamma \to 1^-} 1 - c(1-\gamma)^{\alpha} = 1$. ♦

Theorem 5.4. *The condition* $\lim_{\gamma \to 1^-} 1 - c(1-\gamma)^{\alpha} = 1$ *ensures convergence of the model-free learning algorithm.*

Proof: A value of $\gamma < 1$ ensures convergence of traditional model-free learning algorithms to a unique solution as highlighted in [89]. When the value of $\gamma = 1$, the model-free algorithm may not converge [89]. From Lemma 5.3, $\lim_{\gamma \to 1^-} 1 - c(1-\gamma)^{\alpha} = 1$. As such, when $\gamma = 1$, $\gamma_B(\gamma) = 1$. Thus, in such scenarios the quantitative reward function may not converge as well. However, since $0 < \gamma < 1$ and $\alpha \in (0,1)$ is a constant while c is also a constant in $[0,1]$, the range of the function $\gamma'_B(\gamma)$ always lies in the interval $(0,1)$ since the reward function defined in Equation 5.6 ensures that $c \neq 0$. As a consequence, when $\gamma < 1$, the condition $\gamma'_B(\gamma) < 1$ is satisfied ensuring convergence of the algorithm with the quantitative reward function. ♦

Algorithm 7: Safe Q-Learning

```
1 Input : LTL formula $\phi$, MDP $M$
2 Translate LTL into LDBA $A_\phi$
3 Construct the product $M'$ of $M$ and $A_\phi$
4 Initialize $Q(\{\langle S' \rangle Q'\},\ \Lambda)$ on $M'$
5 for $i$ = 0, 1, ... do
6     $s(t)' \leftarrow$ observe the application state
7     $\lambda \leftarrow \epsilon$-greedy$(s', q')$
8     Observe the next state $s'(t+1)$ and the reward obtained
9     Update $Q(s, \lambda)$ using Equation 5.7 with Quantitative Büchi Rewards and Discount
       Factor as in Theorem 5.1
```

The Q-Learning agent synthesizes the auto-scaling policy using the solution space M' as the Q-Matrix with the above reward function and discount function. The agent executes an auto-scaling decision λ using the ϵ-greedy policy and updates the Q-value in the Q-Matrix for the corresponding action according to Equation 5.7 where ζ is the learning rate.

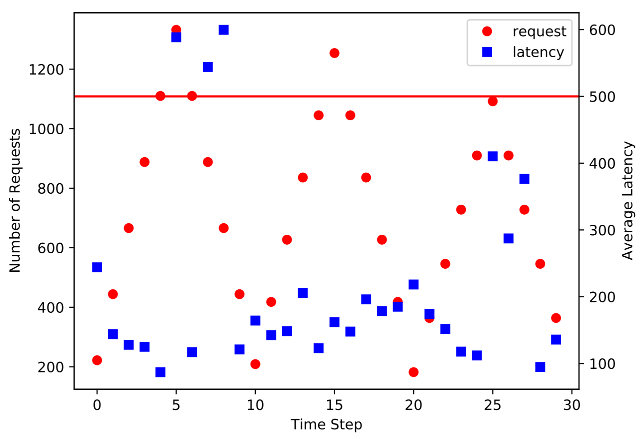
$$Q(s'_t, \lambda) \leftarrow Q(s'_t, \lambda) + \zeta \left[R'_{(B')} + \Gamma_{(B')} \operatorname{argmax}_{\lambda}[Q(s'_{t+1}, \lambda)] - Q(s'_t, \lambda) \right] \tag{5.7}$$

Example 5.2.12. Each state of the Q-matrix is associated with 3 Q-values corresponding to the three auto-scaling decisions. Consider the states $\{\langle 1, 20 \rangle\ q_0\}, \{\langle 1, 30 \rangle\ q_0\}$ and $\{\langle 1, 30 \rangle\ q_1\}$. We consider Q-values from the initial experimental setup of Virtual Machines in Section 5.3

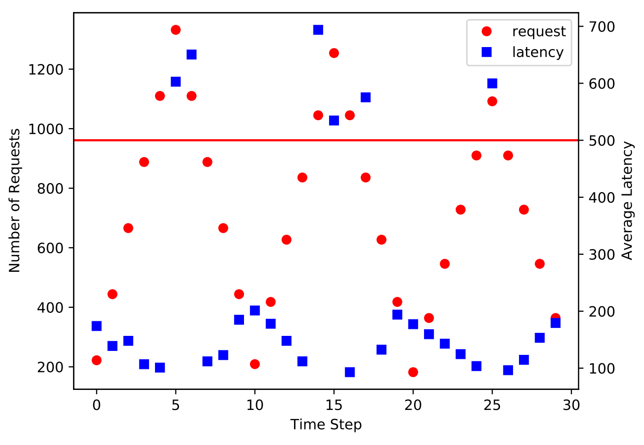
where the discretization interval is set to $10ms$. The corresponding Q-values for each state are [1.52, 1.15, 1.19], [2.18, 1.54, 9.46] and [1.17, 9.89, 7.35]. In state $\{\langle 1, 20\rangle\ q_0\}$, when the agent decides to retain the same container configuration and in the next time step, moves to state $\{\langle 1, 30\rangle\ q_0\}$, the Q-value corresponds to the retain configuration instance, i.e, the first element of [1.52, 1.15, 1.19] is updated as 1.52 + 0.4[0.07 + 0.93 * 9.89 - 1.52] since $\gamma(B') = 1 - (0.3 * (1 - 0.95)^{0.5}) = 0.93$ where $c = 1/30$ (corresponding to the averageLatency component) and the reward is $1 - \gamma(B') = 0.07$. Note that the high value of γ corresponds to a safe decision and hence a higher future discounted reward. Similarly, in state $\{\langle 1, 30\rangle\ q_0\}$ when the agent decides to retain the same container configuration and in the next time step, transitions to state $\{\langle 1, 30\rangle\ q_1\}$, the value of $\gamma(B') = 0$. This is because the LDBA component has transitioned to state q_1 which is a non-Büchi accepting state resulting in 0 reward in accordance with Equation 5.5. ■

5.3 Results and Discussion

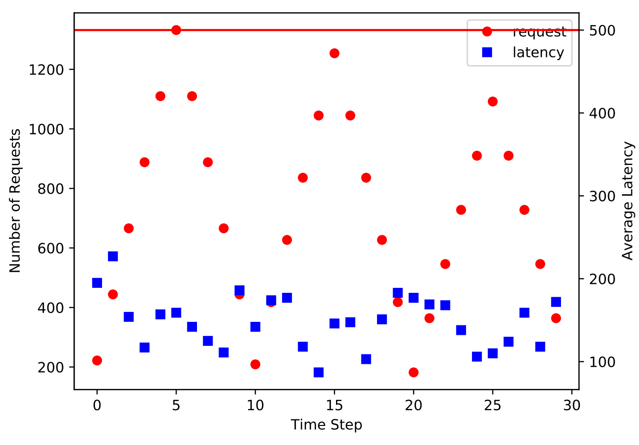
In this section, we discuss our experimental setup and findings. We first evaluate our approach on a Virtual Machine setup with a single application. In order to demonstrate the effect of resource contention in the presence of multiple service requests and the benefits of Safe-RL based auto-scaling, we consider a simplified setup comprising three Virtual Machines, each with two vCPUs, 4GB of RAM and 40GB of Virtual Hard Disk, representative of a single MEC site comprising three edge servers. We use the Social-Network application from the DeathStarBench benchmark suite [48]. Initially, a single container instance of the application is deployed on the test setup. Service invocations are then generated with the *wrk* workload generator included in the benchmark [48]. Figure 5.9 demonstrates the performance of four auto-scaling policies: no auto-scaling, rule-based [10], Reinforcement Learning (RL) based [12] and Safe-RL based (our approach). As expected, when no auto-scaling is employed, several latency violations are encountered. The rule-based auto-scaling policy effectively reduces the overall user latency but is unable to automatically adjust the number of container instances with variation in service requests. The RL based policy is able to automatically learn auto-scaling decisions to adjust the number of container instances in accordance with the service request traffic distribution but encounters a number of latency threshold violations. The Safe-RL based approach ensures adherence to latency requirements as well as automatic adjustment in accordance to the request distribution. We now describe a real-world test-bed based representation of a single edge site with heterogeneous resources and then demonstrate numerical simulation based large scale experiments with inter edge site communication.



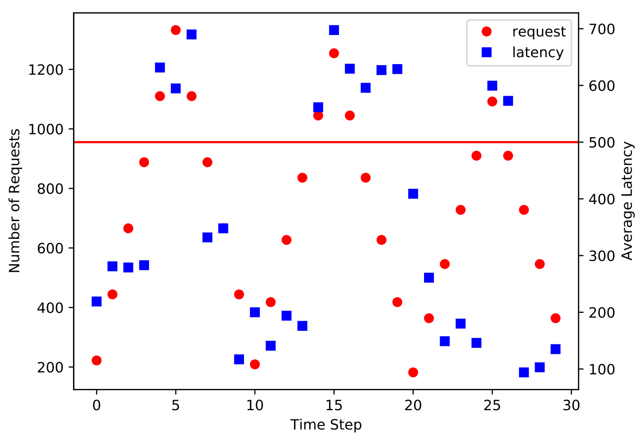
(a) Rule Based



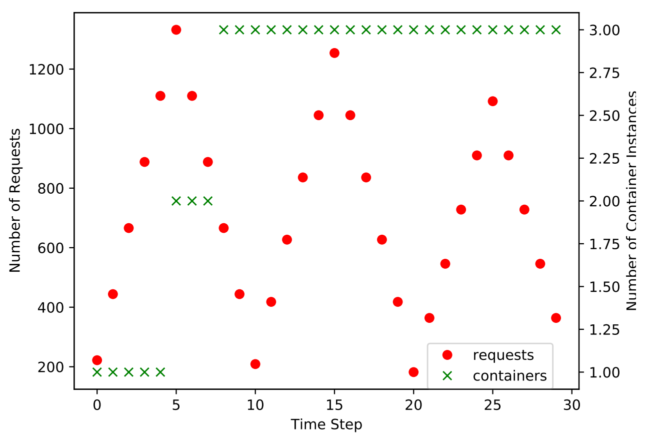
(b) RL Based



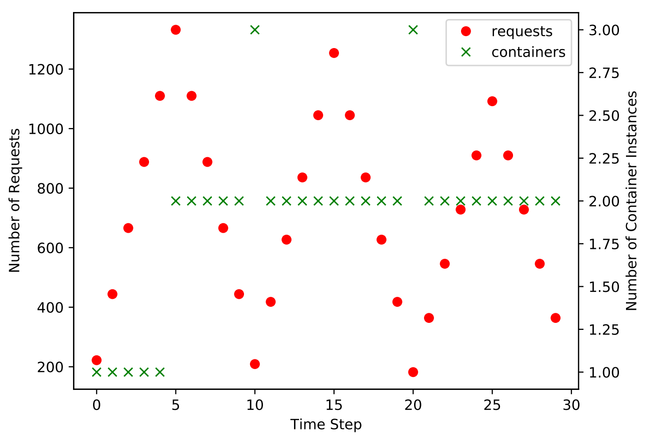
(c) Safe-RL Based



(d) No Auto Scaling

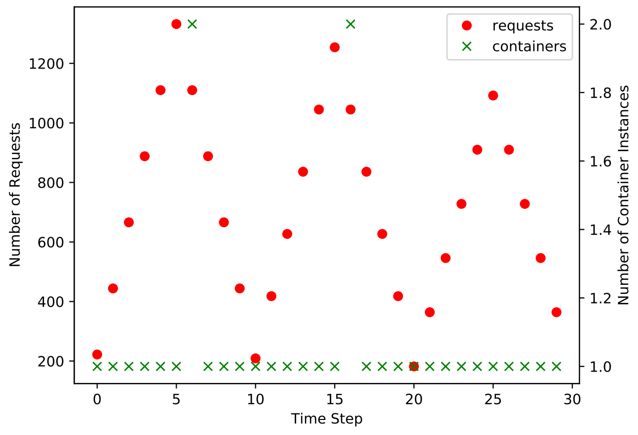


(e) Rule Based - Containers

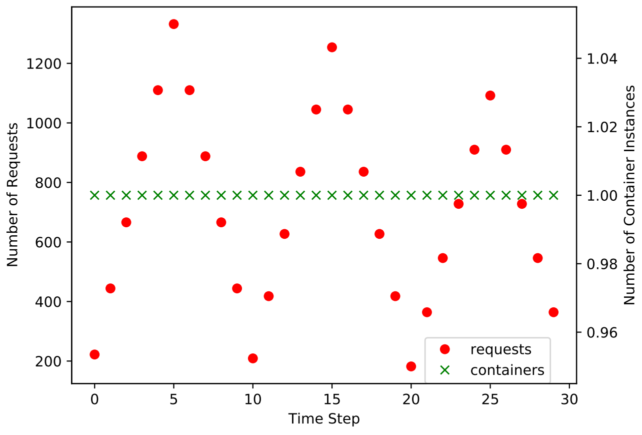


(f) RL Based - Containers

FIGURE 5.9: Average User Latencies Incurred and Number of Container Instances for Various Auto Scaling Policies



(g) Safe-RL Based - Containers



(h) No Auto Scaling - Containers

FIGURE 5.9: Average User Latencies Incurred and Number of Container Instances for Various Auto Scaling Policies

5.3.1 Real-World Test-Bed based Experimentation

5.3.1.1 Applications

We use applications from the DeathStarBench benchmark suite [48], the Social Network application and the Media Microservices application, as shown in Table 5.3. The Social-Networking application is a CPU intensive application. YOLO utilizes both CPU and GPU for object detection [100]. The Media Microservices application is representative of both being CPU intensive as well as graphics oriented. As a consequence, multiple such applications deployed simultaneously can lead to high resource contention. For each application, we randomly generate service invocation requests in the intervals as specified in Table 5.3. The number of service requests is initially increased in the specified interval. Subsequently, the number of service requests follows a decreasing pattern in the specified interval. Each such increasing and decreasing pattern of service request invocation is considered as an episode for the RL agent. Such patterns effectively characterize service variability workloads [10, 11, 12]. We measure the latency of the Social Network and Media Microservices application using the provided workload generator and measure the latency of YOLO as the time taken for the object recognition task to be completed. We compare our Safe-RL based approach with the Rule-Based approach [10]

Application	**Service Request Range**	**Rule-Based Scaling**	**LTL Specification**
Social Network	[1000, 50000]	$averageLatency > 500$: add instance	$G([400, 500) \implies X[300, 400))$
Media Streaming	[1000, 50000]	$averageLatency > 500$: add instance	$G([400, 500) \implies X[300, 400))$
Object Detection	[1, 6]	$averageLatency > 200$: add instance	$G([150, 200) \implies X[150, 100))$

TABLE 5.3: Applications, Number of Service Requests and Corresponding Rules

and a standard RL based approach [12]. The rule-based auto-scaling policies operate on the principle of observing the latency encountered by users and adjusting the number of container instances according to the observed latency [10]. Such mechanisms are thus quite generic and can also be used in MEC. For the Rule-Based approach we consider the rules as summarized in Table 5.3 for each application. We utilize a value of 500ms for the social networking and media applications (non safety-critical) while we utilize a value of 200ms for the safety-critical object-detection application. We initially deploy the applications in a single edge server to simulate contention. The three auto-scaling policies run in the background, monitoring the average latencies, triggering auto-scaling actions as and when necessary.

5.3.1.2 Edge Site Deployment and Configuration:

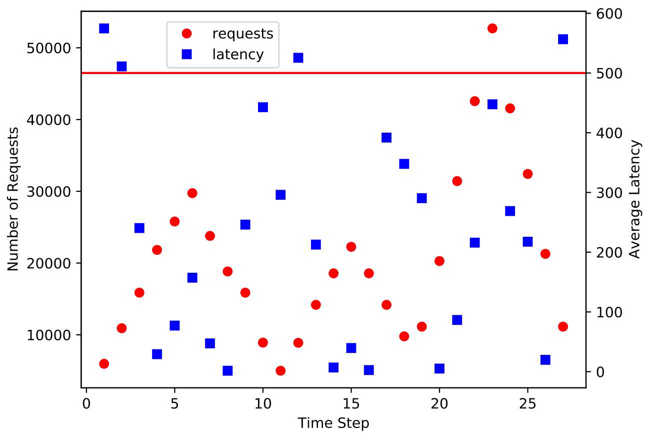
We use an Edge Site with three servers with the following heterogeneous resource configurations: i) an Intel Xeon E5-1650 processor with 128GB of RAM and an NVIDIA Quadro P4000 GPU accelerator, ii) an Intel Xeon E5-1650 processor with 64GB of RAM and an NVIDIA Quadro P4000 GPU accelerator and iii) an Intel Xeon E2 processor with 8GB of RAM. All three machines comprise Standard Magnetic Hard-Disk Drives. We use such configurations to account for the heterogeneous MEC environments and to determine the resulting implications. The third machine does not incorporate a GPU accelerator and is used for deploying both the Social Network and Media Microservices application containers while the two other machines include GPU accelerators and are utilized for all three applications. Since our auto-scaling approach is distributed, such a setup is representative of auto-scaling policies running at each edge site. For this particular set of experiments, we consider a single edge site with communication between edge sites, i.e, service requests invoked within this particular edge site can only be provisioned by servers within the edge site. Such a setup conforms to our MDP model in Section 5.2.3 when $neighbourhoodCount$ is NULL. As a consequence, there is no additional access latency via the backbone network.

5.3.1.3 Training the RL agent

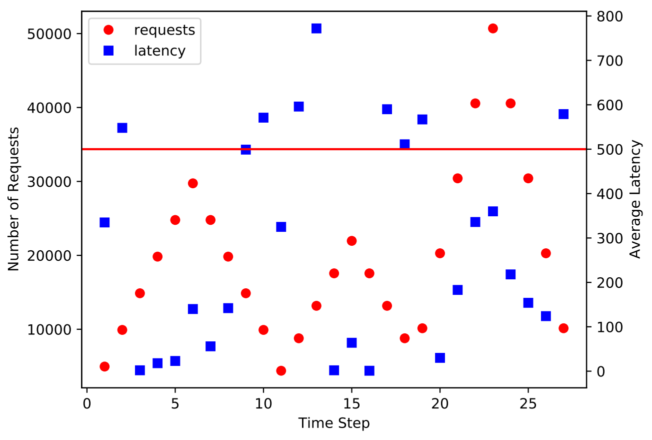
Since we make use of a discrete time slotted model, the training process of the RL agent is crafted to restrict decision making, state changes and reward observations at discrete time intervals. Safe Q-Learning Algorithm 7 initializes the state space of the MDP agent with the composition of the original Auto-Scaling MDP and the LDBA corresponding to the LTL formula. The initial state is set to $noOfContainers = 1$, since, initially, we consider only a single container instance while the corresponding LDBA component is initialized to the start state of the LDBA, i.e, q_0. In our experiments, we set the discrete time slot duration to 2 seconds. We consider a slot duration of 2 seconds to consider the evolving MEC environment where service request traffic invocations can vary rapidly. The Q-values are all initialized to 0. At the start of each discrete time step, the RL agent executes one of the three auto-scaling decisions. The container deployment configuration is updated to reflect the decision of the agent. Then, a random number of service requests are invoked for a duration of 2 seconds, the duration of the discrete time slot. The average latency of the service requests invoked is calculated. The workload generator of the Social Network and Media Microservices applications return average latencies of the service requests invoked in the time interval as specified by the duration parameter. For YOLO, we calculate the average latency with respect to the number of object recognition tasks invoked and the respective observed latencies. After the duration of the discrete time step, the reward for the decision undertaken by the RL agent is calculated based on the new state thus obtained and the Q-values updated. We use $\zeta = 0.4$ as the learning rate, $\gamma = 0.95$ as the discount factor, $\alpha = 0.5$ for the discount function $\gamma_{B'}$ and $w_{latency} = w_{cost} = 0.5$ as the parameters for the training process. The discretization interval is set to $100ms$ for the Social Network and Media Streaming applications and $50ms$ for the Object Detection application.

5.3.1.4 Experimental Results and Analysis

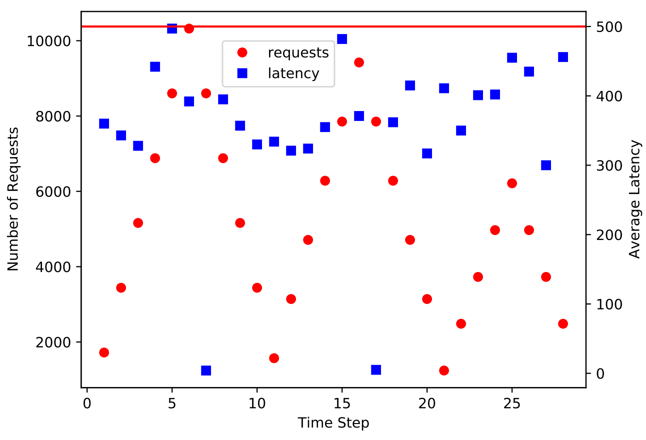
Figures 5.10, 5.11 and 5.12 show the performance of the three auto-scaling policies for the Social Network, Media Streaming and YOLO Object Detection applications respectively. Similar to the results obtained in the Virtual Machine based experimental setup in Section 5.1, the standard RL based auto-scaling policy incurs several latency violations in case of the Social Network application as shown in Figure 5.10b. The standard RL based auto-scaling policy mostly incurs such violations when the number of service invocations is on the higher side. The Safe-RL based policy avoids such pitfalls. Both the RL based and Safe-RL based policy are able to automatically adjust to service request invocation variability unlike the Rule-based autoscaler as inferred from Figures 5.10e and 5.10f. The rule based policy is unable to dynamically adapt as depicted in Figure 5.10d. It is however interesting to note that unlike the case of the single application scenario with Virtual Machines, the number of container instances spawned by both RL policies are not correlated with the number of service request invocations for the Social Network application. Such a scenario occurs since resource contention from service request invocations of other applications impact the overall average latency when considering multiple applications. The RL agents which trigger auto-scaling decisions in accordance with the observed average latency of the applications, are able to adapt automatically to such variations.



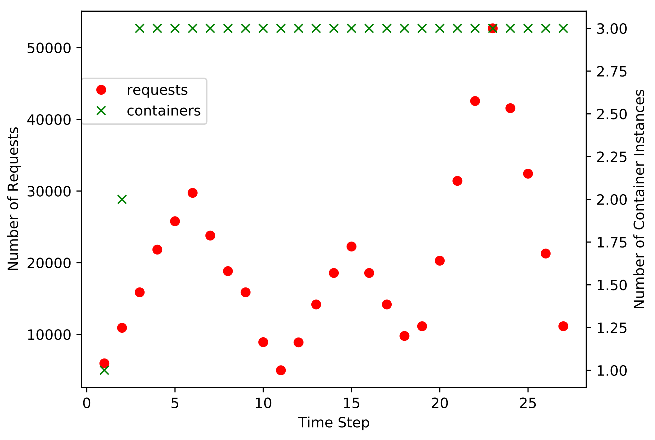
(a) Rule Based Policy



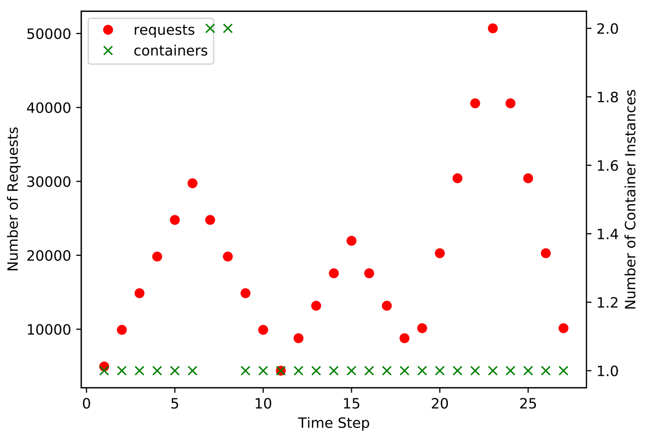
(b) RL Policy



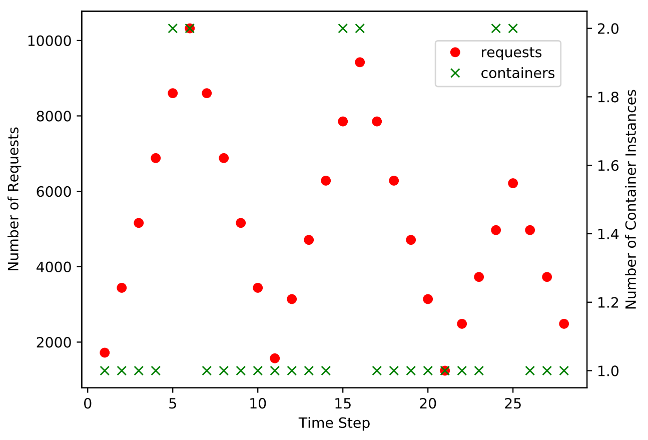
(c) Safe-RL Policy



(d) Rule Based Policy

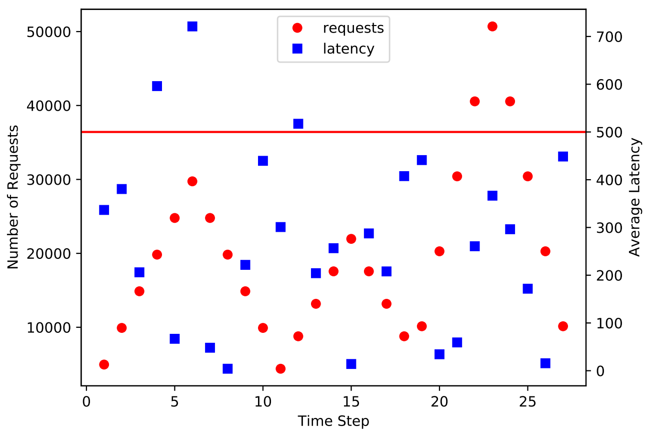


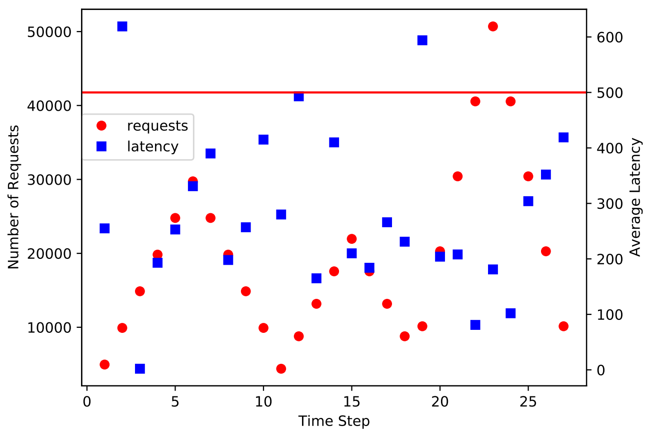
(e) RL Policy

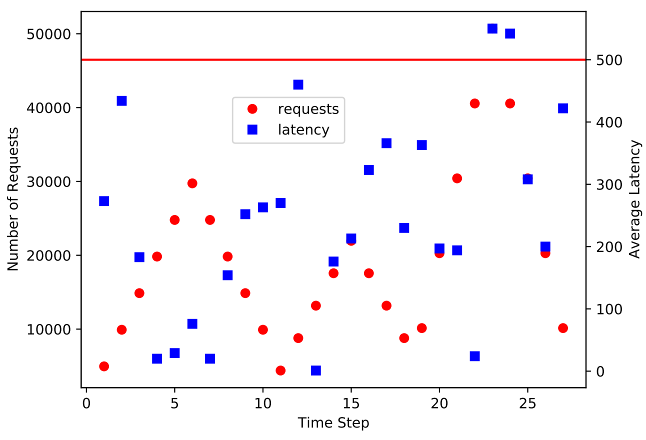


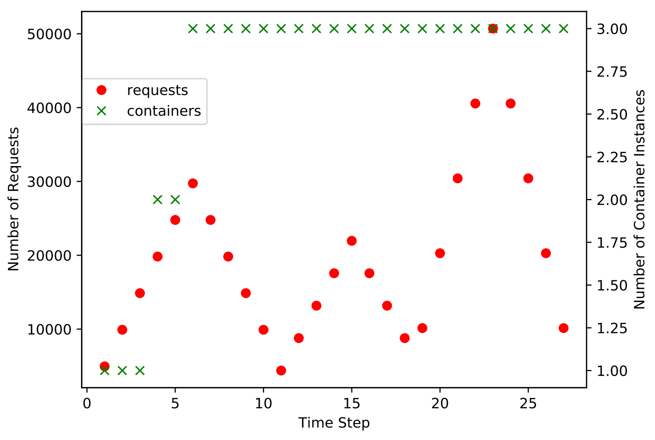
(f) Safe-RL Policy

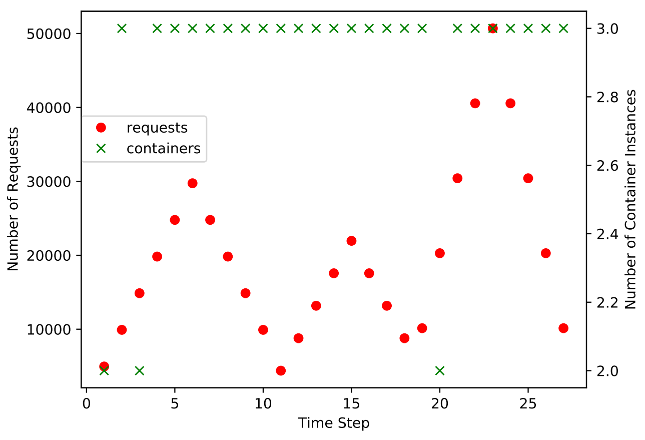
FIGURE 5.10: Average Latencies and Number of Containers for Social Network Application


(a) Rule Based Policy


(b) RL Policy


(c) Safe-RL Policy


(d) Rule Based Policy


(e) RL Policy

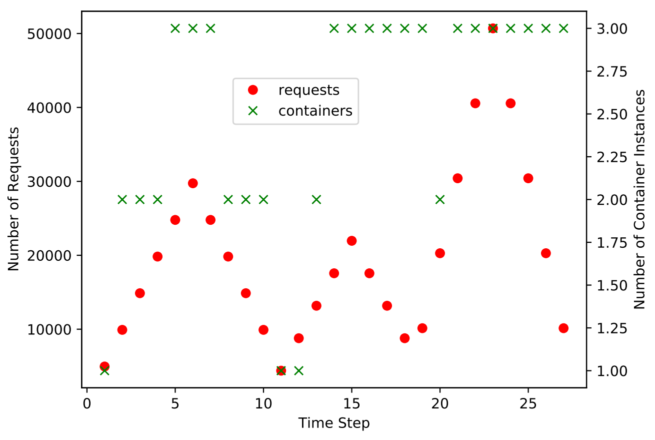
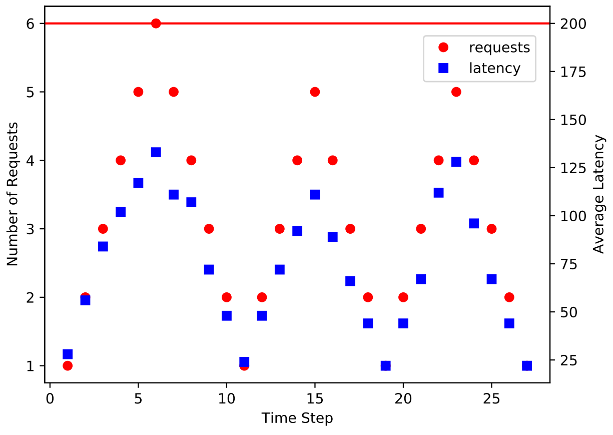
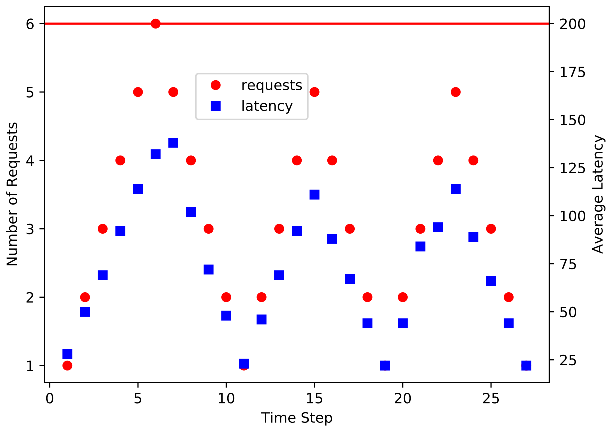

(f) Safe-RL Policy

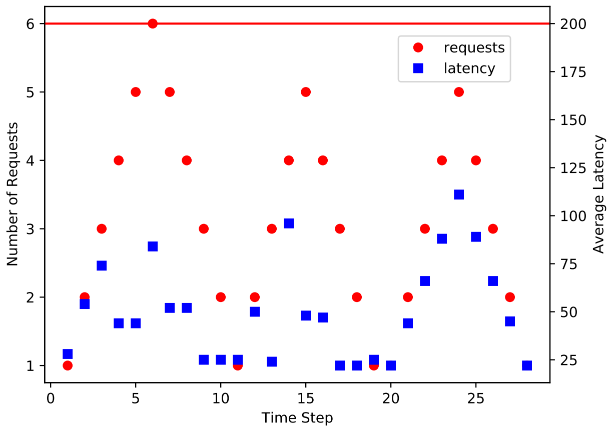
FIGURE 5.11: Average Latencies and Number of Containers for Media Services Application



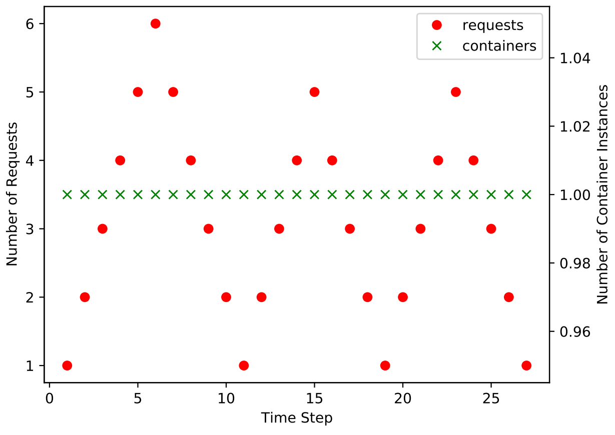
(a) Rule Based Policy



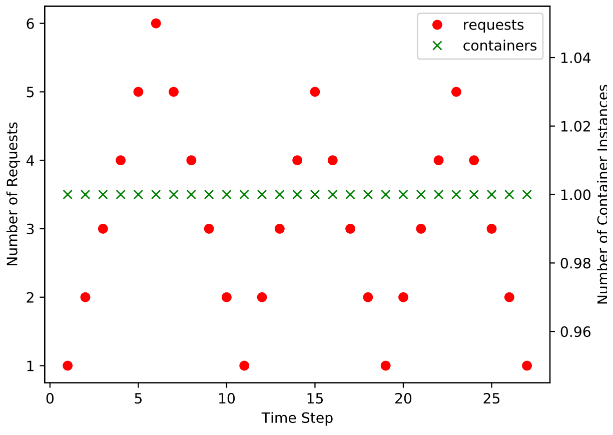
(b) RL Policy



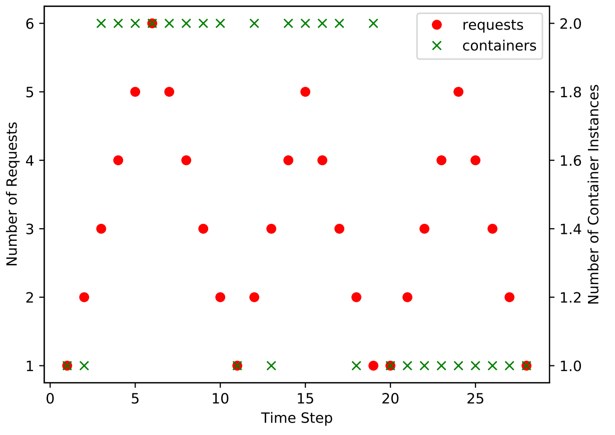
(c) Safe-RL Policy



(d) Rule Based Policy



(e) RL Policy



(f) Safe-RL Policy

FIGURE 5.12: Average Latencies and Number of Containers for YOLO Object Detection

Figures 5.10b and 5.10c demonstrate the comparative average latencies incurred with the Standard RL based approach and the Safe-RL based approach. Note that the average latencies incurred when utilizing the Safe-RL based approach is higher in several instances even with much lower number of service requests as compared to the Standard RL based approach. However, in all such instances, the average latency is within the $500ms$ threshold for the Safe-RL approach. Such scenarios confirm concordance with our objective of deriving auto-scaling policies ensuring adherence to latency specifications as opposed to overall latency minimization. Additionally, consider the observed latencies as a result of the Safe-RL policy as depicted in Figure 5.11c for the Media Streaming application. In Figure 5.11c, the Safe-RL auto-scaling policy incurs two latency violations when the average latency incurred is above the threshold requirement of $500ms$. Such a scenario occurs only when the maximum number of permissible containers (3 in this case), have already been provisioned. As such, the auto-scaling agent could not have provisioned additional resources to circumvent the violation. Figure 5.12 demonstrates the average latency and the number of containers provisioned for the Object Recognition application. Safe-RL provisions a higher number of container instances in this case, however achieves a lower overall average latency amongst the service requests.

5.3.1.5 Utilizing Maximum Latency Values

In the experimental setup for the social network application with virtual machines, we utilized the average latency of users to execute auto-scaling decisions. Table 5.4 lists a comparison between the number of container instances utilizing average latency and maximum latency of end users from the virtual machine setup replicated. The nature of the latency plots incurred when considering maximum latency are similar to the plots depicted with average latency and are omitted due to brevity. However, the number of container instances spawned by each RL based algorithm is higher when considering the maximum incurred latency as highlighted in Table 5.4. More notably, the Safe-RL based approach spawns an even higher number of container instances, however incurs lowest latency threshold violations. Thus, utilizing the maximum latencies associated with service requests in general results in a higher number of provisioned container instances.

	RL Based		**Safe RL Based**	
Requests	**Latency Value Utilized**			
	Avg	**Max**	**Avg**	**Max**
	No. Of Containers			
222	1	2	2	3
444	1	2	2	3
666	1	3	2	3
888	1	3	2	3
1110	1	3	2	3
1332	2	3	3	3

Table 5.4: Effect of Max Latency

	% of Violations	**Average No. of Containers at each edge site**	**Auto-Scaling Decision Running Time (in miliseconds)**
Rule-Based	15.77	3.48	0.0059
RL Set 1	29.21	2.66	0.2911
Safe RL Set 1	14.54	3.09	0.3340
RL Set 2	26.11	2.88	0.2813
Safe RL Set 2	12.21	3.37	0.3088

Table 5.5: Large Scale Simulation Based Experimental Setup

5.3.2 Large Scale Numerical Simulation Experiments

5.3.2.1 Experimental Setup

We use numerical experiments to simulate large-scale scenarios. We use locations from the "Existing Commercial Wireless Telecommunication Services Facilities in San Francisco" [98] dataset as MEC edge site locations. Each edge site is associated with 4 edge servers. For each edge site, we consider $\chi = 2$, i.e, we consider inter edge site communication within 2 hops. For each inter-edge communication, we assign a random additional access latency between 0 to 50ms for edge sites within a 1-hop access and between 0 to 100ms for edge sites within a 2-hop access. We generate service request invocations within each edge site randomly. We consider the same three applications we utilized for the real-world test-bed setup. We utilize the latency values generated from a single server on the test-bed application setup in Section 5.3.1 with the three applications being accessed simultaneously. We utilize the actually measured value of $averageLatency$ from the test-bed in scenarios when the auto-scaling policy results in a single server. We utilize the value of $averageLatency/2$ for scenarios where the auto-scaling policy results in two servers and a value of $averageLatency/numberOfServers$ in general to simulate multiple MEC servers within an edge site. The other parameters are set identical to Section 5.3.1. We perform two sets of experiments, Set 1 with $w_{latency} = 0.5$ and $w_{cost} = 0.5$ and Set 2 with $w_{latency} = 0.7$ and $w_{cost} = 0.3$ to demonstrate the impact of the reward mechanism. The results are summarized in Table 5.5.

5.3.2.2 Results and Analysis

As can be inferred from Table 5.5, the percentage of violations incurred by the Safe-RL approach is lowest demonstrating the effectiveness of our approach. The Rule-Based approach incurs lower violations as compared to the Standard-RL based approach but it spawns a much higher number of containers on an average. Note that the percentage of violations for the Safe-RL based approach is lower for Set 2 experimental scenarios as compared to Set 1 experiments. Such a scenario occurs since the $w_{latency}$ is assigned a value of 0.7 in Set 2 experiments specifying a higher weightage towards the latency value obtained. However, in Set 2 experiments, a higher average number of container instances are spawned by the Safe-RL approach. Such scenarios depict the trade-off involved in the weighted reward mechanism for latency and cost associated with the number of container instances spawned. The same scenario occurs with the Standard-RL based approach since we utilize the same rewarding mechanism with the Standard-RL approach, however, with the reduced state space without incorporating the LDBA based composed MDP and the Safe-RL discount function. Note that the Rule-Based approach does not include such a rewarding mechanism and hence only one set of results is listed. The Rule-Based approach incurs the lowest running time while the Safe-RL based approach incurs the highest. Such scenarios occur since a Rule-Based approach takes a simple decision making approach after observing the incurred latencies. The Standard-RL and Safe-RL based approaches incur higher running time since they perform additional lookup operations on the state space table to execute the auto-scaling decisions.

5.4 Conclusion

In this chapter, we propose a Safe-RL based horizontal auto-scaling policy that maximizes the probability of adherence to application specific latency requirements. We model the MEC environment and auto-scaling decision making using an MDP. We represent latency requirements in LTL and introduce a quantitative reward mechanism to characterize the MDP composed with the LTL specification and demonstrate the convergence properties of the mechanism. We utilize a test-bed setup with benchmark applications to demonstrate how such an approach outperforms traditional RL based approaches to alleviate resource contention arising out of multiple applications accessing shared resources simultaneously. The service allocation, service placement and auto-scaling policies we have explored so far work agnostic to the possibility of failures in the MEC ecosystem. In the next chapter, we study the implications of MEC server failures.

Chapter 6

Fault Recovery in MEC

6.1 Introduction

MEC servers are more susceptible to failures than their cloud counterparts due to their large scale and distributed nature [15]. In the event of an MEC server failure, the containers which are allocated to the faulty server have to be re-initialized at other MEC servers. A fault-recovery procedure determines the MEC servers to be utilized to re-initialize these containers. Indeed, we have a number of possibilities for this design, considering the different factors (e.g. reliability, latency, resource contention etc.). To the best of our knowledge, the issue of fault-tolerance and in particular, fault-recovery strategies in containerized MEC environments has been relatively less studied. Designing fault-recovery strategies in MEC is a complex task due to: i) the distributed nature of MEC servers with each server catering to specific geographical areas with varying access latencies; ii) the possibility of further admissible failures at the servers where the containers are to be re-initialized; iii) the large number of recovery possibilities for re-initializing containers and iv) a myriad of real-time and non real-time application containers competing for shared resources. In this chapter, we derive fault-recovery strategies by considering all the above scenarios.

In this chapter, we propose a two-fold fault-recovery strategy, namely, local recovery and global recovery. We use the local recovery procedure for real-time applications with high priority within a small geographical area while the global recovery procedure is used for non-critical applications within larger geographical areas or in scenarios when the local recovery strategy cannot be used to determine any guarantees on the recovery process. We use formal methods for the synthesis of local recovery strategies to derive probabilistic guarantees. We model the fault-recovery procedure using a Markov Decision Process. We also model the failure scenarios of

MEC servers as another MDP. MDPs allow effective characterization of the non-deterministic choices of server availability for designing the fault recovery strategy while also allowing us to characterize the stochastic nature of server failures. We capture the complex interactions between the MEC servers and the fault recovery procedure as a Turn-Based Stochastic Multi-Player Game [38]. We formalize the fault-recovery strategy synthesis objectives as expressions in Probabilistic Alternating Temporal Logic [37] and use a probabilistic model checker [38] to synthesize recovery strategies while additionally considering the possibilities of multiple subsequent failures. Further, we design a heuristic based global recovery strategy considering server failure probabilities where local recovery is infeasible. Large-scale simulations with real-world datasets demonstrate the effectiveness of our approach. In summary, the main contributions of this chapter are:

- We model the MEC environment as a composition of several MDPs and represent the interaction between these MDPs as a Stochastic Multiplayer Game.
- We demonstrate how the latencies resulting out of resource contention amongst multiple applications can be modeled with the reward formulation of the SMG.
- We demonstrate how recovery objectives capturing user-perceived latencies can be specified in Probabilistic Temporal Logic and utilize these formal specifications to synthesize fault-recovery strategies.
- We design a two-fold recovery strategy: local recovery to synthesize strategies with probabilistic guarantees characterizing small geographical areas and a heuristic based global recovery over larger geographical areas.
- We experimentally demonstrate the effectiveness of our approach on benchmark datasets.

The rest of this chapter is organized as follows. Section 6.2 presents a motivating example. Section 6.3 discusses the modeling framework. Section 6.4 descibes the generation of fault-recovery strategies. Section 6.5 details the obtained results. Section 6.6 concludes the chapter.

6.2 A Motivating Example

In this section, we present a motivating example to explain the problem context addressed in this chapter. We consider the MEC scenario depicted in Figure 6.1 as an example to explain our problem context. Each edge server is associated with a failure probability depicted along

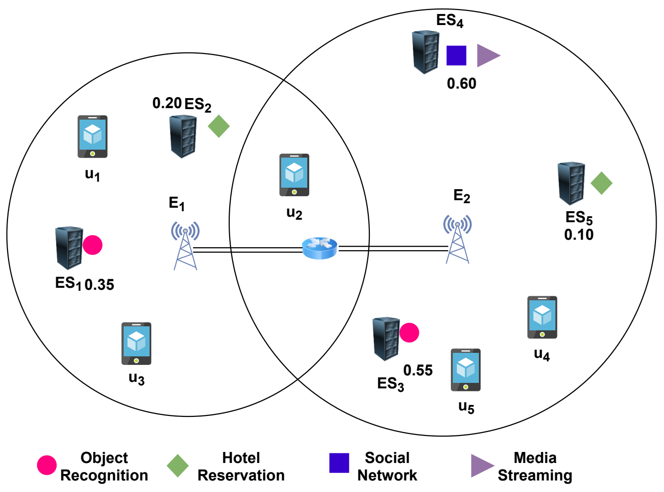


FIGURE 6.1: Multi-Access Edge Computing Scenario

Edge Site	Edge Server	Users
E_1	ES_1, ES_2	u_1, u_2, u_3
E_2	ES_3, ES_4, ES_5	u_2, u_4, u_5

Server	Applications
ES_1	Object Recognition
ES_2	Hotel Reservation
ES_3	Object Recognition
ES_4	Social Network, Media Streaming
ES_5	Hotel Reservation

TABLE 6.1: MEC Scenario Configuration

with the server. For example, ES_4 has a probability of failure 0.6. In this chapter, and in the previous chapter, we assume that all microservices associated with an application instance are deployed on the same edge server. We assume that initially, application service containers are deployed on MEC servers randomly. The example scenario in Figure 6.1 depicts a snapshot of such a scenario in which some application containers have been deployed on the MEC servers with several users connected to these servers and utilizing the services hosted therein. Services deployed on a server are assigned priority levels with real-time services assigned the highest priority. Whenever a failure occurs at an MEC server, the containers deployed therein have to be re-initialized at other MEC servers to ensure continuity of service. We explain in the following subsections how our recovery strategy can help mitigate some of the latencies prevalent in traditional fault tolerant strategies.

6.2.1 Failure Aware Strategy

Let us assume server ES_4 fails as depicted in Figure 6.2. The containers allocated to ES_4 must now be re-initialized at a different server. A traditional failure-aware Fault-Tolerant strategy [16] selects the most reliable server to re-initialize the container. In this scenario, since ES_5 is the highest reliable server (with the lowest failure probability value 0.10) associated with the edge site E_2, the failure-aware strategy selects ES_5 as the server for the containers to be re-initialized. However, as a result of re-initializing both containers at ES_5, users may incur an added overall latency arising out of resource contention with both the Social Network and Hotel Reservation services co-located at ES_5.

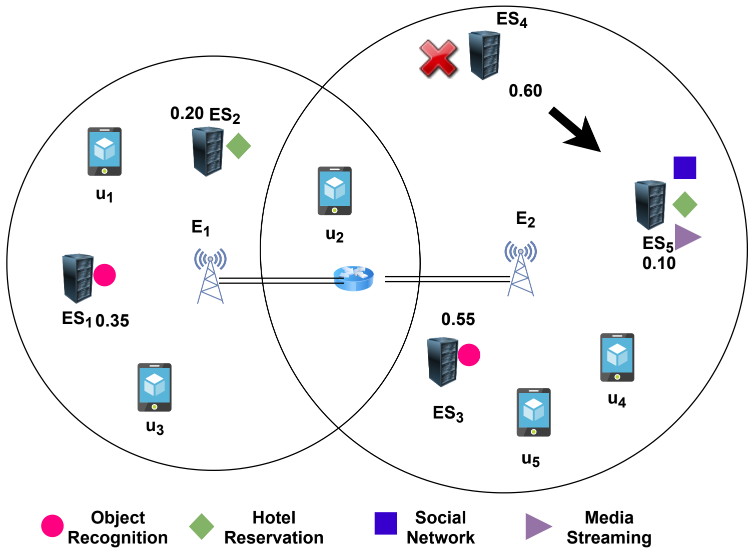


FIGURE 6.2: Failure Aware Fault-Recovery

6.2.2 Performance Aware Strategy

A performance-aware strategy employing load-balancing [118] initializes one of the containers at ES_3 and another container at ES_5, thereby effectively mitigating the shared resource contention. However, such a strategy is oblivious to the possibility of server failures. Consider the scenario in Figure 6.3 arising out of executing the performance-aware load balancing recovery strategy. Server ES_3 has a high probability of failure, i.e., 0.55. Thus, in the event of a failure, the container would need to be re-initialized once again at a different server leading to additional latency overheads.

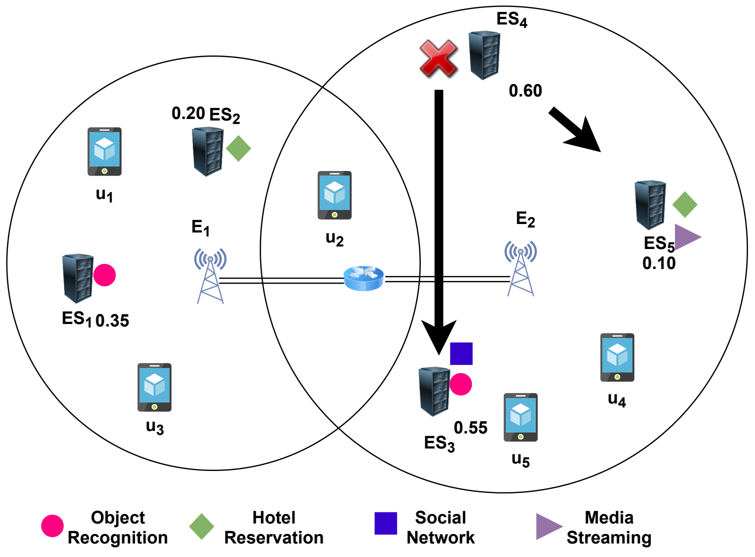


FIGURE 6.3: Performance Aware Fault Recovery

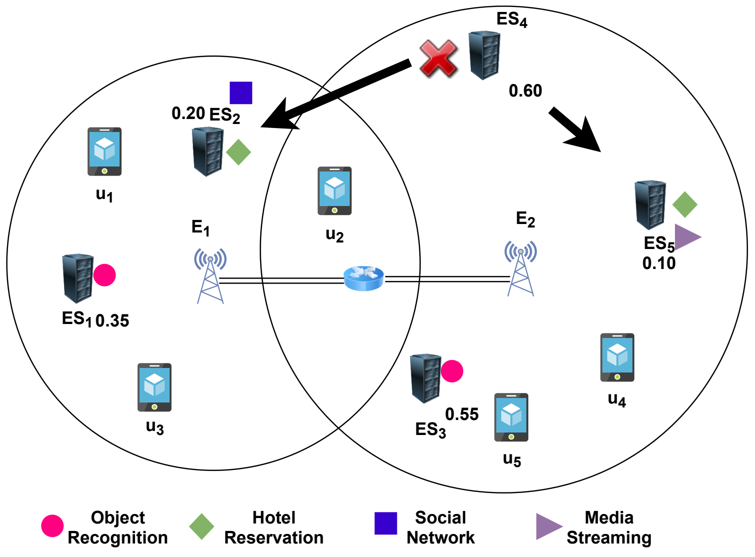


FIGURE 6.4: Stochastic Game Fault Recovery with Low Reliable Servers

6.2.3 Stochastic Multi-Player Game Based Failure and Performance Aware Strategy: Our proposal

To overcome the limitations of the above strategies, we propose a Stochastic Game based Fault-Recovery Strategy, discussed in detail in Section 6.3. When server ES_4 fails, our strategy re-initializes one container at ES_5 and the other container at ES_2 considering both the relative performance impact of running both containers at a single server versus the possibility of further failures at other servers. Additionally, unlike both the failure-aware and performance-aware strategies which are agnostic to the nature of services, our strategy re-initializes the Media Streaming Service before the Social Network Service considering the fact that Media Streaming is a real-time service and attributes more immediate attention to its deployment. This is formalized later in our discussion in Section 6.3.1. Note that our strategy allocates one of the containers to ES_2 which is not associated with edge site E_2. Even though such a strategy implies communication between the re-initialized container at ES_2 with its users via the backbone communication network, it successfully mitigates the higher latencies encountered in the event of a failure of ES_3 whose reliability is lower.

Additionally, consider a similar scenario in Figure 6.5 where ES_3 is instead a highly reliable node with the probability of failure being 0.15. In such a scenario, our Stochastic Game based strategy re-initializes one container at ES_5 and the other container at ES_3, since the probability that ES_3 fails is lower. While for the simple system at hand, we have a limited number of possibilities to consider to address the failure, this is not the case for a system at scale, for which deriving the recovery strategy is a non-trivial task. In a realistic scenario, in addition to possibilities of multiple failures, resource contention, and differentially prioritized services, each

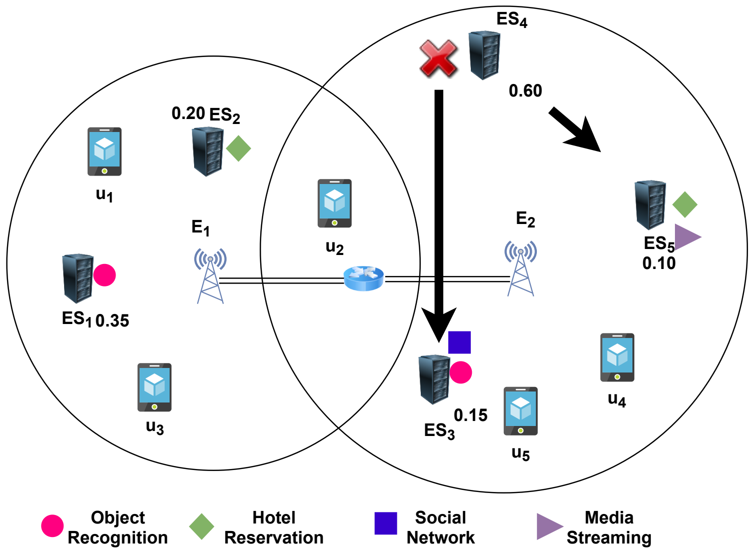


FIGURE 6.5: Stochastic Game Fault Recovery with High Reliable Server

application container is associated with a memory requirement whilst each server has a fixed memory capacity which determines whether a container can indeed be initialized at a particular server [119]. Designing a fault-recovery strategy encompassing all such scenarios is a complex task. We thus use MDPs to model the stochastic nature of failures, the choices of failure-recovery strategies using non-determinism, and the entire space of container-server allocation bindings taking into consideration memory characteristics of servers and application containers. We capture their interactions as a Stochastic Multi-Player Game. The SMG characterizes local recovery within an edge site for highly prioritized services. In scenarios where local recovery is infeasible, we design a heuristic based global recovery characterizing multiple edge sites within a χ-hop neighbourhood. In the following sections, we describe in detail our modeling and strategy synthesis approach.

6.3 Formal Model of Fault-Recovery

In this section, we first formally define the fault-recovery strategy synthesis problem. We then describe our model of turn-based stochastic multiplayer games using which we derive the fault-recovery strategy.

6.3.1 Problem Formulation and Assumptions

The MEC system follows a discrete time-slotted model as in our earlier chapters. The location of the users and services deployed at edge servers are updated at the beginning of each discrete time slot of μ seconds. The failure status of each server is updated at the beginning of each discrete time slot of ν seconds. Our discrete time slotted model is illustrated in Figure 6.6.

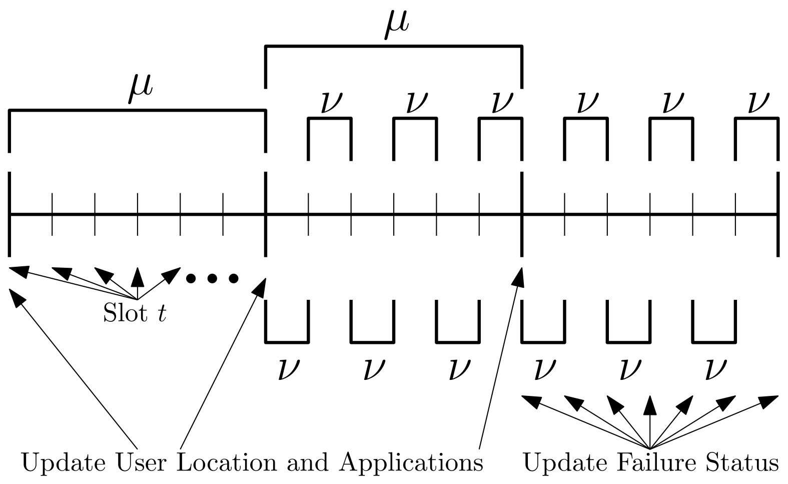


FIGURE 6.6: Discrete Time Slotted Model with Failure Updation

We consider the following in this chapter.

- An MEC system comprising n MEC edge sites, $E = \{E_1, E_2, \ldots E_n\}$, where each site is represented by its latitude and longitude coordinates and associated with a set of servers $ES_j = \{ES_1, ES_2, \ldots, ES_m\}$ as considered earlier.
- We assume edge sites do not share edge servers, i.e., $ES_x \cap ES_y$ is empty, where $1 \leq x, y \leq n, x \neq y$.
- Each server $ES_x \in ES_j$ is associated with a probability of failure ES_x^{Π} and fixed memory capacity $ES_x^{memcapcaity}$.
- For each E_j, we consider k-fault tolerance where at most $k < m$ servers can fail simultaneously at any discrete time slot t.
- Similar to the previous chapter, we assume all microservices of a particular application are deployed together at an edge server. Henceforth in this chapter as well, we refer to an application container as the set of all microservices associated with the application.
- Each server $ES_x \in ES_j$, is associated with a set of application containers $C_{ES_x}^t = \{c_1, c_2, \ldots c_q\}$ at any discrete time instant t with each container having a memory requirement $mem(c_i)$.
- We represents by c_i all the application containers associated with the i-th application.
- Each application container $c_i \in C_s^t$ requires an initialization time $c_i^{inittime}$.

- Each container $c_i \in C^t_{ES_x}$ is associated with a set of users $U^t_{c_i}$ availing of its services.
- Each container $c_i \in C^t_{ES_x}$ is associated with a priority ω_i according to the nature of the application, i.e., real-time applications are assigned higher priority values while non real-time applications are assigned lower priorities.
- Prior studies [120, 121, 122] dealing with container-based applications have established that it is possible to determine apriori which applications when deployed on the same server lead to high resource contention. In our work, we assume such an apriori characterization is available.

A server $ES_x \in ES_j$ is defined as *faulty* at any discrete time step t if the computational facilities of the server is unavailable to any of the users. At any discrete time slot t, upon failure of $ES_x \in ES_j$, the objective of the fault-recovery strategy is to re-initialize the containers $C^t_{ES_x}$ allocated to ES_x within τ discrete time steps, such that the additional latency incurred by the users $U^t_{c_i}$ is minimized. To consider the relative criticality of the applications while re-initializing containers, for a faulty server ES_x, we consider the top κ real-time application services with highest priorities which need to be re-initialized before the other application services. We thus resort to the following two-fold strategy:

- *Local Recovery:* generates fault recovery strategies with probabilistic guarantees from an MDP model within an edge site involving the κ highest priority real-time services when recovery is indeed possible within τ discrete steps.
- *Global Recovery:* a heuristic, used for lower priority services and scenarios where local recovery is infeasible within τ steps for the κ real-time services.

We model the local recovery strategy synthesis problem as a composition of several MDPs. For each edge site E_j, we consider the following MDPs:

- the MDP model of stochastic server failures depicting $\{1, 2, \ldots k\}$ further admissible failure scenarios.
- the MDP model of the fault-recovery procedure which incorporates non-deterministic choices of servers for container re-initialization.
- the MDP model of the faulty server within an edge site, i.e., the server from which containers are to be re-initialized.

- the MDP models of servers which are live, i.e., servers at which the containers can be re-initialized.
- the MDP model of a container representing the required time for initialization.
- the MDP model of the timer representing each container's initialization time.
- the controller MDP which orchestrates the turn-based game between the server and the fault recovery procedure.

Whenever a failure occurs at an edge site, all the above MDP models are initialized corresponding to that particular edge site. The initial state of the MDP model of stochastic server failures reflects the state of the faulty server along with all possible further k-admissible stochastic failures. The MDP model of the faulty server is initialized with the containers to be re-initialized in order of their priority. The MDP models of the live servers are initialized with their remaining memory capacity. The timer MDP denotes the number of discrete time steps within which all the containers are to be re-initialized. During the recovery process, the MDP model of the fault-recovery procedure non-deterministically selects the servers where each container is to be re-initialized. The controller MDP orchestrates a turn-based game between the MDP model of the fault-recovery procedure and the MDP model of stochastic server failures. We explain each model in the following sub-sections and then integrate these models using MDP composition.

6.3.2 Stochastic Model of Server Failures

For each edge site E_j, we construct an MDP to represent all possible further admissible k-server failures once a particular server has failed. We use a representation similar to [22] of all possible server failure combinations. The states of the MDP represent the failure status of each server associated with E_j. Faulty servers are indicated with circles around their identifiers.

Example 6.3.1. Figure 6.7 depicts all further admissible k-server failures for an edge site with 4 servers ES_1, ES_2, ES_3 and ES_4 when the server ES_1 has initially failed. The state $\boxed{\textcircled{ES_1}, ES_2, ES_3, ES_4}$, the initial state, represents ES_1 having failed while ES_2, ES_3 and ES_4 are all active. We assume $k = 3$, i.e. at most 3 servers can fail in this example. The state $\boxed{\textcircled{ES_1}, \textcircled{ES_2}, ES_3, ES_4}$ represents the scenario with ES_1 and ES_2 having failed while ES_3 and ES_4 are active. ■

Transition Representation: There are two types of transitions describing stochastic failures:

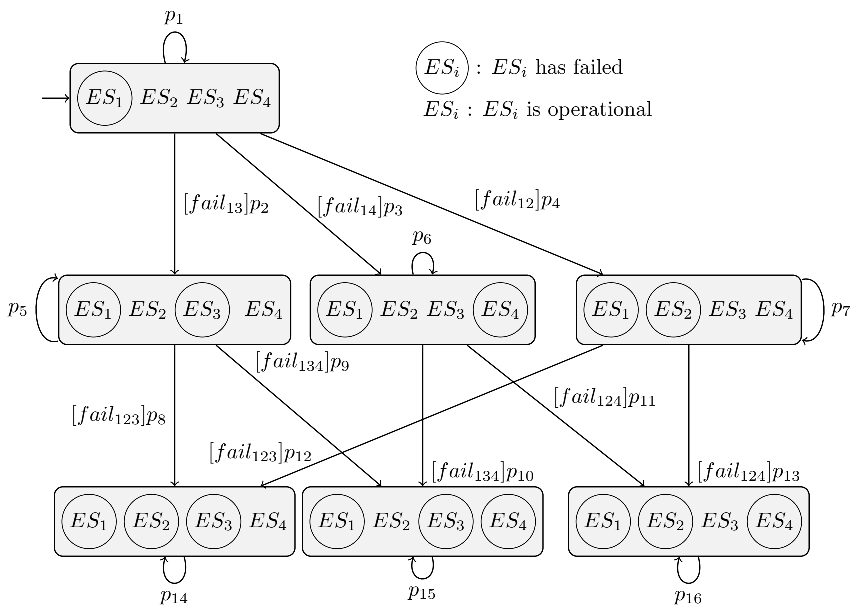


FIGURE 6.7: Stochastic Model of Server Failures

- Transitions representing no further admissible failures: From a particular state, such transitions represent scenarios when no other server fails. Such transitions are thus represented by self-loops in each state.
- Transitions representing further admissible failures: Such transitions depict scenarios where additional server failures occur leading to a change in the state of the MDP.

For each state, we define a probability distribution over all outgoing transitions from the state, representing the probabilities of server failures.

Example 6.3.2. The self-transition in state $\boxed{\enclose{circle}{ES_1}, ES_2, ES_3, ES_4}$ denotes the situation that ES_1 has failed in the current state, and in the subsequent discrete time step, no other server failure occurs. The transition from $\boxed{\enclose{circle}{ES_1}, ES_2, ES_3, ES_4}$ to $\boxed{\enclose{circle}{ES_1}, ES_2, \enclose{circle}{ES_3}, ES_4}$ denotes that server ES_1 has failed in the current state and subsequently ES_3 has also failed in the subsequent discrete time step. Note that a direct transition from $\boxed{\enclose{circle}{ES_1}, ES_2, ES_3, ES_4}$ to $\boxed{\enclose{circle}{ES_1}, \enclose{circle}{ES_2}, \enclose{circle}{ES_3}, ES_4}$ is not possible as we consider a discrete time slotted model with only one failure allowed in each time step. From state $\boxed{\enclose{circle}{ES_1}, ES_2, ES_3, ES_4}$, failure scenarios represent probability distributions where $p_1 + p_2 + p_3 + p_4 = 1$. ■

Each transition representing further admissible failures is associated with a synchronization label $[fail_{serverids}]$. Whenever such a transition is executed in the MDP model for stochastic server failures, a corresponding transition with the same synchronization label is executed in the Fault-Recovery procedure MDP which we discuss in detail next.

6.3.3 Model of the Fault-Recovery Procedure

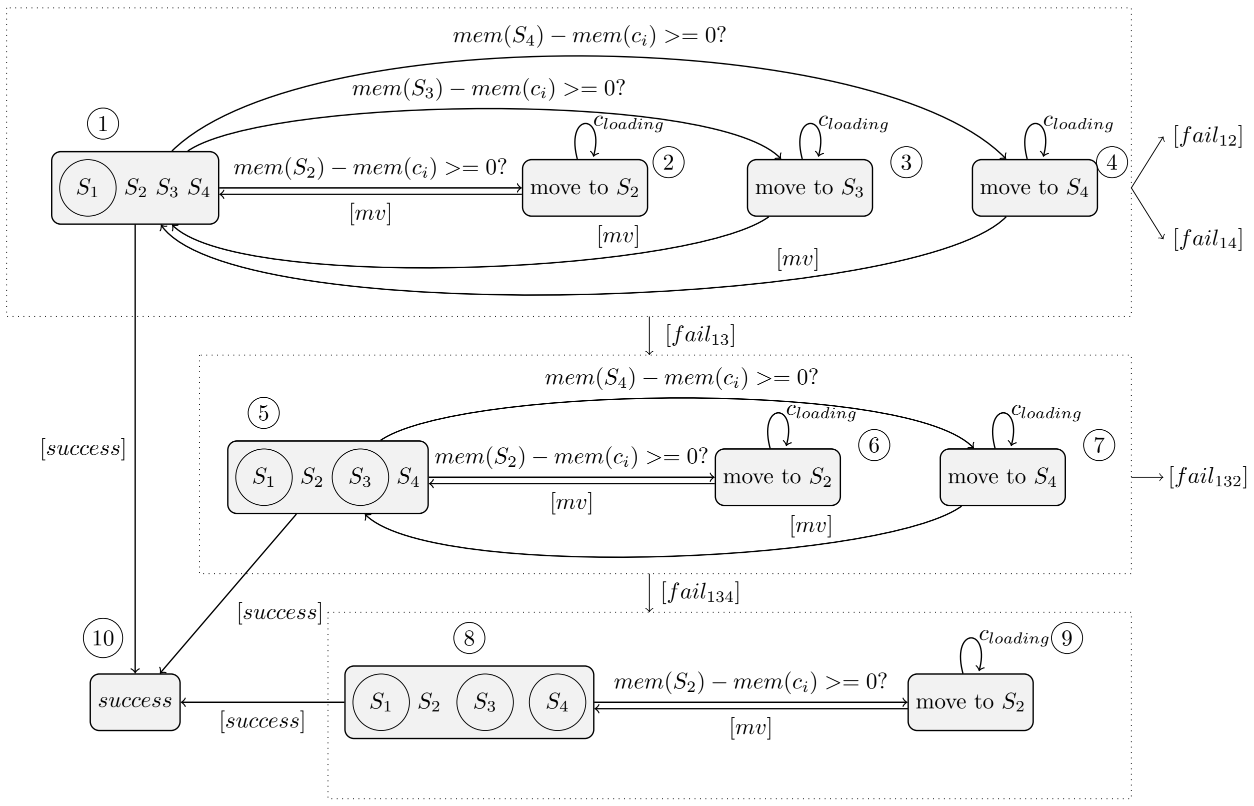


FIGURE 6.8: Model of Fault-Recovery Procedure for each edge site E_j

The Fault-Recovery procedure MDP incorporates non-deterministic choices of servers where containers can be re-initialized. The MDP models all the k-failure scenarios and the respective non-deterministic choices in each failure scenario. The Fault-Recovery procedure can be viewed as comprising several blocks: blocks representing a single faulty server and blocks representing further admissible faulty servers. Additionally, each block comprises the corresponding recovery options. Transitions between such blocks are synchronized with transitions labelled by $[fail_{serverids}]$ of the MDP model for stochastic server failures. Figure 6.8 outlines the MDP model of the Fault-Recovery procedure.

Blocks Representing Singleton Faults: Such blocks correspond to scenarios where a single server has failed and all other servers are available for recovery. Once a failure has been detected, the containers allocated to the faulty server need to be re-initialized on other live servers associated with the edge site E_j. The containers which are to be re-initialized are modeled by the MDP in Figure 6.9 and the remaining memory capacity of the servers where such containers are to be re-initialized are modeled by the MDP in Figure 6.10. Additionally, the initialization time of the containers is modeled by the MDP in Figure 6.11. The transitions within each block of the fault-recovery procedure are synchronized with these MDPs. These models and the transition synchronizations are explained in detail later.

Intuitively, the recovery strategy begins by non-deterministically selecting a live server on which the container is to be re-initialized. Such a non-deterministic choice is contingent on the availability of the required memory for initializing the containers. The recovery procedure then begins initializing the container on the selected server. Upon initialization of the container, the memory capacity of the selected server is decremented by the container's memory requirements. Additionally, the state of the MDP model corresponding to the faulty server is updated to reflect the next container to be re-initialized. The process is repeated until all the containers have been re-initialized in which case the MDP transitions to state $\textcircled{10}$ or there is no capacity available at any of the servers associated with E_j to re-initialize the containers. If the memory capacity of all the available servers within the edge site is exhausted, we employ a global recovery procedure, detailed in Section 6.4.2, where other edge sites are utilized at the cost of additional latencies incurred due to access via the backbone network [3].

Example 6.3.3. State $\textcircled{1}$ of Figure 6.8 depicts the scenario where server ES_1 has failed while servers ES_2, ES_3 and ES_4 are live. In such a scenario, all containers which were assigned to ES_1 have to be re-distributed amongst ES_2, ES_3, and ES_4. Since ES_1 has failed, the MDP model of a faulty server in Figure 6.9 is initialized for ES_1 with containers that were running there. Additionally, three MDP models are initialized with the remaining memory capacity of ES_2, ES_3 and ES_4 corresponding to the MDP model of live servers as in Figure 6.10. States $\textcircled{2}$ - $\textcircled{4}$ denote the recovery procedure for such a scenario. The states $\textcircled{2}$, $\textcircled{3}$ and $\textcircled{4}$ represent non-deterministic choices of servers ES_2, ES_3 and ES_4 respectively. Each non-deterministic choice is contingent upon available memory to initialize the container under consideration. The condition $mem(ES_2) - mem(c_i) \geq 0$ is used to check whether ES_2 has the requisite memory capacity to initialize container c_i. Similar check conditions are utilized for the other servers. The self-loops at these states indicate a container initialization in progress. Upon successful initialization of a container, the MDP returns to state $\textcircled{1}$. The remaining containers are processed identically. When all containers which were allocated to ES_1 have been re-initialized at either ES_2, ES_3 or ES_4, the MDP transitions to state $\textcircled{10}$. ■

Further Admissible Failures: Consider the scenario where a single server $ES_x \in ES_j$ has failed, as a result of which the recovery procedure is initialized. Let us assume that at a certain point of time when the recovery procedure is active, $ES_y \neq ES_x$ fails as well. Such scenarios are modeled by $[fail_{serverids}]$ transitions in the MDP model of server failures. Whenever such a transition is executed in the MDP model of server failures, the corresponding inter-block transition labelled as $[fail_{serverids}]$ is also executed in the MDP model of the fault-recovery procedure. We do not depict all such possible transitions in the MDP in Figure 6.8 for brevity. Additionally, the MDP models of the servers are re-initialized to reflect the new state of the servers and the recovery process begins afresh. The re-initialization is executed upon the synchronized action $[init]$.

Example 6.3.4. The transition from Block 1 to Block 2 is synchronized with the transition $[fail_{13}]$. Thus, whenever the MDP model of server failures executes the transition $[fail_{13}]$, the fault-recovery MDP also executes the inter-block transition $[fail_{13}]$. From Block 1, $[fail_{12}]$ and $[fail_{14}]$ are also present which transition to blocks representing the recovery procedures for scenarios where servers ES_1 and ES_2 have both failed and where servers ES_1 and ES_4 have failed respectively. Such blocks are not depicted in detail in Figure 6.8. ■

Blocks Representing Multiple Faults: Recovery from multiple faults is handled identically, however, with a different set of non-deterministic choices for re-initialization as compared to singleton fault recovery. The set of non-deterministic choices for re-initialization corresponds to the non-faulty servers.

Example 6.3.5. From state $\textcircled{5}$ of Figure 6.8, which corresponds to multiple failures of servers ES_1 and ES_3, the choices of servers to cater to re-initialization are ES_2 and ES_4. Thus, the recovery process has two non-deterministic choices for re-initialization at state $\textcircled{5}$ as compared to state $\textcircled{1}$, where there are three choices. ■

In addition to the transitions and blocks depicted in Figure 6.8, transitions and blocks corresponding to all possible failure scenarios as outlined in the MDP model of server failures are also present. For example, the transition $[fail_{12}]$, corresponding to the scenario where both servers ES_1 and ES_2 have failed and the block representing the corresponding recovery choices in such a scenario are also present. Such blocks and transitions are omitted for brevity in Figure 6.8. We next describe in detail the MDP models of faulty and live servers and the MDP model of containers on which the fault-recovery MDP synchronizes.

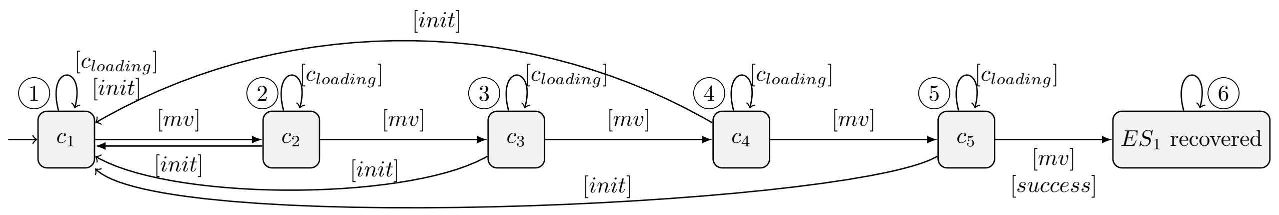


Figure 6.9: Models of Faulty Servers

6.3.4 Model of Faulty and Live Servers

Each server ES_j is associated with two models, "ES_j fail" and "ES_j live" depending on its failure status. Figure 6.9 depicts the MDP model for a faulty server while Figure 6.10 depicts the MDP model of a live server. We describe each below.

Faulty Server: The Faulty Server model is initialized with the κ top-priority real-time application containers. There are two types of states associated with such servers: i) states denoting the κ containers associated with the server prior to failure and ii) a recovery state which denotes all such containers have been re-initialized at alternative servers. The κ containers are en-queued in order of their descending priorities. All transitions for the faulty server model are labelled with four different types of synchronization labels. When the fault-recovery module non-deterministically selects one of the available servers, it transitions to the state depicting the server on which it has invoked the container re-initialization. During the initialization process, the transitions labelled by $[c_{loading}]$ denote the scenario when the container initialization has not yet been completed. A successful initialization is denoted by the transitions labelled by $[mv]$. Note that in the model of the faulty server, each state representing a container has two such transitions, the self-transitions labelled with $[c_{loading}]$ and the transition to the subsequent lower or equal priority container labelled with $[mv]$. Note that similar transitions exist in the fault-recovery controller. The $[init]$ action synchronizes re-initialization in the event of further admissible failures. The $[success]$ action synchronizes with the fault-recovery controller when all κ containers have been re-initialized. The fault-recovery controller, in such a scenario, transitions to the *success* state.

Example 6.3.6. Figure 6.9 depicts the model of the faulty server ES_1. States 1 - 5 denote 5 real-time application containers which were hosted at ES_1 prior to failure. State 6 denotes the scenario that all 5 containers have been successfully re-initialized on different servers. In this example, we assume 5 containers were allocated to ES_1. In other circumstances, the number of states in the chain will indicate the current number of allocated containers. The

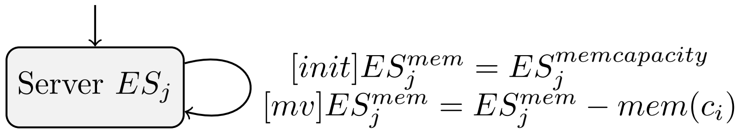

Server ES_j
$[init]ES_j^{mem} = ES_j^{memcapacity}$
$[mv]ES_j^{mem} = ES_j^{mem} - mem(c_i)$

FIGURE 6.10: Model of Live Servers

self-transition in state $\textcircled{1}$ indicates container initialization progress synchronized with the fault-recovery model. The transition from $\textcircled{1}$ to $\textcircled{2}$ is synchronized with the $[mv]$ action which denotes that c_1 which was associated with ES_1 has been re-initialized at a different server. The transition from $\textcircled{5}$ to $\textcircled{6}$ denotes all containers which were allocated to ES_1 have been re-initialized. Whenever a subsequent failure occurs, the transitions labelled with $[init]$ are executed to reset the chain of containers and the recovery process is re-initialized. ■

Servers which are live: For live servers, we only initialize the capacities associated with them, since containers already allocated to these do not form a part of the recovery procedure. Each live server is thus represented by the MDP comprising a single state initialized with the remaining capacity upon failure detection. It comprises two transitions synchronized with the labels $[init]$ and $[mv]$. The $[init]$ action synchronizes with the fault recovery procedure on detection of subsequent failures. Whenever a subsequent failure is detected, the server model is re-initialized with its remaining memory capacity. The $[mv]$ action similarly synchronizes on successful initialization of a container at the server under consideration. The $[mv]$ action decrements the memory requirements of the container from the available memory upon initialization.

Example 6.3.7. Figure 6.10 depicts the MDP structure of servers ES_2, ES_3 and ES_4 when there is no failure associated. Note that each MDP is identical, with the exception of the memory capacity of the server. The MDP comprises a single state initialized with the remaining capacity upon failure detection. In the MDP model of server failures, whenever the transition from $\boxed{\textcircled{ES_1}, ES_2, ES_3, ES_4}$ to $\boxed{\textcircled{ES_1}, ES_2, \textcircled{ES_3}, ES_4}$ is executed, the $[init]$ transition is executed at ES_2 and ES_4 since these are the two remaining live servers. When a container initialization has been completed at a particular live server, the $[mv]$ transition updates the remaining memory capacity as $ES_j^{mem} = ES_j^{mem} - mem(c_i)$. ■

The server MDPs are synchronized with the model of containers to consider their initialization timing characteristics.

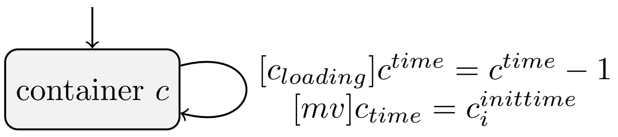

container c
$[c_{loading}]c^{time} = c^{time} - 1$
$[mv]c_{time} = c_i^{inittime}$

FIGURE 6.11: Model of Containers

6.3.5 Model of Container

Each container is represented by a MDP comprising a single state with the required number of discrete time steps for initialization. Figure 6.11 depicts the MDP model of containers corresponding to an application service. It comprises two transitions synchronized with the labels $[c_{loading}]$ and $[mv]$. The $[c_{loading}]$ action synchronizes with the fault recovery procedure as well as the MDP model of the servers to indicate a container initialization in progress. The $[mv]$ action similarly synchronizes upon successful initialization of a container. The $[mv]$ action additionally resets the value of c^{time} with the initialization time of the subsequent container to be initialized from the faulty server.

6.3.6 Model of Timer

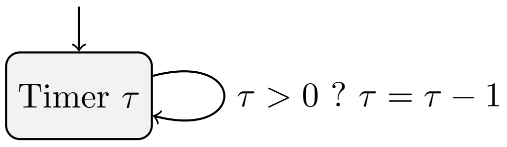


FIGURE 6.12: Model of Timer

The timer MDP also comprises a single state as depicted in Figure 6.12. Upon detection of fault, the timer MDP is initialized with the value τ representing the number of discrete time steps within which the fault-recovery procedure should be completed successfully. In each discrete step, the value of the timer is decremented by 1 until it reaches the value 0. If the MDP model of the fault-recovery procedure reaches the state *success* when the value of τ is greater than or equal to 0, the fault-recovery procedure is deemed to have successfully completed. However, there is a possibility that the *success* state cannot be reached with τ discrete time steps. In such scenarios, we utilize the global recovery procedure outlined in Section 6.4.2.

```
global c1ds2 : bool init false;
global c2ds2 : bool init false;
global c3ds2 : bool init false;
                              ....

global c1ds3 : bool init false;
global c2ds3 : bool init false;
global c3ds3 : bool init false;
                              ....
```

FIGURE 6.13: PRISM Boolean Variables Representing Container Deployments on Servers

6.3.7 Modeling Container Deployments and Users

The MDP model of the Fault-Recovery procedure non-deterministically selects where to deploy the containers from the faulty server. However, it does not contain information pertaining to where the individual containers are deployed. In order to incorporate the information representing the servers on which individual containers are deployed, we use Boolean variables represented by PRISM global variables as depicted in Figure 6.13. For each container, we include a Boolean variable for each server to indicate the deployment of the container on the server. Whenever the fault-recovery MDP completes the initialization of a container at a particular server, the corresponding Boolean variable is set to *true*. Thus, the set of Boolean variables indicate all possible choices of container-server bindings.

Additionally, prior to failure, each container is associated with a set of users availing its services. We model such users using the PRISM reward formulation. Upon server failure, when the model is initialized, the number of users associated with each container at the faulty server is represented by an integer constant as depicted in Figure 6.14. The container-server bindings determine which containers are re-initialized on which MEC servers. The PRISM reward formulation "allocation" is used to model resource contention amongst application containers. The reward formulation utilizes the Boolean container deployment indicator variables in order to assign a low reward value to strategies where competing containers are co-located on the same MEC server while assigning a high reward value to strategies where such resource competing containers are deployed on different servers. The reward formulation is depicted in Figure 6.14. In order to incorporate the impact of users accessing the services of the containerized applications in the synthesized strategies where competing containers are not deployed on the same server, we weigh the reward value by the number of users accessing the containers.

```
//number of users associated with each container
const int c1u;
const int c2u;
const int c3u;

rewards "allocation"
    c1ds2! = true & c2ds2! = true : c1u + c2u + c3u;
    c1ds3! = true & c2ds3! = true : c1u + c2u + c3u;
    c1ds4! = true & c2ds4! = true : c1u + c2u + c3u;
    ....
    c1ds2 == true & c2ds2 == true : 0;
    c1ds3 == true & c2ds3 == true : 0;
    c1ds4 == true & c2ds4 == true : 0;
    ....
endrewards
```

FIGURE 6.14: PRISM Reward Function Definition

Such a reward function, thus, effectively characterizes both the impact of container deployment as well as the impact of the number of users accessing the container.

Example 6.3.8. We utilize the abbreviation $c1ds2$ to denote that Container1 is deployed on Server ES_2. All other Boolean variables are defined analogously. We utilize the abbreviation $c1u$ to denote the number of users availing of the services of Container1. In the Reward Formulation in Figure 6.14, there are three containers that are to be re-initialized. Container1 and Container2 are competing containers and thus deploying them on the same server leads to high resource contention. Thus, in scenarios when containers 1 and 2 are not assigned to the same server, a positive reward value equal to the total number of users availing of the services of containers 1 and 2 prior to failure is assigned. On the other hand, in scenarios where the recovery procedure re-initializes both containers at the same server, the reward value of 0 is assigned to such strategies. All such scenarios are characterized by the Boolean variables indicating where the containers are deployed. ■

The interactions between all the MDP models is characterized by their composition which we describe in detail next.

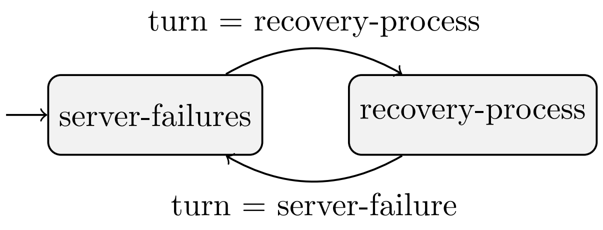


FIGURE 6.15: Turn-Based Game Controller

6.3.8 Composition of MDP models

The aforementioned models systematically abstract the behaviour of the various entities of the MEC system. Note that, all the transitions associated with the aforementioned models and their respective failure probabilities are not shown explicitly for brevity. The overall model is obtained by parallel composition of these MDPs along with the Turn-Based Controller as in Figure 6.15 and is denoted as:

$$\boxed{G_{MEC} = M_{server-failures}||M_{fault-recovery}||M_{servers}||M_{container}||M_{game}||M_{timer}}$$

In our model of the Stochastic Game, there are two players: the MDP model of server failures and the MDP model of the Fault-Recovery Controller. The MDP model of server failures probabilistically induces failures according to the probability distribution of server failures into the MEC system while the MDP model of the Fault-Recovery Controller non-deterministically selects servers where to re-initialize containers to avert such failures. Transitions of the MDP model of server failures are executed only when the Turn-Based Controller is in state $server - failure$, while the transitions associated with the MDP model of the Fault-Recovery controller are executed only when the Turn-Based Controller is in state $recovery - process$. Thus, the MDP Turn-Based Controller indicates which model is allowed to execute transitions while alternating between the two MDPs. As the fault-recovery procedure is triggered after a server has failed, the game is initialized with $turn = recover - process$, i.e, the Fault-Recovery MDP progresses first. Then, the control goes to the MDP model of server failures, which probabilistically selects server failures. The game proceeds, alternating between these two entities. We use this model to derive local fault-recovery strategies with probabilistic guarantees as described next.

6.4 Fault-Recovery Strategy Synthesis

We use the Turn-Based SMG G_{MEC} to synthesize local fault-recovery strategies satisfying the following probabilistic specification:

Given the SMG G_{MEC} find the recovery actions for $M_{fault-recovery}$ which maximizes the probability that $M_{fault-recovery}$ reaches the *success* state within time τ.

The "success" state of $M_{fault-recovery}$ is reached only when all κ containers associated with the faulty server have been re-initialized at alternative servers. Thus, our objective specifies the reachability probability associated with the "success" state of the MDP $M_{fault-recovery}$. We use temporal logic to formally specify the above objective. Specifically, to express properties for SMGs we use the logic rPATL - Probabilistic Alternating-time Temporal logic with Rewards. The above objective can be formally expressed with the rPATL query as follows:

$$\Phi_1 = \langle\langle recovery - process\rangle\rangle P_{\{max=?\}}[F(success)]$$

In this specific scenario, we are interested in the event $F(success)$ where F is the temporal operator eventually [37]. The event denotes that eventually, the fault-recovery strategy reaches the *success* state as depicted in the Fault-Recovery MDP in Figure 6.8. $P_{max=?}$ is the PRISM operator which calculates the maximum probability of the event specified in brackets. Thus, our rPATL objective encodes the maximum probability of reaching the *success* state. The player for which the maximum probability value is sought is specified as $\langle\langle recovery{-}process\rangle\rangle$. Hence, the synthesis problem for an SMG aims to find the optimal strategy π which resolves the non-deterministic choices for the *recovery-process* player. Formally the synthesis problem for G_{MEC} is defined as below.

Definition 6.1 [**SMG Strategy:**]
Given the SMG G_{MEC}, a strategy π is a set of rules to resolve all non-deterministic choices of a player ***recovery-process*** *such that for all opponent strategies σ, where the opponent is* ***server-failures****, the resolution of the non-deterministic choices satisfies the property F. The set of rules is a mapping from each container to the server where it is to be re-initialized, i.e., $\pi : c_i \rightarrow s_x$.*

$$G_{MEC}^{\pi,\sigma} \models \; F, \quad \forall \text{ server-failures } \sigma$$

We use PRISM-Games [38] to implement G_{MEC} along with the specification F. PRISM-Games utilizes Probabilistic Model Checking to determine whether G_{MEC} satisfies F or not. For our rPATL property, it utilizes Probabilistic Model Checking to determine the numerical value of the probability with which F can be satisfied. Thus, it returns a value in the closed interval $[0, 1]$. Model checking systematically explores all states and transitions in the model to check whether it satisfies the given property. PRISM-Games synthesizes a strategy for the player $recovery - process$, in terms of maximizing the probability of reaching the "success" state, for all possible strategies that the player $server - failures$ may choose. Thus, PRISM-Games systematically explores the search space generated by G_{MEC} considering all possible interactions between the two players and their respective choices of actions.

Note that the MDP models of G_{MEC} are initialized to reflect the state of the servers within the edge site where the failure occurred. Thus, G_{MEC} characterizes Local Recovery. Additionally, G_{MEC} only considers the probability associated with the reachability of the success state and hence does not characterize resource contention modeled in terms of rewards. In order to consider the effect of resource contention when re-initializing containers, we use a reward based property using a two-fold local and global recovery described in the following sub-sections.

6.4.1 Local Recovery

The specification F when used in conjunction with the SMG G_{MEC} in PRISM-Games, calculates the maximum probability of satisfying the Temporal Logic Property $F(success)$. When the probability of satisfaction is 1, the fault recovery procedure can generate strategies to ensure successful re-initialization of all the containers which were allocated to the faulty server within τ discrete time-steps. The strategy generated by PRISM-Games with the property Φ_1 only generates strategies that maximize the value of the probability of satisfaction of Φ_1. The property Φ_1, however, does not take into consideration the cumulative reward obtained by such strategies. Since in our formulation, we model the container-server bindings along with the users availing of the services of the containers by utilizing the rewards associated with each state, we use a second property Φ_2 to generate strategies that maximize the cumulative reward obtained from such strategies. The property Φ_2 in rPATL is specified below.

$$\Phi_2 = \langle\langle recovery - process\rangle\rangle R_{\{max=?\}}[F(success)]$$

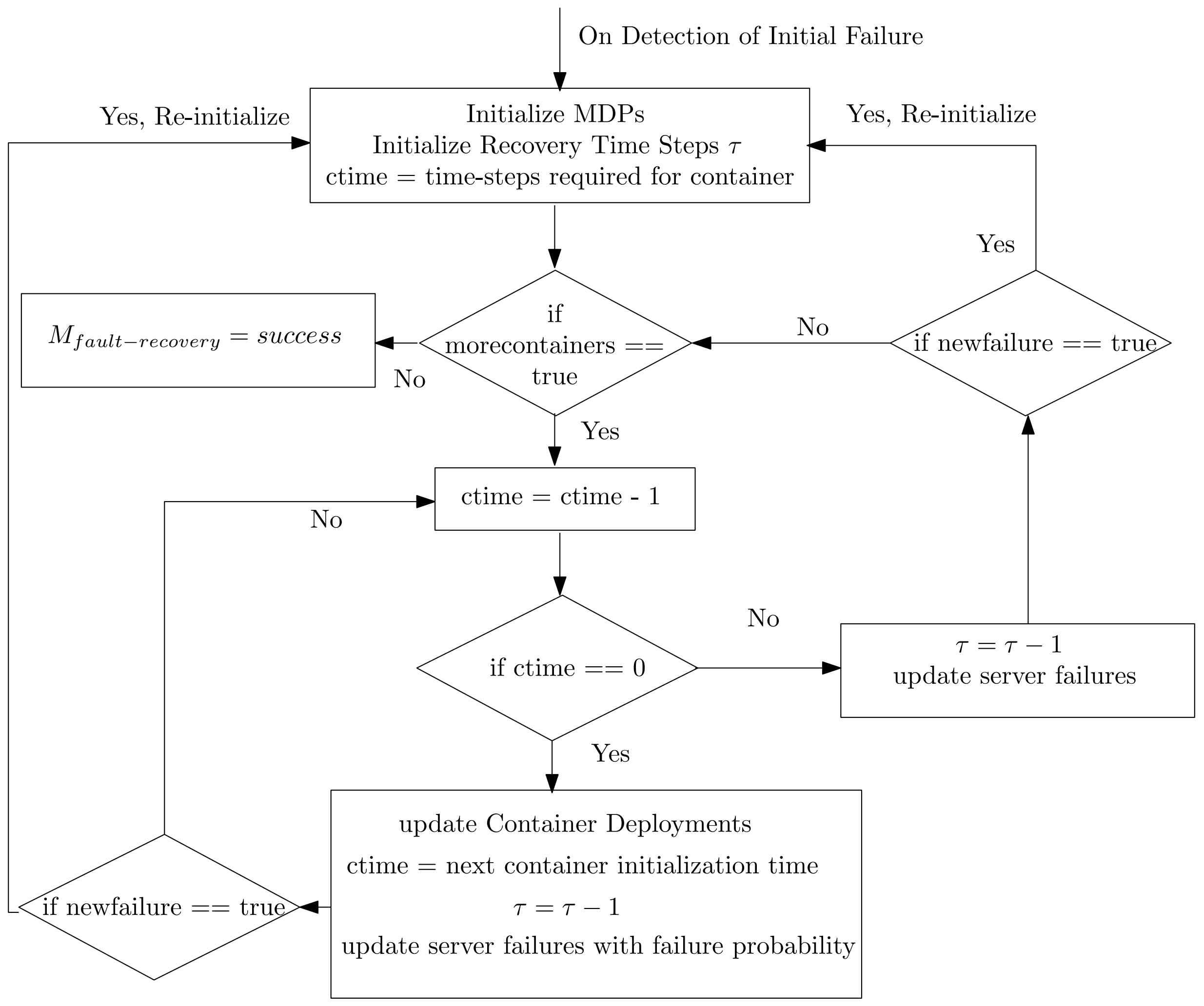

On Detection of Initial Failure
Initialize MDPs
Initialize Recovery Time Steps τ
ctime = time-steps required for container
Yes, Re-initialize
Yes, Re-initialize
Yes
$M_{fault-recovery} = success$
if morecontainers == true
No
No
if newfailure == true
Yes
ctime = ctime - 1
No
No
if ctime == 0
$\tau = \tau - 1$
update server failures
Yes
update Container Deployments
ctime = next container initialization time
$\tau = \tau - 1$
update server failures with failure probability
if newfailure == true

Figure 6.16: Flowchart for Local Recovery Process

The property Φ_2 is identical to property Φ_1 with the exception of the PRISM Reward operator R being utilized in place of the P operator. When the resulting value of property Φ_1 is 1, the strategy generated with property Φ_2 is utilized to re-initialize the containers *within the edge site.* However, when the probability of satisfaction lies in the Real interval [0, 1), denoting inclusive of 0 but exclusive of 1, there is a probability that all containers cannot be re-initialized within the desired recovery time τ, either due to possibility of server failures or inadequate memory availability. In such scenarios, we use the global recovery procedure which we discuss next.

6.4.2 Global Recovery

We design a heuristic approach to drive the global recovery strategy owing to its large scale nature. The global recovery procedure first determines the containers which are to be re-initialized, depending on the outcome of the local recovery procedure. If the local recovery

Algorithm 8: Fault-Recovery Strategy Synthesis

```
1  Initialize $G_{MEC}$ for $E_j$
2  Calculate $\Phi_1 = \langle\langle recovery - process \rangle\rangle P_{\{max=?\}}[F(success)]$
3  if $Pr(\Phi_1) == 1$ then
4      Derive Strategy with $\Phi_2 = \langle\langle recovery - process \rangle\rangle R_{\{max=?\}}[F(success)]$
5      Execute Recovery Strategy derived with $\Phi_2$                 ▷ Local Recovery
6  else
7      $C_{\text{recover}} \leftarrow$ Set of Containers to be recovered from the Faulty Server
8      $S_{\text{recover}} \leftarrow$ Set of Edge Servers in $\chi$-hop neighbourhood of $E_j$      ▷ Global Recovery
9      $c_{id} \leftarrow [1...|C_{\text{recover}}|]$             ▷ $c_{id}$ can take values in range 1 to $|C_{\text{recover}}|$
10     $s_{id} \leftarrow [1...|S_{\text{recover}}|]$             ▷ $s_{id}$ can take values in range 1 to $|S_{\text{recover}}|$
11     sort $S_{\text{recover}}$ in ascending order of Probability of Failures
12     while containers left to be recovered do
13         re-initialize $c_{id}$ at $s_{id}$          ▷ re-initialize with prioritized failure probability
14         $c_{id} \leftarrow (c_{id} + 1)$
15         $s_{id} \leftarrow (s_{id} + 1) \ \% \ |S_{\text{recover}}|$
```

procedure determines it can successfully re-initialize the κ top-priority containers, only the remaining containers are involved in the global recovery process. On the other hand, if the local recovery process ascertains that the probability of a successful recovery within τ for the κ top-priority applications is less than 1, all containers associated with the faulty servers including the κ top-priority containers are involved in the global recovery. Algorithm 8 describes the global recovery process. The global recovery procedure determines all available servers within a χ-hop neighbourhood of the faulty MEC server (Line 8) and sorts them in ascending order of probability of failure (Line 11). A server identifier variable, s_{id}, indicates which server is to be utilized to re-initialize the current container (c_{id}). Upon successful initialization of the current container, s_{id} is incremented by 1 to denote the next server where the next container is to be re-initialized. Thus, re-initialization is carried out in a round-robin manner to evenly distribute the load amongst the various servers (Lines 12 - 16).

Lemma 6.1. *Algorithm 8 requires $O(|S_{recover}|log|S_{recover}|)$ time to sort the χ-hop neighbourhood edge servers in their ascending order of failure probability. Further, it iterates over all containers in $O(|C_{recover}|)$ time to re-allocate them to $S_{recover}$.* ♦

Note that the global recovery strategy spans multiple edge sites. As a consequence, at any discrete time step, there is a possibility of simultaneous failures at different edge sites. In such scenarios, triggering the global recovery algorithm simultaneously can lead to inconsistencies. To ensure synchronization, the global recovery algorithm proceeds if and only if no other global recovery procedure is active within the χ-hop neighbourhood. If another recovery is in progress, the global recovery algorithm waits until the earlier recovery process has been completed. Note that such a mechanism is not required for the local recovery procedure since it acts only within an edge site and we assume edge sites do not have common servers.

6.5 Results and Discussion

We perform extensive simulated experiments on large-scale scenarios to show the efficacy of our approach, and compare its performance against a) failure-aware approach [16] and b) performance-aware approach [118]. All experiments are conducted on a machine with an Intel Xeon E5-1650 Processor with 128GB of RAM. In the following, we describe in detail our experimental setup and the results obtained.

6.5.1 Experimental Setup

We use the 'Existing Commercial Wireless Telecommunication Services Facilities in the San Francisco' dataset [98] as earlier. Within each edge site, we randomly generate the number of edge servers between 1 and 4. We assign access latencies between edge sites via the backbone network randomly in proportion to the distance between the edge sites. The 'Existing Commercial Wireless Telecommunication Services Facilities in San Francisco' dataset does not contain any availability information to denote the associated probability of failures. We use the PlanetLab dataset as described in Section 2.6.5 to simulate server failures. The value of ν is set to 1 second duration, simulated with these probabilities. We perform simulation with San Francisco taxi dataset. We consider 12 representative applications in our setup to characterize the impact of number of applications. Each taxi is associated with a randomly generated application number out of the 12 available applications. Each new coordinate update of taxis from the dataset is treated as the discrete time-step simulated as μ in intervals of 1 minute. We consider a recovery time of $\tau = 60$ discrete time-steps corresponding to 60 ν discrete time-steps. We set the timeout for local recovery procedure to 10 seconds. For each application, we consider container memory usage and initialization times from the DeathStarBench benchmark suite by randomly selecting a container. We consider discretized memory usage in intervals of 50MB. We fetch the corresponding containers from Docker Hub [49] initially. We then note the starting times of each containerized microservice by invoking a fresh Docker container start after stopping all running containers. We use these times as the deployment time of containers. To simulate the effect of resource contention, we select 4 random applications out of the 12. We then assign to each of the 4 applications a different application which when deployed together leads to high resource contention.

Servers	κ **Containers**	**States**	**Time for Model Construction (in seconds)**	**Time for Property F (in seconds)**	**Time for Property H (in seconds)**
3	2	413	0.057	0.025	0.062
3	1	127	0.034	0.016	0.008
3	4	253	0.046	0.019	0.034
3	5	529	0.086	0.026	0.052
4	1	15400	0.565	0.277	0.385
4	3	19816	0.647	0.299	2.051
4	4	19816	0.641	0.363	2.152
4	5	17206	0.396	0.259	0.632
4	6	60867	1.901	1.550	5.026
4	7	NA	Timeout	Timeout	Timeout

Table 6.2: Model Sizes, Construction Time and Verification Times in PRISM-Games

6.5.2 Model Analysis

A PRISM model generated at runtime on the event of a failure is specified in Appendix B. We study the impact of various parameters such as the number of prioritized real-time application containers κ and the number of edge servers in the edge site on the probability of reaching the "success" state. Figure 6.17 depicts the probabilities when κ is varied from 1 and 5 with the number of servers taken as four. As can be inferred from the figure, a lower number of containers corresponds to a higher probability of success. It is interesting to note that for all scenarios considered in Figure 6.17, the probability of reaching the "success" state reaches near 1 when $\tau = 50s$. This suggests the existence of a threshold value of τ beyond which for a given container and server configuration, successful local recovery will always be possible. Additionally, we vary the other parameters, i.e., memory requirements of the containers, their respective initialization times, and available server capacities to characterize their impact on the recovery process. Figure 6.18 depicts four scenarios where κ is fixed at four, whilst server resources, as well as their probabilities of failures are varied. The probability of successful recovery varies in each such scenario. Thus, each parameter in our model uniquely characterizes the probability of the local recovery procedure entering the "success" state.

Table 6.2 lists PRISM-Games model characteristics of some of the randomly generated scenarios from our experimental setup. With an increase in the number of servers, the number of states increases. Even with a large number of states, PRISM-Games is able to both construct the model and synthesize strategies quickly for use in an online on-demand manner. However, while the local recovery strategy can ensure probabilistic guarantees, we found via our experiments that it can only generate such results in an online on-demand manner when $\kappa <= 6$, beyond which PRISM-Games incurs a timeout thereby justifying our two-fold strategy. We thus use a value of $\kappa = 6$ for our experiments. Note that when the number of containers is fixed at 4 but the

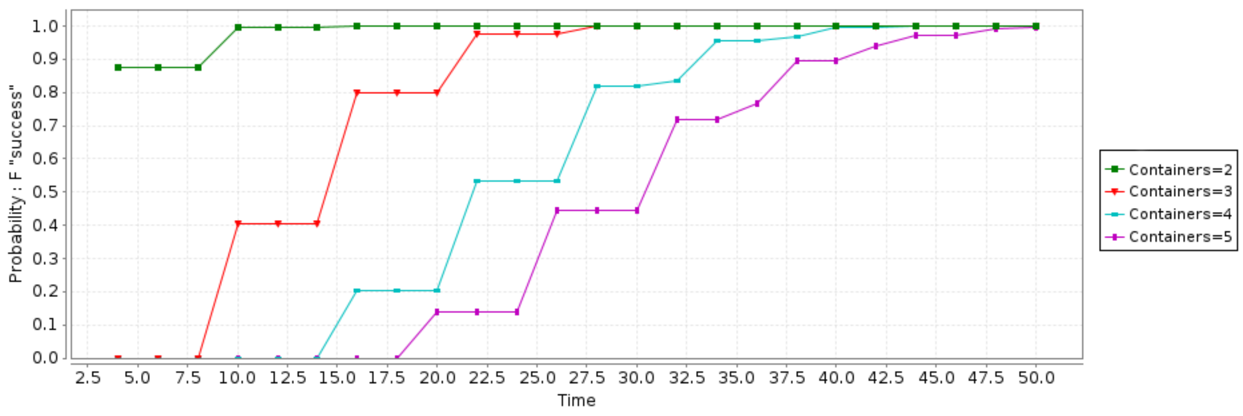


FIGURE 6.17: Probability of Successful Recovery by Varying Number of Containers κ

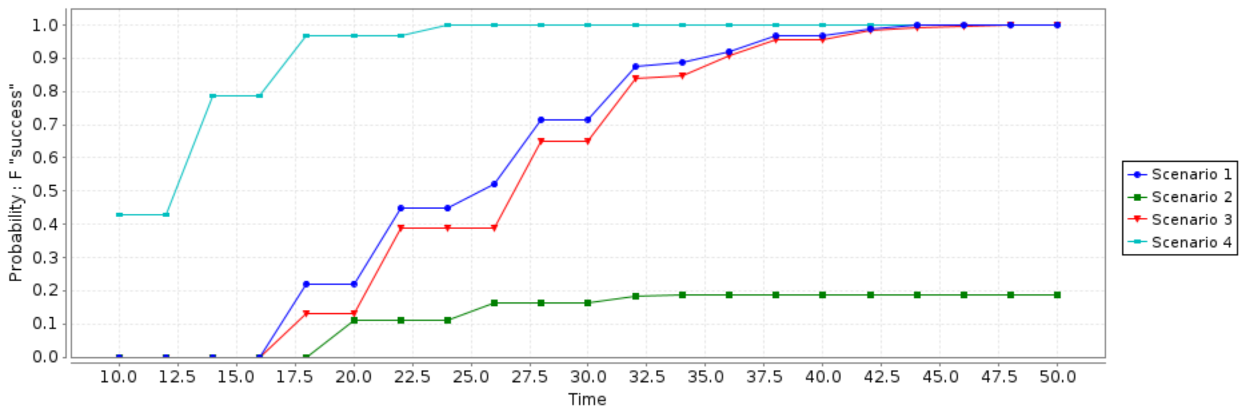
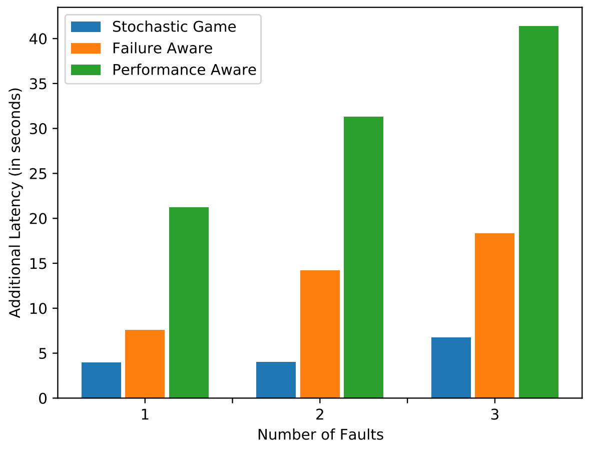


FIGURE 6.18: Probability of Successful Recovery by Varying Container Memory Requirements and initialization Times

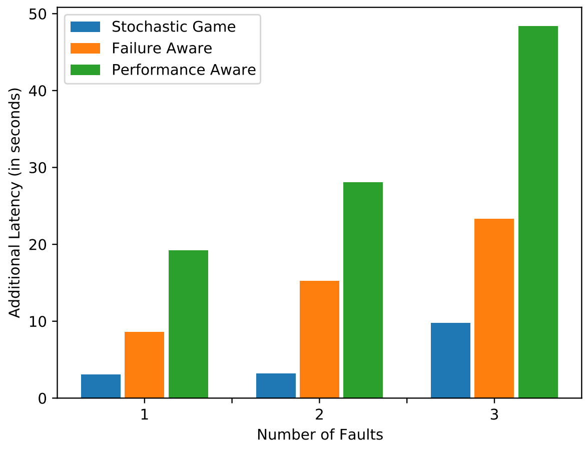
number of containers to be recovered are 3 and 4 respectively, the number of states is identical. Such scenarios occur since the model is also characterized by the number of available servers and their respective capacities. Thus each parameter in our model uniquely characterizes the number of states.

6.5.3 Simulation Results and Discussion

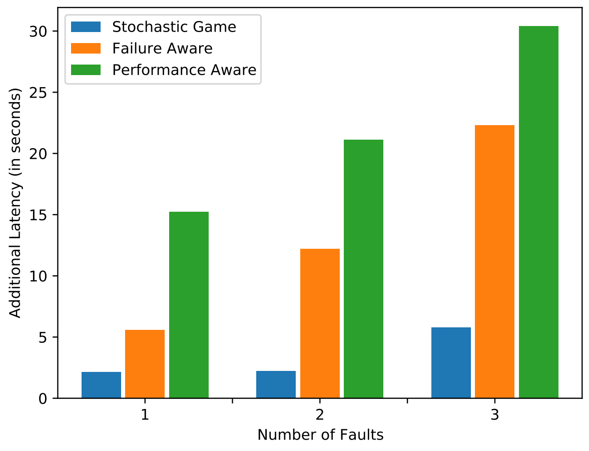
We vary the total number of users as 150, 250, and 350 while keeping the number of servers associated with each edge site and their memory capacities constant. We analyze the results considering the average latencies within a particular edge site since our approach is distributed and replicated within each edge site. In Figure 6.19, we plot the additional latency incurred by users due to the Failure Aware Strategy, the Performance Aware Strategy, and our Stochastic Game based formulation. With an increase in the number of faults, the additional latency incurred by all three strategies increases. Such scenarios occur since an increase in the number of



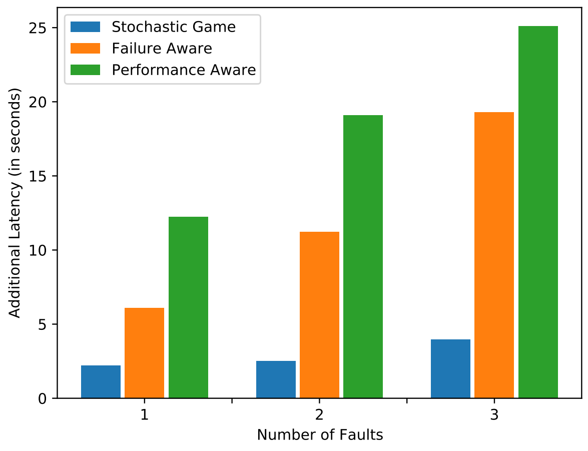
(a) Number of Users = 150



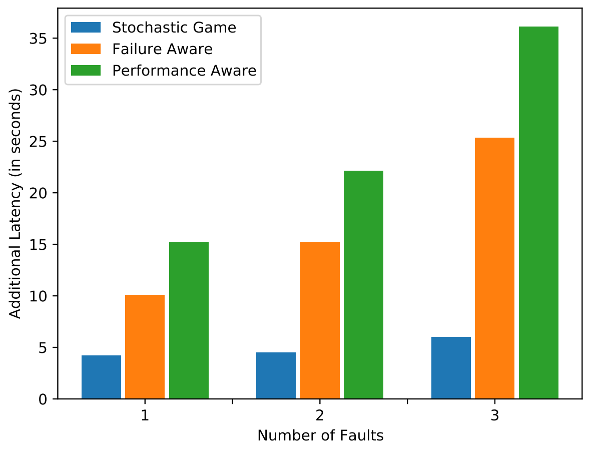
(b) Number of Users = 250



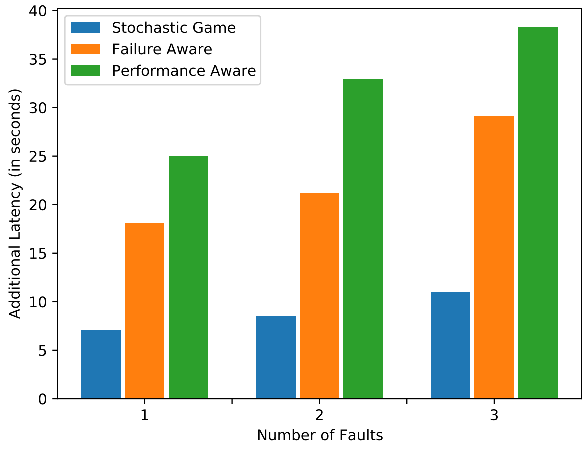
(c) Number of Users = 350



(d) Number of Containers = 3



(e) Number of Containers = 4



(f) Number of Containers = 5

FIGURE 6.19: Additional Latency Incurred with Variable Number of Users and Containers

faults results in higher resource contention among the deployed containers at non-faulty servers. However, with an increase in the number of users, there is no strict increasing/decreasing pattern as observed in Figures 6.19a - 6.19c. Such scenarios conform to our analytical model analysis in Section 6.5.2, wherein we vary the different parameters associated with our model depicted in Figure 6.18. Further, in order to investigate the impact of the number of containers on the recovery process, we vary the number of containers as 3, 4 and 5, while keeping the number of users fixed at 250 and the number of servers associated with each edge site and their memory capacities also constant. Figures 6.19d - 6.19f depict the additional latencies incurred in such scenarios. With an increase in the number of containers, the additional latency increases in all such scenarios in conformance with our analytical model analysis wherein we vary the number of containers as depicted in Figure 6.17. Our Stochastic Game based formulation incurs lower latencies in all such scenarios depicting the effectiveness of our approach. It is interesting to note that the Performance Aware Strategy being agnostic of failures re-initializes containers to enhance load balancing and thereby incurs the highest latencies when subsequent failures do indeed occur. Re-initialization of containers as a result of faults thus has a much more critical impact on additional user perceivable latencies.

6.6 Conclusion

In this chapter, we use formal methods to derive a distributed fault-recovery synthesis strategy for MEC. We model the fault-recovery strategy synthesis problem using a Stochastic Multiplayer Game as a composition of MDPs. Further, we demonstrate how to encode failure recovery objectives in rPATL. We use a combination of local recovery with probabilistic guarantees aided by a heuristic global recovery to drive the failure-recovery process. Experiments on benchmarks demonstrate the effectiveness of our approach. In this chapter and the preceding three chapters, we study the design of MEC policies. In the next chapter, we propose a verification framework to quantitatively characterize the performance of an MEC policy.

Chapter 7

Modeling and Verification of Service Allocation Policies

7.1 Introduction

While the earlier chapters deal with policy design, the main subject of this chapter is policy modeling and verification. In recent years, several service allocation policies taking into consideration different scenarios and optimization metrics have been proposed by several authors in literature. A key issue with service allocation policies is that they do not inherently ensure any quantitative guarantees on the performance metrics, e.g. service request waiting times to be always within desired requirements. Additionally, the consideration of MEC server failures has received much less attention in performance analysis of such allocation policies. Providing performance guarantees is a complex task due to: i) the large configuration space of the user-request-server bindings that a policy can adhere to; ii) the stochasticity of user mobility and

This work is published as:

- Kaustabha Ray and Ansuman Banerjee. "Modeling and Verification of Service Allocation Policies for Multi-Access Edge Computing Using Probabilistic Model Checking", In IEEE Transactions on Network and Service Management 18, no. 3 (2021): 3400-3414.
- Kaustabha Ray and Ansuman Banerjee. "Trace-driven Modeling and Verification of a Mobility-Aware Service Allocation and Migration Policy for Mobile Edge Computing", In Proceedings of IEEE International Conference on Web Services, pp. 310-317, 2020.

service invocation request patterns; iii) the stochasticity of MEC server failures; and iv) the unpredictability of latencies incurred by tasks executed on edge servers. Traditional performance based modeling strategies either develop analytical models to derive performance bounds [123] and mathematically derive analytical bounds on each performance metric characteristic under consideration or resort to simulation resulting in inadequate representation of non-deterministic behaviour. In contrast, the motivation of this work is to design a formal framework to automatically analyze service allocation policies quantitatively without the requirement of analyzing and deriving bounds on each performance characteristic analytically.

To address quantitative verification of allocation policies, we develop a framework to generate probabilistic models of MEC policies. We use a trace-driven approach to generate models where the detailed implementation of the policy is unknown. We learn probabilistic models from MEC system logs where only the sequence of events denoting user service invocations, their allocations to MEC servers, request time-outs, MEC server availability status and so forth are recorded. A trace-driven modeling allows characterization of allocation policies without requiring detailed analytical modeling of the design of each policy. Additionally, to analyze the impact of MEC server failures, we model the interactions between the different MEC components as a Turn-Based Stochastic Multiplayer Game also constructed from MEC system logs. Further, to quantitatively analyze allocation policies against performance requirements, we use Probabilistic Model Checking to derive quantitative guarantees on systems with probabilistic behaviour. We encode performance properties as quantitative statements in derivatives of temporal logic, and a probabilistic model checker is employed to verify the same on the model. Property analysis using Probabilistic Model Checking systematically explores all possible executions of the model to derive quantitative bounds on the constructed model without the requirement of having to manually derive individual analytical bounds on each property [38, 93].

We demonstrate how DTMCs can be utilized to characterize classical allocation policies which do not adaptively change by interacting with the environment. Unlike their classical counterparts, Reinforcement Learning based policies involve interactions with the environment in the form of a reward signal characterizing the effect of service request allocations generated by the policy. As opposed to classical allocation policies, the task of analyzing these RL based policies involves characterizing each decision taken by the policy, quantifying each decision's effectiveness. In this chapter, we demonstrate how a trace-driven approach can be used to characterize such policies. Additionally, to characterize the impact of MEC server failures on service allocation policies, we propose a novel trace driven game model to derive insights into the workings of such policies. The main highlights of this chapter are:

- We formally define and model classical allocation and Reinforcement Learning based allocation policies.
- We model the components of the MEC environment and represent the interaction between these components as a Stochastic Multi-Player Game.
- We describe several performance metrics and specify how scenarios can be encoded into formal properties to quantitatively verify such metrics.
- We present experiments on some popular allocation policies of recent MEC literature on benchmark datasets.

The rest of this chapter is organized as follows. Section 7.2 presents a motivating example. Section 7.3 discusses the formal modeling of policies and methods for verifying properties on such models. Section 7.4 details the obtained results. Section 7.5 concludes the chapter.

7.2 Motivating Example

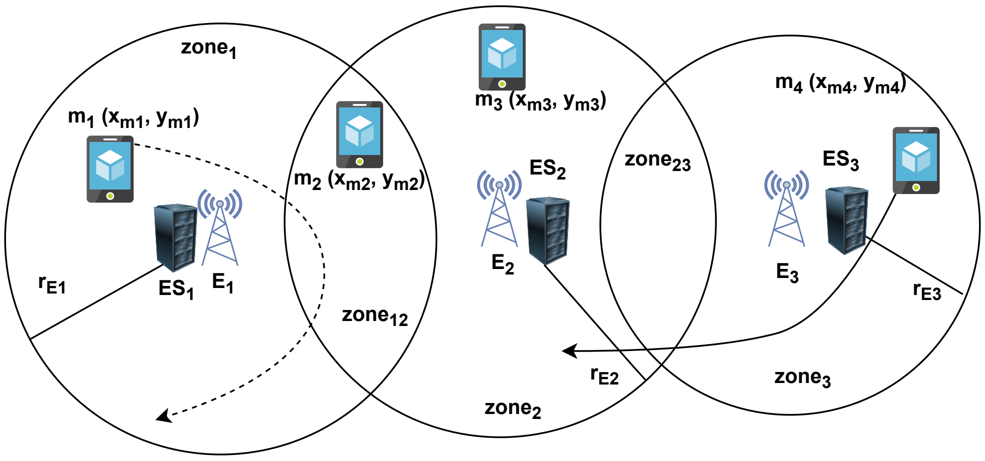

zone$_1$
m$_1$ (x$_{m1}$, y$_{m1}$)
r$_{E1}$
ES$_1$
E$_1$
m$_2$ (x$_{m2}$, y$_{m2}$)
zone$_{12}$
m$_3$ (x$_{m3}$, y$_{m3}$)
ES$_2$
E$_2$
r$_{E2}$
zone$_2$
zone$_{23}$
m$_4$ (x$_{m4}$, y$_{m4}$)
ES$_3$
E$_3$
r$_{E3}$
zone$_3$

FIGURE 7.1: Representative MEC Server Allocation Scenario

We illustrate the problem context on a representative mobility-aware allocation policy with a simple example for ease of explanation. Consider an MEC system comprising three edge sites E_1, E_2, and E_3 associated with edge servers ES_1, ES_2, and ES_3 respectively as shown in Figure 7.1. Unlike in the previous chapters, for sake of simplicity of illustration, we consider that each edge site is associated with a single edge server as considered in recent allocation policies [6, 7, 8, 47, 124]. Each edge site is represented by its latitude and longitude coordinates. For example, the coordinate of edge site E_1 is (x_{E_1}, y_{E_1}). Further, the edge sites E_1, E_2, and E_3 are associated with service radii r_{E_1}, r_{E_2} and r_{E_3} respectively representing their coverage area as considered earlier. A gaming service provider deploys its services identically across all such MEC servers. Each MEC server has a capacity associated with it, denoting the number of user service requests it can provide. An implicit assumption to such a capacity model is that the gaming service provider can estimate the resource requirements for an invocation of its gaming service. Hence each service request can be assumed to require identical resource requirements with minor runtime deviations [6]. Each edge site caters to service requests from devices within a specific radius termed as the *service zone* of the server. Service zones can be overlapping, i.e. within such zones, a service request from a particular user can be allocated to one of the servers covering that particular service zone. In Figure 7.1, $zone_1$ denotes the area under the coverage area of server ES_1 alone. Similarly, $zone_{12}$ denotes the area under the overlapping coverage area of servers ES_1 and ES_2. The service requests from devices present in the overlapping service zones of ES_1 and ES_2, i.e., $zone_{12}$ can be allocated to either ES_1 or ES_2. Consider the scenario shown in Figure 7.1. Device m_1 can only be allocated to server ES_1, m_2 can be allocated to either server ES_1 or ES_2 while m_3 and m_4 can be allocated to servers ES_2 and ES_3 respectively.

Let us assume each server has a capacity of simultaneously serving 3 gaming service invocations. Additionally, service requests from devices in overlapping zones allocated to one of the applicable servers can be migrated to other applicable servers in the same overlapping zones. The example in Figure 7.1 comprises 4 mobile devices, m_1, m_2, m_3 and m_4. Assume m_1 follows a trajectory as indicated by the dashed curved arrow in Figure 7.1 while m_2 and m_3 remain static. The representative policy, being mobility-aware, assigns the requests from m_1 to server ES_1 and requests from m_2 and m_3 to ES_2. Service requests from m_1 can only be allocated to ES_1 when m_1 is in $zone_1$. However, once m_1 moves into the $zone_{12}$, the request can be allocated to either server. The request from m_1 which was allocated to ES_1 is migrated to ES_2 considering the trajectory of m_1's movement towards the zone serviced exclusively by ES_2. In such a scenario, the requests from m_1, m_2 and m_3 are allocated to server ES_2. In such a system state, consider a device m_4, following a trajectory indicated by the solid curved arrow in Figure 7.1, invokes the gaming service after reaching the zone $zone_2$ exclusively served by ES_2. The invocation request from m_4 can no longer be accepted immediately since server ES_2 has already reached its capacity. However, had m_1 been continued to be served from ES_1, the request from m_4 could have been onboarded. It may be noted that for a different arrival pattern and a different representative policy, this may not have served the intended purpose as well.

Given an allocation policy, determining its suitability for an unknown mobility pattern and requesting distribution and verifying whether it meets the desired performance metrics is the problem we attempt to address in this work. While for the simple scenario at hand in Figure 7.1, the request-server binding configurations are limited, in a realistic scenario, there exists a myriad of service request invocation patterns and bindings. Additionally, server failures can have a critical impact on such bindings. Thus, to systematically model such policies, we formulate a trace-driven framework to model and analyze its conformance to performance metrics. In the following, we discuss in detail the formal modeling of policies and quantitative verification of performance metrics.

7.3 Modeling and Verification of Allocation Policies

In this section, we present our formal model of allocation policies and representative metrics of performance characteristics. Before we present the formalism, we discuss the context of an allocation policy. As earlier, in a particular area, there are m MEC servers, $ES = \{ES_1, ES_2, \ldots ES_m\}$. Each MEC server s_j is represented by its latitude and longitude coordinates and has a service radius r_j associated with it. C_j denotes the maximum number of service invocations the j-th MEC server can simultaneously cater to. At any point of time t, $U(t) = \{u_1, u_2, u_3, \ldots\}$ specifies the list of users who invoke the service in the area. The location of each user $u_i \in U(t)$ is specified by its latitude and longitude coordinates at timepoint t. A user $u_i \in U(t)$ can only avail of the services of a server $ES_j \in ES$ if it is located within the area spanned by the service radius of the server. The set of servers and the set of users form the environment. We now formally define an allocation policy:

Definition 7.1 [**Allocation Policy:**]
An allocation policy at any point of time, t, is a mapping $\pi(t) : U(t) \rightarrow ES$.

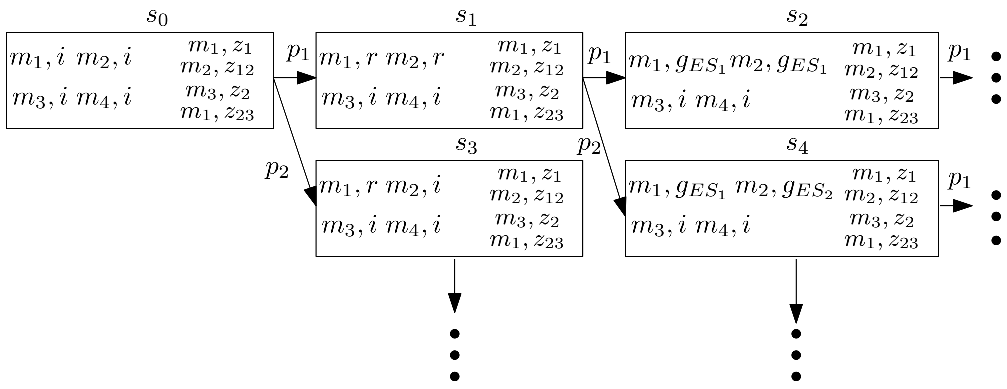
The policy can be considered as an agent that starts from an initial configuration, and at each point of time t, accumulates requests from individual devices. Based on the proximity of servers and the direction of movement, the agent produces the allocation $\pi(t)$. In the following subsections, we discuss two modeling strategies depending on the nature of a policy's interaction with the environment. Subsequently, we formally model the impact of MEC server failures on such policies.

7.3.1 Classical Allocation Policies

We refer to classical allocation policies as those in which the agent's interaction with the environment is limited to receiving user requests and executing user-server allocation bindings. In this section, we model such classical policies as a DTMC, described in the following discussion.

Representation of Policy DTMC: Consider an MEC system comprising a set of devices and a set of MEC servers. Since the locations of the servers remain fixed, these can be specified in terms of a set of fixed discrete coordinates. We discretize the set of possible locations of a device into several zones as follows: we create a new zone for each area where requests are provisioned by a single MEC server, additionally, we create a new zone for each area where requests can be provisioned by multiple MEC servers.

Example 7.3.1. We create 5 zones for the example in Figure 7.1. Henceforth, we use the abbreviation z_1 to denote $zone_1$ and so forth. z_1 refers to the area under the service control of server ES_1 alone, z_{12} refers to the area under the service control of both servers ES_1 and ES_2 and z_2 refers to the area under the service control of ES_2 alone. Similarly, z_3 and z_{23} are under control of servers ES_3 and ES_2, ES_3 respectively. ■

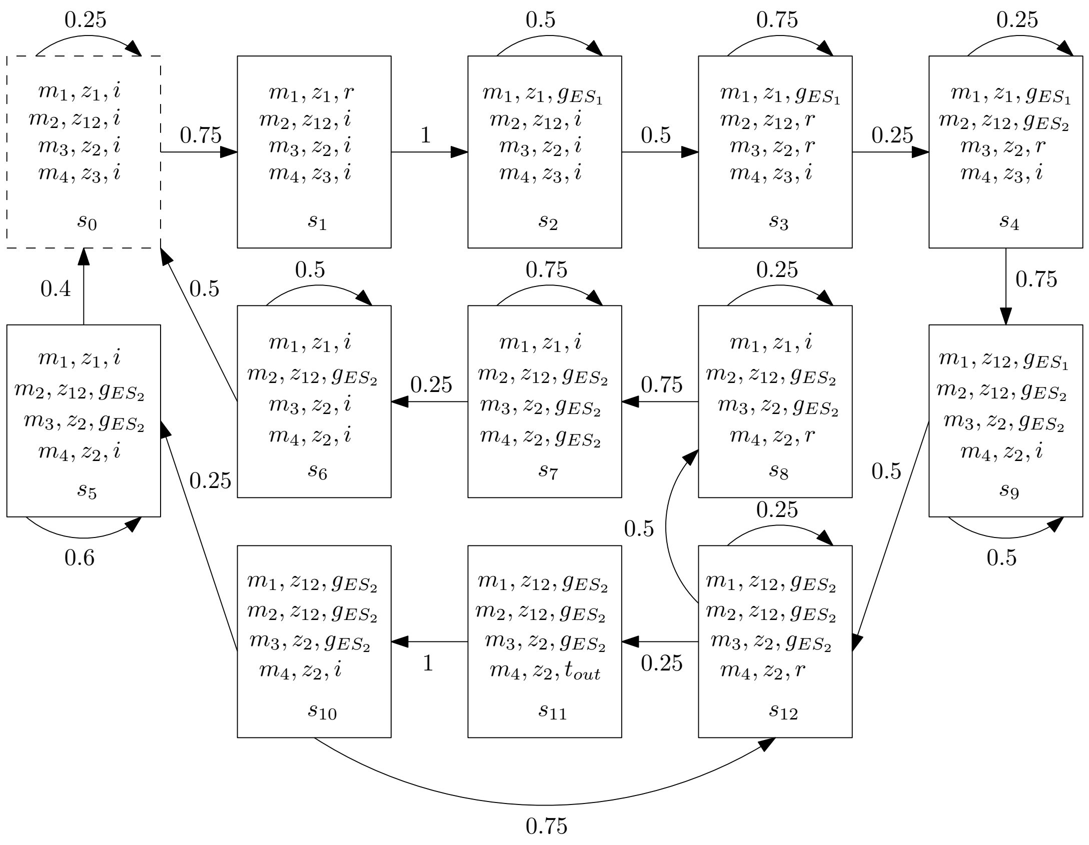


s_0
m_1, i m_2, i
m_3, i m_4, i
m_1, z_1 m_2, z_{12} m_3, z_2 m_1, z_{23}
p_1
p_2
s_1
m_1, r m_2, r
m_3, i m_4, i
m_1, z_1 m_2, z_{12} m_3, z_2 m_1, z_{23}
p_1
p_2
s_2
m_1, g_{ES_1} m_2, g_{ES_1}
m_3, i m_4, i
m_1, z_1 m_2, z_{12} m_3, z_2 m_1, z_{23}
p_1
s_3
m_1, r m_2, i
m_3, i m_4, i
m_1, z_1 m_2, z_{12} m_3, z_2 m_1, z_{23}
s_4
m_1, g_{ES_1} m_2, g_{ES_2}
m_3, i m_4, i
m_1, z_1 m_2, z_{12} m_3, z_2 m_1, z_{23}
p_1

FIGURE 7.2: Representation of Policy DTMC

State Representation of DTMC: Each *state* of the DTMC represents the states of the devices and the representative zones of the devices where they are located as represented by the policy. The states are represented as bit vectors where the bits correspond to Boolean Atomic Propositions representing the states of the devices as users move and invoke service requests, and the policy agent as it executes user-server bindings.

Transition Representation of DTMC: Transitions occur when the state of a device changes and are represented with probabilities from one state to another. Such a state change for a device can occur when a device invokes a service request or when a request is bound to a server or when a request times out. From a particular state, the policy can reach multiple successor states. Figure 7.2 depicts scenarios involving multiple successor states. From state s_0, representing all current idle devices, successor states would denote all possible service request enumerations. Idle devices and service requests are denoted by i and r respectively. Similarly, from state s_1, where both devices m_1 and m_2 invoke service requests, successor states denote all possible user-server bindings. For a request-server binding, there are multiple possibilities in zones which are under the control area of multiple servers. For example, in s_2, m_1 and m_2's requests have been allocated to server ES_1 represented by g_{ES_1} while in s_4, m_1 and m_2's bindings are (ES_1, ES_2) respectively denoted by g_{ES_1} and g_{ES_2} respectively. Transitions represent probability distributions, hence, the sum of outgoing probabilities from one state to all successor states is always 1. Thus, for each state, $\sum_{i=1}^{n} p_i = 1$. The labelling function is a mapping from the bits to the state of the device which each bit represents. The transition probabilities are thus utilized to model both mobility of users across service zones as well as service request invocations and their resulting user-server bindings. Note that the transition probabilities in different service zones can follow different distributions.



m_1, z_1, i m_2, z_{12}, i m_3, z_2, i m_4, z_3, i s_0
m_1, z_1, r m_2, z_{12}, i m_3, z_2, i m_4, z_3, i s_1
m_1, z_1, g_{ES_1} m_2, z_{12}, i m_3, z_2, i m_4, z_3, i s_2
m_1, z_1, g_{ES_1} m_2, z_{12}, r m_3, z_2, r m_4, z_3, i s_3
m_1, z_1, g_{ES_1} m_2, z_{12}, g_{ES_2} m_3, z_2, r m_4, z_3, i s_4
m_1, z_1, i m_2, z_{12}, g_{ES_2} m_3, z_2, g_{ES_2} m_4, z_2, i s_5
m_1, z_1, i m_2, z_{12}, g_{ES_2} m_3, z_2, i m_4, z_2, i s_6
m_1, z_1, i m_2, z_{12}, g_{ES_2} m_3, z_2, g_{ES_2} m_4, z_2, g_{ES_2} s_7
m_1, z_1, i m_2, z_{12}, g_{ES_2} m_3, z_2, g_{ES_2} m_4, z_2, r s_8
m_1, z_{12}, g_{ES_1} m_2, z_{12}, g_{ES_2} m_3, z_2, g_{ES_2} m_4, z_2, i s_9
m_1, z_{12}, g_{ES_2} m_2, z_{12}, g_{ES_2} m_3, z_2, g_{ES_2} m_4, z_2, i s_{10}
m_1, z_{12}, g_{ES_2} m_2, z_{12}, g_{ES_2} m_3, z_2, g_{ES_2} m_4, z_2, t_{out} s_{11}
m_1, z_{12}, g_{ES_2} m_2, z_{12}, g_{ES_2} m_3, z_2, g_{ES_2} m_4, z_2, r s_{12}
0.25
0.75
1
0.5
0.5
0.75
0.25
0.25
0.4
0.5
0.5
0.75
0.25
0.75
0.25
0.75
0.6
0.25
0.5
0.25
0.5
1
0.25
0.5
0.75

FIGURE 7.3: DTMC Model of a Representative Policy

Example 7.3.2. We use a trace driven approach to learn the DTMC from logs captured from policy simulations. Figure 7.3 shows a representative example construction on the example in Figure 7.1. Evidently, the total configuration space comprises more states than what is captured by the trace driven DTMC. The rationale behind a reduced design is that only the situations which occur in the logs are captured in the DTMC while scenarios which do not occur in the simulation of the policy are absent. State s_0 represents the initial configuration of the system. Initially, m_1, m_2, m_3 and m_4 are in zones z_1, z_{12}, z_2 and z_3 respectively. Additionally, all the devices are idle with none of them having invoked any service request. This is represented by i in Figure 7.3. Thus, in each state, all true APs are shown in the DTMC while all false APs have been excluded for brevity which are nonetheless present. For example, in state s_0, all

Algorithm 9: Generate DTMC Representation

```
Input   : Log File of Device Induced Configurations
Output: Device Induced Policy DTMC Model
1  $C \leftarrow$ Time-annotated Configurations in Log File
2  $UC \leftarrow$ Unique Configurations $\in$ C        ▷ UC is a vector where NULL represents the
     configurations are not Unique
3  $S \leftarrow [1 ...$ Number of Unique Configurations $C]$
4  $D \leftarrow$ NULL                                  ▷ List of Vectors of Each State
5  $d \leftarrow |\bigcup_{i=1}^{|C|} UC[i]|$          ▷ Number of bits for each state
6  $map \leftarrow NULL$                                ▷ Map of All Bits to Device State
7  $k \leftarrow 0$
8                                                       ▷ Compute State Vectors
9  foreach $u \in UC$ do
10     foreach $b \in u$ do
11         if $b \notin map$ then
12             $map[b] \leftarrow k$
13             $k = k + 1$
14 foreach $s \in S$ do
15     $v \leftarrow$ initialize a vector of $d$ bits set to 0
16     foreach $b \in UC[s]$ do
17         Set $map[b]$ bit to 1 in $v$
18     $D$.append($v$)
19                                                      ▷ Compute Transition Probabilities
20 foreach $c \in C$ do
21     $ind \leftarrow$ indices of all occurrences of $c$ in $C$
22     $indsuccessor \leftarrow$ indices of successor states of $ind$
23     $nsuccessor \leftarrow |indsuccessor|$
24     foreach $s \in indsuccessor$ do
25         if $transition\ from\ c\ to\ s\ is\ not\ defined$ then
26             $nevent \leftarrow$ |occurrences of $c$ to $s$|
27             $p \leftarrow nevent/nsuccessor$
28             add transition from $c$ to $s$ with probability $p$
```

false APs corresponding to states s_1 to s_{12} are also present. In s_0, m_1 invokes a service while the other devices remain idle, as depicted by the transition from s_0 to s_1. The only change in the APs for s_1 is from i to r for m_1 representing an invocation. Such a request is bound to server ES_1 represented by state s_2. Note that there are no other choices since m_1 is located in the zone exclusively served by ES_1. Additionally, consider a scenario in the log files where m_2 and m_3 both make service requests simultaneously with the request from m_1 still active, denoted by state s_3. Request-Server bindings for such invocations are depicted subsequently by s_4 and s_9. In s_9, the policy, taking the trajectory of m_1 into account, migrates m_1's service binding from server ES_1 to server ES_2 depicted by state s_{12}. In such a state, an invocation from m_4 in z_2 keeps waiting since the capacity of ES_2 has already been reached, denoted by the self transition to state s_{12}. The request eventually times out, denoted by s_{11}. The device re-initiates the service request denoted by the immediate transitions to s_{10} and eventually to s_{12} in the next two discrete time steps. It may be noted that self transitions are absent in such cases. The transition probabilities are thus probability distributions over each state's successor states as observed from the traces. All other scenarios occur with probability 0 in the DTMC model since they do not occur in the simulation logs and are not shown explicitly. For example, in s_3, the simultaneous requests from m_2 and m_3 were not granted simultaneously by any server and is absent in the log files. Hence a transition from s_3 to s_9 which would have depicted such a state transition has probability 0. ■

Algorithm 9 outlines the DTMC construction process. Lines 9-18 of the algorithm calculate the bit vector representation of each state. The algorithm computes the number of bits necessary by finding the unique configurations from the trace and uses a Hash Table to produce the vector by setting appropriate bits to 1. Lines 20-28 compute the transition probabilities for each state by calculating frequency distributions of the respective successor states. The DTMC constructed is device induced since the states of the DTMC evolve according to the behaviour of the policy as captured in the simulation logs.

7.3.2 Reinforcement Learning Based Allocation Policies

Building on our formal model of classical service allocation policies, we now describe in the following a trace-driven modeling of RL based allocation policies. In RL based allocation policies, the agent, after observing the state of the environment, produces $\pi(t)$, corresponding to which it receives a reward. Reward signals quantify the agent's choice of decisions to produce the request-server bindings. We model RL based policies as Markov Decision Processes (MDPs) which allow effective representation of the non-deterministic choices of actions that the policy can undertake in each state, namely, allocating user requests and migration of already allocated requests. Formally, RL based policies are thus modeled as MDPs described as below.

State Representation of MDP: We use the notion of zones as discussed in Section 7.3.1 to represent the locations of users and servers. States of the MDP are represented identically as bit vectors similar to the state representation of the DTMC as discussed in Section 7.3.1. For each state $s \in S$, N_A denotes the number of allocated requests, N_T denotes the number of timed-out requests and N_R denotes the number of total service requests. For example, in state s_2, $N_A = 2$, $N_T = 0$ and $N_R = 2$. We utilize N_A, N_T and N_R in our reward formulation as explained later.

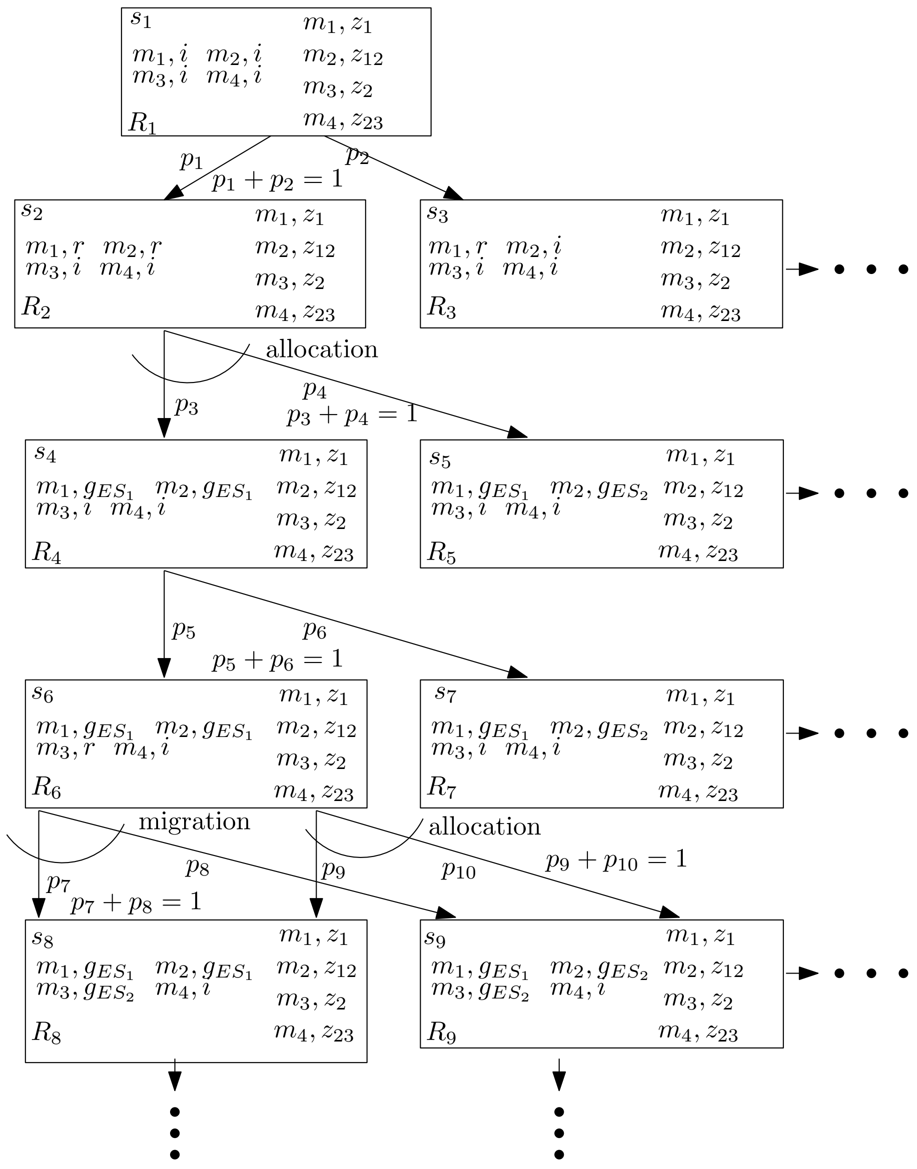


FIGURE 7.4: Representation of Policy MDP

Transition Representation of MDP: Transitions occur when either the state of a device changes or the policy agent decides to allocate a pending request to a server or migrates an already assigned request to a different server. Figure 7.4 depicts the state space involved in such a representation. For a particular state, successor states include all possible scenarios constituting changes in the states of devices as well as all possible scenarios constituting the policy agent's actions pertaining to allocation and migration. From state s_0, successor states determine all possible service request enumerations similar to the DTMC representation in Section 7.3.1. However, actions corresponding to the policy agent, i.e., allocation and migration are represented non-deterministically as shown in state s_6. Successor states from s_6 denote all possible allocation and migration actions of the policy in such a scenario. The successor states corresponding to each action in a state form a probability distribution. Thus, for a particular state and for a particular action, the sum of outgoing probabilities is always 1. All possible outgoing transitions for a particular state corresponding to allocation and migration actions are represented by transitions enclosed by quarter circles in Figure 7.4. For example, from s_6, the quarter circle corresponding to migration includes transitions to s_8 and s_9 with probabilities p_7 and p_8 respectively with sum as 1. Similarly, p_9 and p_{10} correspond to the allocation action represented by the quarter circle. This is unlike the DTMC representation where all outgoing transitions from each state jointly represent a probability distribution. However, all outgoing transitions from a state which represent state changes of devices form a single probability distribution. Thus, such transitions and their corresponding probability distributions are identical to the DTMC representation. For example, the outgoing transitions from s_1 represent device state changes and thus a single probability distribution is represented with outgoing transitions to s_2 and s_3 with $p_1 + p_2 = 1$.

Reward Representation of MDP: Each state s_i is additionally augmented with a reward R_i. The policy agent is a sequential decision making entity, the results of which lead the environment to states as represented in the MDP. Reward R_i thus serves as an indicator to the decisions taken by the policy agent leading to the corresponding environment representation of s_i. The labelling function is retained as in the DTMC representation.

Calculation of Rewards: All states are assigned a reward in the closed interval $[-1, 1]$. Additionally, only those states which are successor states representing the decisions taken by the policy agent are assigned non-zero rewards. Reward for each state is computed as follows:

$$R_s = \frac{N_A - N_T}{N_R}; \forall s \in S, \text{where } N_R > 0$$

Example 7.3.3. Figure 7.5 outlines a representative trace-driven MDP construction of an RL based policy using the motivating example outlined in Figure 7.1. State s_0 represents the initial configuration of the system in which all the devices are idle. m_1's service invocation is represented by the change in the bit vector from i to r from s_0 to s_1. This is identical to the DTMC representation since only m_1's state change is involved. Since the request from m_1 can only be allocated to A, only one successor state from s_1 corresponding to the *allocation* action is present with probability 1. However, when m_2 and m_3 simultaneously make service requests, there are multiple servers which can provision m_2's service request. In the example, we assume there were scenarios where the policy agent allocated m_2's request to server ES_1 in some scenarios and server ES_2 in certain other scenarios. Note that such choices depend on the policy under consideration. Our model incorporates a general framework to allow all such

Algorithm 10: Generate MDP Representation

```
Input  : Log File of Device Induced Configurations
Output: Device Induced Policy MDP Model
1  $C \leftarrow$ Time-annotated Configurations in Log File
2  $UC \leftarrow$ Unique Configurations $\in$ C        ▷ UC is a vector where NULL represents the
     configurations are not Unique
3  $S \leftarrow [1 \ldots$ Number of Unique Configurations $C]$
4  $D \leftarrow$ NULL                                  ▷ List of Vectors of Each State
5  $d \leftarrow |\bigcup_{i=1}^{|C|} UC[i]|$              ▷ Number of Bits for Each State
6  $map \leftarrow NULL$                               ▷ Map of All Bits to Device State
7  $k \leftarrow 0$
8                                                     ▷ Compute State Vectors
9  foreach $u \in UC$ do
10     foreach $b \in u$ do
11         if $b \notin map$ then
12             $map[b] \leftarrow k$
13             $k = k + 1$
14 foreach $s \in S$ do
15     $v \leftarrow$ initialize a vector of $d$ bits set to 0
16     foreach $b \in UC[s]$ do
17         Set $map[b]$ bit to 1 in $v$
18     $D$.append($v$)
19                                     ▷ Compute Action Transition Probabilities
20 foreach $c \in C$ do
21     $ind \leftarrow$ indices of all occurrences of $c$ in $C$
22     $S_{alloc} \leftarrow$ successor states with *allocation* action
23     $S_{mig} \leftarrow$ successor states with *migration* action
24     $S_{device} \leftarrow$ successor states with *device* action
25         ▷ calculate probability distribution for *alloc*, *mig* and *device* actions according to
       transition probability computation from Algorithm 9
26     foreach $action \in alloc, mig, device$ do
27         $f_{choice} \leftarrow$ frequencies of choices in $S_{action}$
28         add transitions with the probability distributions $f_{choice}/|S_{action}|$ from $c$ to
             $s \in S_{action}$
```

possible representations. The *allocation* action from s_3 is thus a probability distribution over the choices of request-server bindings which are part of the logs. The probability distribution is represented by the quarter circle corresponding to the allocation action representation in Figure 7.4. In this specific instance, the transitions to s_4 and s_{10} represent this probability distribution. In the event that m_1's request which had already been bound to server ES_1, is migrated to server ES_2 owing to m_1's trajectory, the outgoing transitions from s_{10} which indicate *migration* action, form a probability distribution over the decision over whether to

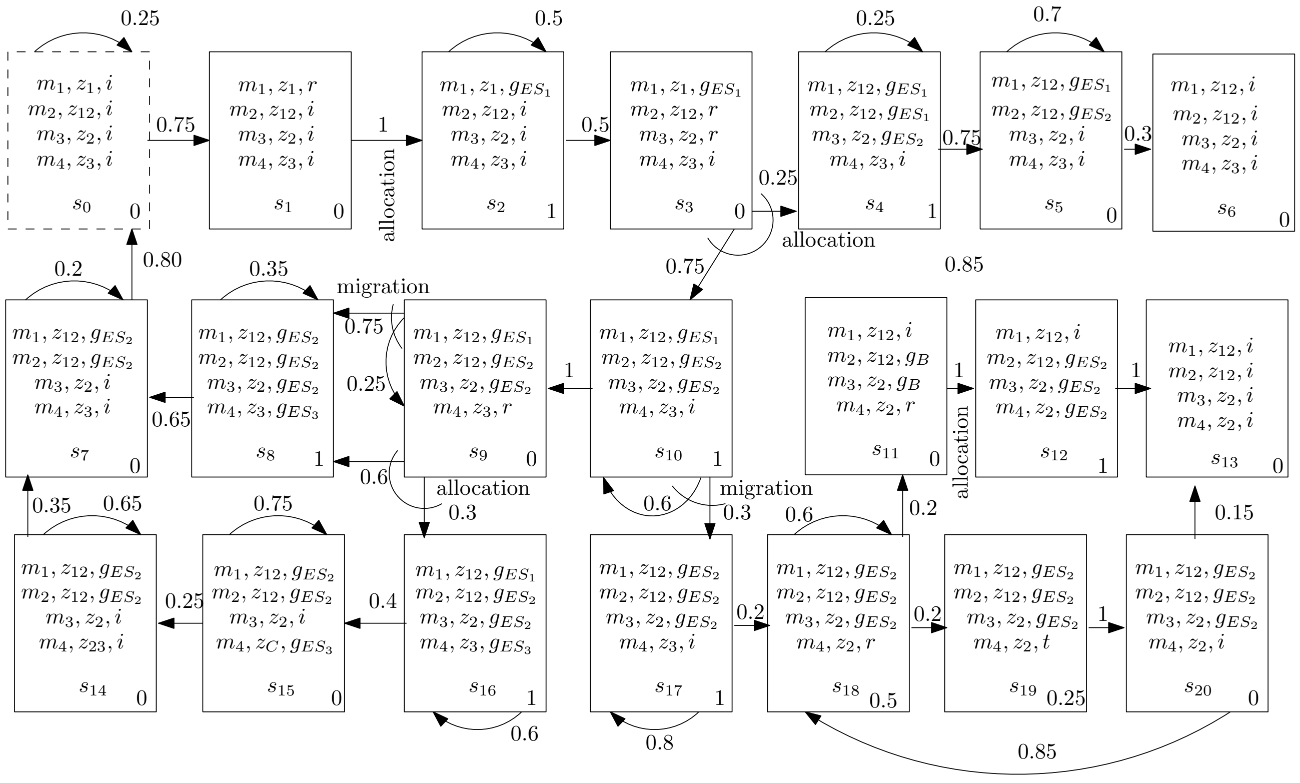


FIGURE 7.5: MDP Model of a Representative Policy

migrate the service request or to retain the original request-server binding. Thus, for *migration* actions, self-loops denote decisions corresponding to migrations not being undertaken in that state. Additionally, consider state s_9, where m_1, m_2 and m_3's requests have been acknowledged and m_4 has additionally initiated a service request. In such a scenario, both *migration* and *allocation* decisions are applicable. Successor states for the *migration* action correspond to the probability distribution over outgoing transitions to s_8 and s_9. Similarly, successor states for the *allocation* action correspond to the probability distribution to s_8 and s_{16}. All other scenarios which do not occur in the logs, occur with probability 0 similar to the DTMC representation and are not represented explicitly, thereby allowing a space efficient MDP. States s_0 and s_1 depict changes in states of the devices and hence are assigned a 0 reward value. In order to calculate the rewards assigned to states resulting from the agent's actions, consider s_2 which represents the resulting system state occurring out of the *allocation* action taken from s_1. Since there is only one device in s_2 whose request has been successfully allocated by the agent, we assign a reward of 1. Additionally, consider s_{19}, where the request from m_4 times out. A reward of 0.5 is assigned to such a state accounting for the time out since $N_A = 3, N_T = 1$ and $N_R = 4$. Each reward value thus corresponds to R_s. ■

Algorithm 10 outlines the MDP construction process. Lines 9-18 of the algorithm calculate the bit vector representation of each state similar to the DTMC construction. Lines 20-28 compute the transition probabilities for each state by calculating frequency distributions of each of the actions. The actions are categorized into three sets S_{alloc}, S_{mig} and S_{device} corresponding to allocation, migration and device actions respectively over which the probability distributions are calculated and assigned. The MDP constructed is device induced similar to the DTMC, however, each action of the policy is represented uniquely in the MDP allowing for their individual characterization.

7.3.3 Modeling MEC Environment Interactions and Server Failures

The DTMC and MDP formulations characterize how an allocation policy behaves with respect to user service requests and the corresponding server bindings. However, they do not incorporate any information with respect to server failures or how the allocation policy performs in failure scenarios. We model server failures and capture the interactions between the different entities in the MEC environment as a Stochastic Multi-Player Game (SMG) as in Chapter 6. An SMG, unlike DTMCs and MDPs, allows quantitative characterization of each player which can be used to yield insights into allocation policy behaviour as we demonstrate in our experiments.

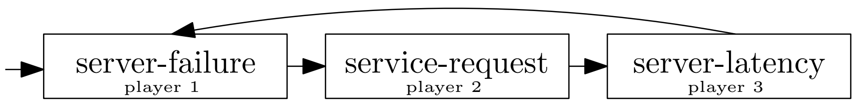


Figure 7.6: SMG Model of MEC environment

We model the MEC environment as an SMG comprising three players: *server-failure*, *service-request* and *server-latency* as depicted in Figure 7.6. The game begins with the player *server-failure* generating the number of active servers according to the probability distribution of failures that can occur. It is then the turn of the player *service-request* to generate user service request invocations. Finally, the control is transferred to player *server-latency* which generates the latencies associated with the service requests generated by the player *service-request* in the previous turn. The control is then transferred back to the player *server-failure* and the entire cycle is repeated.

7.3.3.1 State Representation of SMG

The state space of the SMG G represents all possible resource-allocation scenarios which can occur in an MEC scenario. Figure 7.7 depicts the representation of each state of the SMG G. Each state is defined as a tuple $\langle \mathcal{T}, \mathcal{C}, \mathcal{U}, \mathcal{L} \rangle$ representing Atomic Propositions (APs) corresponding to player turn, a counter representing the number of non-faulty servers, user service

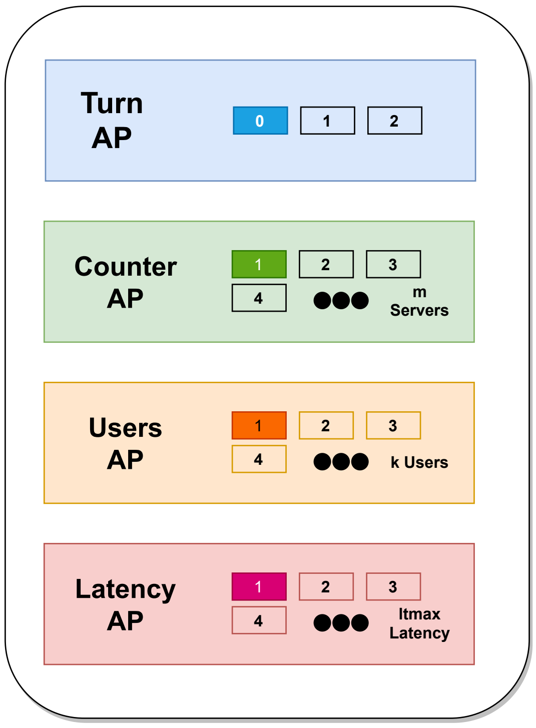


Figure 7.7: State Space of SMG

request invocations and latency respectively. Unlike the earlier DTMC and MDP models, where we considered Boolean APs, for SMGs, we do not consider Boolean APs for the SMG G. The combination of each AP's value represents a particular state's identity. Such identities are depicted with a dark shaded box for each AP as shown in Figure 7.7. The APs we utilize in our model comprise the following:

- TurnAP: Represents which player's turn it is.
- Counter AP: Represents number of non-faulty servers.
- Users AP: Represents number of mobile device users' service request invocations.
- Latency AP: Represents the average latency incurred as a result of the service allocation.

In the state diagram in Figure 7.7, for each player, we depict only a few APs for brevity. For example, for player *server-failure*, we depict 4 integer valued APs and assume there are a total of m possibilities denoted by the three dots within the Counter AP green rectangle representation. This is identically represented for the other players and the corresponding APs. While the Turn APs and the Counter APs represent integer turn and preference values, we utilize a discretized interval representation, similar to Chapters 4 and 5, for Users APs and Latency APs. We use *interval* to represent the value of the discretization interval. The actual values of APs are mapped to the discretized representation as follows: when the value of the APs is in the interval [0 and *interval*), inclusive of 0 but exclusive of *intervals*, the value of the AP is set to 1; when the value of the APs is between $[interval, 2 \times interval)$, the value of the AP is set to 2 and so forth. When the values of the APs exceed $m \times interval$, the AP is set to m. We use such a discretization approach to circumvent individual AP representation which can lead to state space explosion.

7.3.3.2 Models for each SMG player

We model each player of the SMG as an MDP. Each MDP model is then composed to a singleton unit representing the full state and transition space of the SMG.

Model of Server Failures

We utilize an MDP to model all possible server failures as a probability distribution. The player *server-failure* is indicated by the value of the Turn AP $\mathcal{T}$ as 0. Server Failures are represented by the Counter AP whose value ranges from 1 to m, where m is the maximum number of MEC servers under consideration. The Counter AP denotes the number of non-faulty servers at any timepoint t. Figure 7.8 depicts the MDP representation of server failure choices. The probabilities $p_1, p_2, \ldots p_m$ represent a probability distribution over the stochastic server failures where p_1 represents the probability that only one server is active while the remaining have failed, p_2 represents the probability that two servers are active while the remaining have failed while p_m represents the probability that there are no faulty servers. Note that $p_1, p_2, \ldots p_m$ represents a probability distribution. The value of the Counter AP $\mathcal{C}$ is updated according to the probability distribution formally defined as follows:

$$\sum_{i=1}^{m} p_i = 1 \tag{7.1}$$

The value of the Turn AP is set to 1 to indicate it is next the turn of the player *service-request*.

Model of User Service Request Invocations

The player *service-request* is indicated by the value of the Turn AP $\mathcal{T}$ as 1. Service invocations from users of mobile devices are modeled identically to server failures. We utilize an MDP to model all possible service request invocations of users as a probability distribution. However,

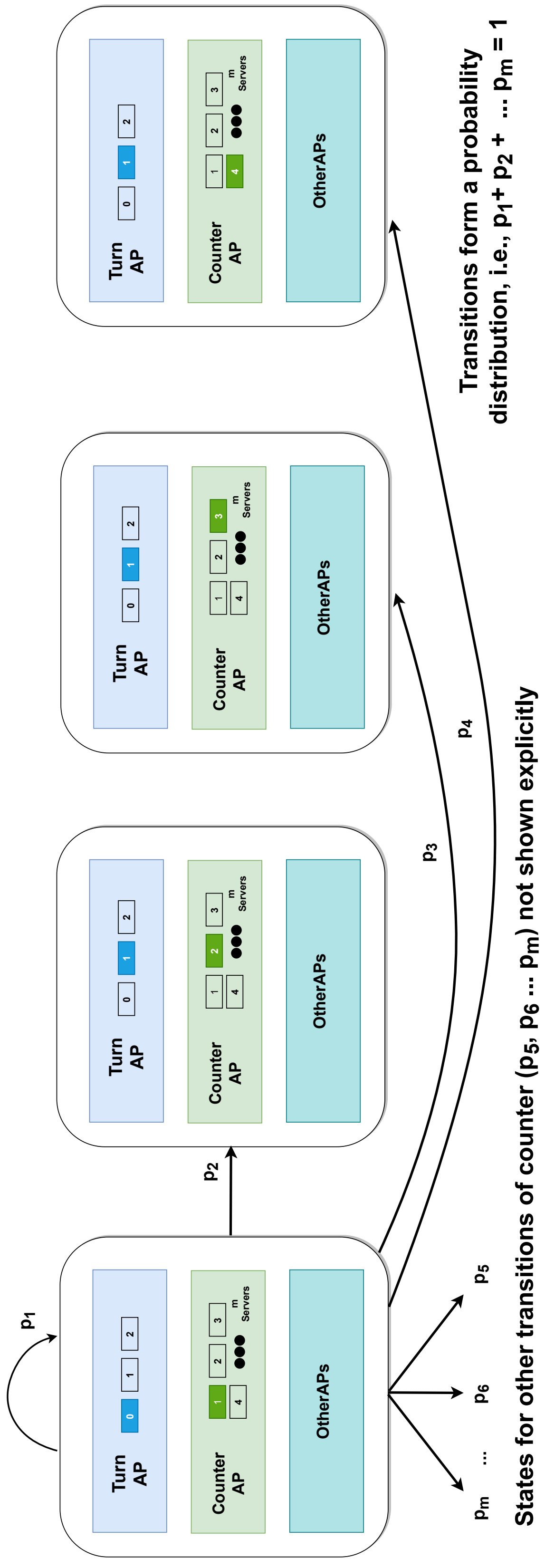

Turn AP
Counter AP
OtherAPs
m Servers
p_1
p_2
p_3
p_4
p_5
p_6
...
p_m
Transitions form a probability distribution, i.e., $p_1 + p_2 + \dots p_m = 1$
States for other transitions of counter (p_5, p_6 ... p_m) not shown explicitly

Figure 7.8: MDP Model of Server Failures

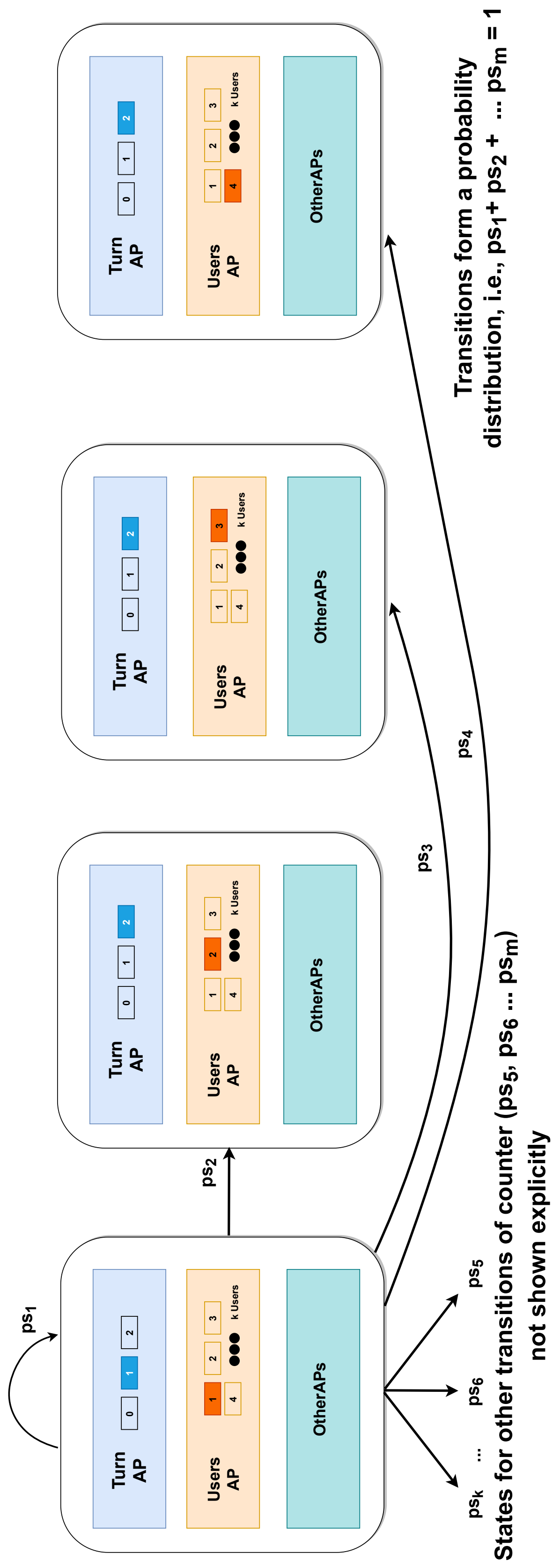

Turn AP
Users AP
OtherAPs
k Users
ps_1
ps_2
ps_3
ps_4
ps_5
ps_6
...
ps_k
Transitions form a probability distribution, i.e., $ps_1 + ps_2 + \dots ps_m = 1$
States for other transitions of counter (ps_5, ps_6 ... ps_m) not shown explicitly

Figure 7.9: MDP Model of User Service Invocations

we utilize the discrete intervalized representation for service requests instead of representing each user individually. We use $U_{interval}$ to represent the value of the discretization interval of users. The probabilities $ps_1, ps_2, \ldots ps_k$ represent a probability distribution over the number of users where ps_1 represents the probability that the number of users is between [0 and $U_{interval}$), ps_2 represents the probability that the number of users is between [$U_{interval}$ and $2 \times U_{interval}$) while ps_k represents the probability that the number of users exceeds $m \times U_{interval}$. The value of the User AP $\mathcal{U}$ is updated according to the probability distribution defined as follows:

$$\sum_{i=1}^{k} ps_i = 1 \tag{7.2}$$

where $ps_0, ps_1, \ldots ps_m$ represents the probabilities of all possible Service Request APs $\mathcal{SR}$. Figure 7.9 depicts the MDP model of the player *service-request*. The value of the Turn AP is set to 2 to indicate it is next the turn of the player *server-latency*.

Model of MEC Server

The player *server-latency* is indicated by the value of the Turn AP $\mathcal{T}$ as 2. The MDP can be viewed as comprising several blocks each guarded conditionally by the values of $\mathcal{C}$ and $\mathcal{U}$ generated by the players *server-failure* and *service-request* in the previous turns. For each unique value of $\mathcal{C}$,$\mathcal{U}$, we define a probability distribution $plt_1, plt_2, \ldots, pltmax$ over all possible latency APs, i.e. $\mathcal{L}$, defined as follows:

$$\sum_{i=0}^{ltmax} plt_i = 1, \forall\ \mathcal{C},\mathcal{U} \tag{7.3}$$

Each action thus generates latencies based on a probability distribution over latency distributions for each possible value of $\mathcal{P}$ and $\mathcal{SR}$. The probability distributions are thus conditionally defined on the values of $\mathcal{P}$ and $\mathcal{SR}$. Figure 7.10 depicts the MDP Model of User Service Invocations for $\mathcal{C} = 1$ and $\mathcal{U} = 1$. Similarly, Figure 7.11 depicts the MDP Model of User Service Invocations for $\mathcal{C} = 2$ and $\mathcal{U} = 1$. We utilize a probability distribution of latencies generated to analyze the average-case behaviour of allocation policies [37]. The value of the Turn AP is set to 0 to indicate the next round of the game.

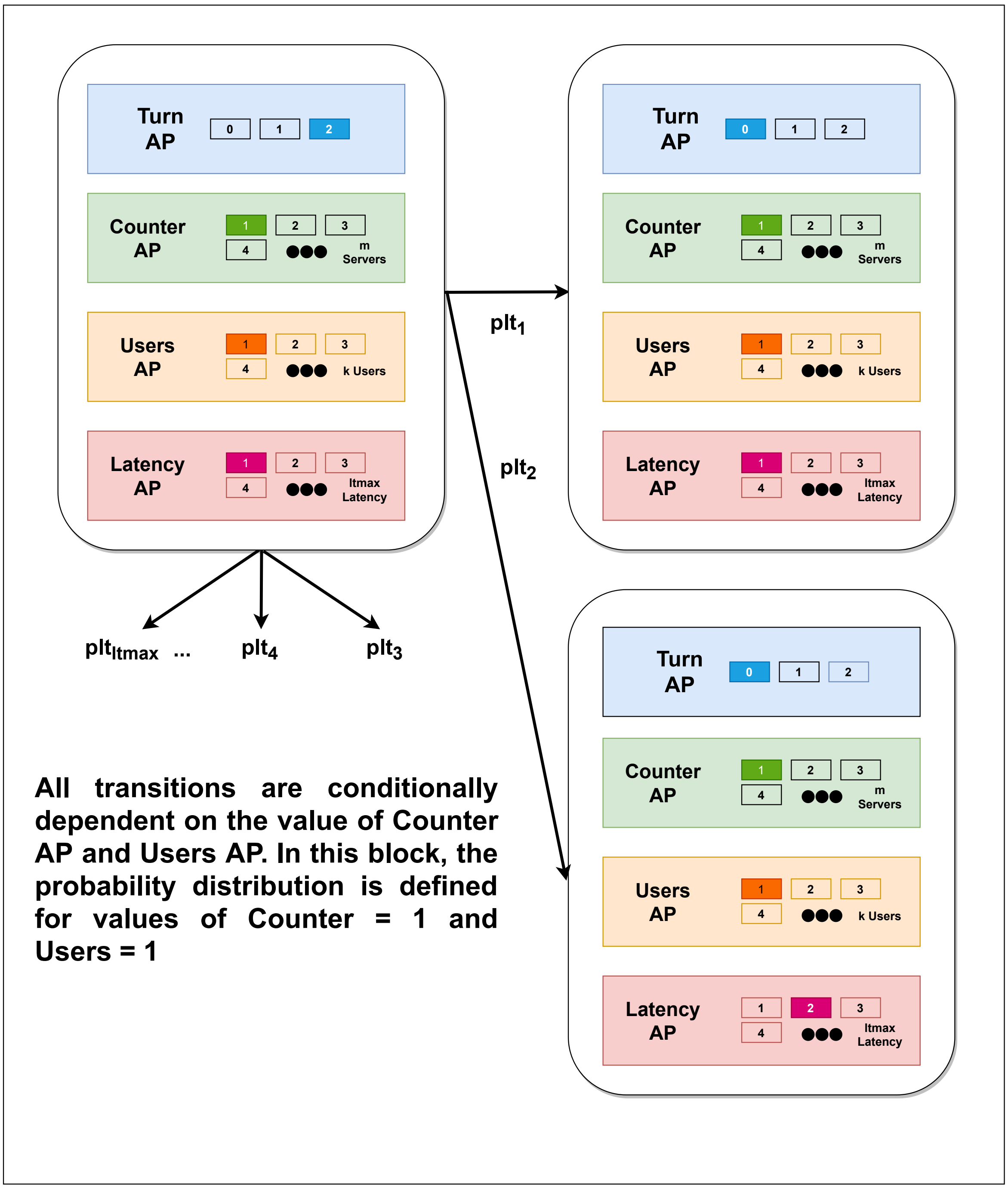

Turn AP
0 1 2
Counter AP
1 2 3 4 m Servers
Users AP
1 2 3 4 k Users
Latency AP
1 2 3 4 ltmax Latency
plt$_1$
plt$_2$
plt$_{ltmax}$ ...
plt$_4$
plt$_3$
All transitions are conditionally dependent on the value of Counter AP and Users AP. In this block, the probability distribution is defined for values of Counter = 1 and Users = 1

FIGURE 7.10: MDP Model of MEC Server for $\mathcal{C} = 1$ and $\mathcal{U} = 1$

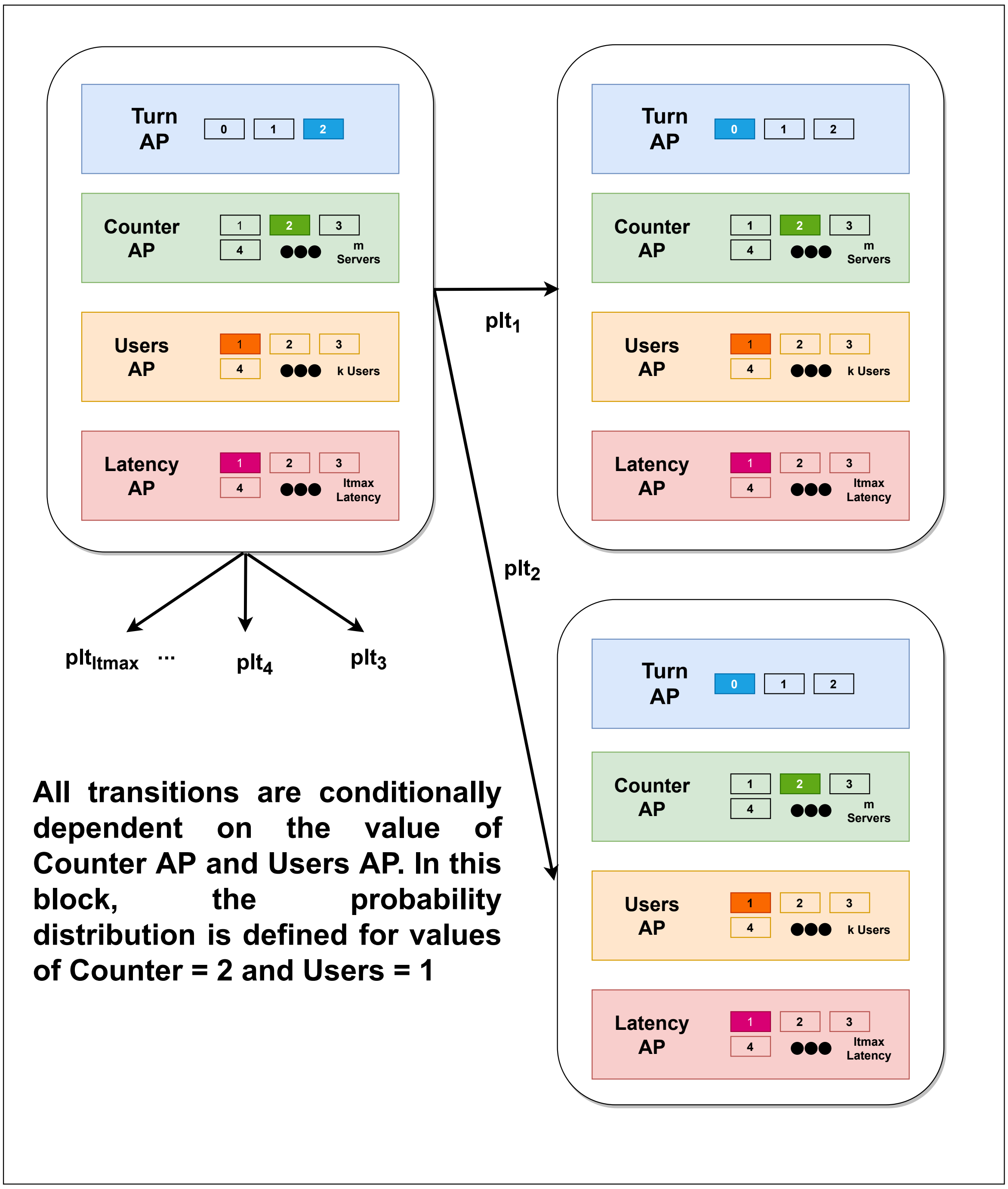

Turn AP
0 1 2
Counter AP
1 2 3 4 ●●● m Servers
Users AP
1 2 3 4 ●●● k Users
Latency AP
1 2 3 4 ●●● ltmax Latency
plt_1
plt_2
plt_{ltmax} ...
plt_4
plt_3
All transitions are conditionally dependent on the value of Counter AP and Users AP. In this block, the probability distribution is defined for values of Counter = 2 and Users = 1

FIGURE 7.11: MDP Model of MEC Server for $\mathcal{C} = 2$ and $\mathcal{U} = 1$

7.3.3.3 Model of Rewards

The probability distributions for each module described above is calculated from the logs available for the allocation policy. Thus, the above modules capture the characteristics of the latencies generated with respect to user service request invocation patterns along with the failure probabilities of the MEC servers. In order to quantitatively characterize how the policy behaves in such scenarios, we associate with each state a reward value depending on the latency observed. The reward values can be set independently according to the Latency AP to associate each latency value with its own reward. The reward values are utilized to characterize the performance of allocation policies in accordance with the distributions of failures, the service invocations and the generated latencies.

7.3.4 Verification of Allocation Policies

In the following, we discuss some characterization of allocation policies and their representations in PCTL and rPATL.

7.3.4.1 Request Waiting Time

When a user issues a service request, the request may be assigned to one of the servers right away, or the device may have to wait until one of the servers has the resources needed for service request allocation. Thus, a formula specifying a request waiting for a timeperiod T before being allocated to a server can be represented in terms of the bounded until operator U_T [37].

$$P_{\geq 0.8}[(m_1 \wedge r)\ U_{<=T}\ (m_1 \wedge g_{ES_1})]$$

The formula checks whether the probability that m_1 remains in the waiting state (denoted by the atomic proposition r) for the next T discrete time steps once it has invoked the service, is greater than 0.8. The Until operator ($U_{\leq T}$) is used to check for service request being granted by the edge server ES_1 within the T discrete steps.

7.3.4.2 Migration Policies

A migration-aware policy may choose to migrate a provisioned device's request to another server in the vicinity when a service request has been assigned to a server in a region that is serviced by multiple MEC servers. The number of potential $(request, allocation)$ pairs in such a scenario is exponential. Thus migrations can play a crucial role in determining the number of user service requests which can be allocated to MEC servers. We can encode $(user, server)$ bindings migrating between servers by the following PCTL formula:

$$P_{\geq 0.8}[(m_1 \wedge z_{12} \wedge g_{ES_1})\ U_{<=T}\ (m_1 \wedge z_{12} \wedge g_{ES_2})]$$

The formula describes the scenario where m_1's request which had already been provisioned by server ES_1, is migrated to server ES_2 due to its trajectory. The formula checks the probability being greater than 0.8 within timeperiod T.

7.3.4.3 Properties of Rewards

In case of RL policies, each state is associated with a reward value. Thus, we encode properties governing the behaviour of the policy with respect to reward evolution over discrete time steps.

$$R_{\geq 10}[F_{\leq 12}\ g_X]$$

The formula makes use of the eventually F operator, which denotes that along all execution paths, the specified condition is satisfied at some point of time in the execution. In this specific example, the formula checks along all executions, the accumulated reward is greater than 10 within 12 time steps in states considering the *grant* atomic proposition (g_X) where X refers to any edge server.

7.3.4.4 Properties of SMG

For the SMG, we consider properties for the player *service-request* since we are interested in analyzing the characterization of service request latencies for a service request distribution in the presence of multiple failures. We utilize the rewards operator since the generated latency is represented in the PRISM-Games rewards module.

$$\langle\langle service - request\rangle\rangle R_{\geq 10}[F\ latency = ltmax]$$

The formula makes use of the eventually F operator, which checks if the cumulative value of rewards for all states comprising a path is greater than 10. This is checked on all execution paths. As discussed above, we encode the desirable performance requirements as PCTL properties and check if a given model indeed honours them. In the following section, we discuss our experimental findings.

7.4 Results and Discussion

In this section, we first discuss our experimental setup and then analyze the results obtained.

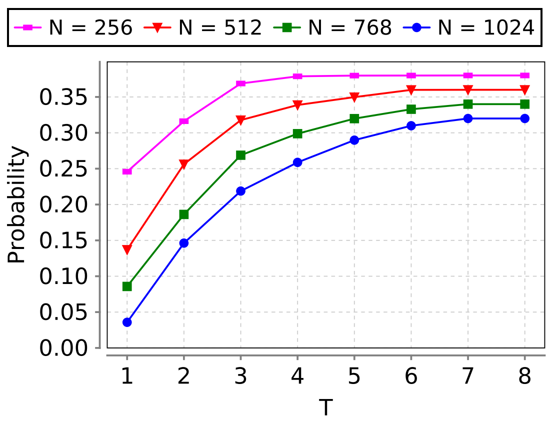
7.4.1 Experiment Design

In this chapter as well, we use the 'Existing Commercial Wireless Telecommunication Services Facilities in the San Francisco' dataset as edge site locations. Each edge site is associated with a single edge server as considered in the policies we evaluate. We use the PlanetLab availability dataset [101] to simulate server failures as described in Section 2.6.5. We perform simulation with the San Francisco taxi dataset. Each new coordinate update of taxis from the dataset is treated as a discrete time-step. Further, we use real world mobility and service request traces from the Telefonica dataset [102] to simulate human movement. We normalize the coordinates of the traces in the Telefonica dataset to conform to the server locations considered in the experiment. For each user, the timepoints of invocation of the Google Maps service is treated as a service request. We vary the number of users as 256, 512, 768 and 1024 [6]. We use the Planet Lab dataset to allocate generated latencies to service requests [125]. From the latency dataset, we assign a unique ID to the servers under consideration similar to the failure simulation setup. The latencies for each user to each server is considered as this uniquely assigned latency. The total latency is thus calculated as the weighted latency where the weight considered is the number of users connected to the server. The allocation policy is always running in the background, determining the request server binding and migrating users to other servers as required, while taking into account user mobility. To analyze the impact of failures, we consider a particular MEC server along with all the servers in its neighbourhood having overlapping service zones. Such a setup allows effective characterization of both the impact of neighbourhood server failures as well as service requests generated in overlapping service zones. For a particular neighbourhood comprising m servers, we consider λ-Fault-Tolerance, where $\lambda < m$. λ-Fault-Tolerance implies at most λ number of servers can fail simultaneously. All experiments are carried out on a machine with an Intel Core i7 Processor with 16GB RAM using Python 3.7 and PRISM as the Model Checking tool. PRISM models for the Turn-Based SMG are presented in Appendix C. PRISM additionally supports calculating the probability of a particular property as opposed to checking probability bounds which we utilize in our experiments.

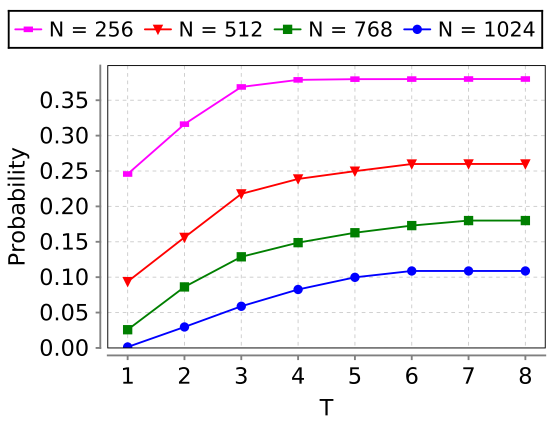
7.4.2 Analysis and Discussion

We use two classical allocation policies, MobMig [6] and Greedy [7] on the experimental setup. MobMig is a mobility aware allocation and migration strategy based on direction vectors. It produces allocations based on a ranking based fitness function taking into consideration the direction of movement and distance of users from MEC servers. Additionally, it uses migration to reallocate requests away from overloaded servers. The greedy algorithm assigns requests to the servers having the maximum resource availability. Further, we use two RL based policies, Migration-Aware [62] and Delay-Aware [9] on the experimental setup to demonstrate that our model can be used to compare performance metrics of service allocation policies in MEC systems quantitatively. The Migration-Aware strategy is based on the model-free approach where the reward function is based on the distance between the location of the user and the allocated MEC server binding along with the migration cost of reallocating requests to different servers. The Delay-Aware strategy is also a model-free approach where the reward function additionally takes into account the computational cost associated with a task and the communication delay between users and MEC servers. It also incorporates allocating service requests by MEC servers to users which are not located within the vicinity of the coverage area by making use of a backbone communication network. Additionally, it makes use of the backend network to connect a cloud server to accommodate requests which could not be catered by MEC servers. As such, for each granted service request, we normalize the estimated reward value from the logs between 0 and 1 to represent the distance between the nearest MEC server and the server where the request is provisioned. Without such a normalization, each granted request would be assigned a reward value 1 as in case of the previously examined policies. We consider the highest latency from the PlanetLab dataset as the latency for accessing the backend cloud.

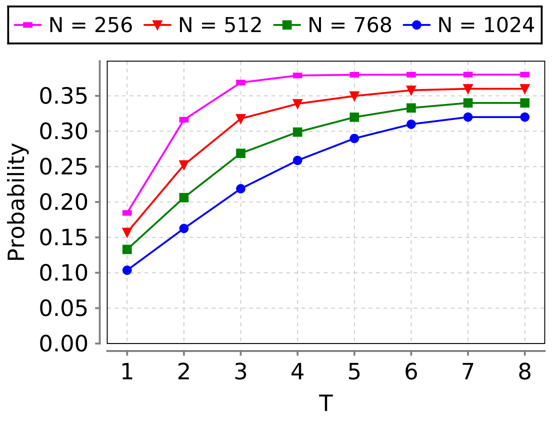
Figure 7.12 depicts the probability that a request from a user is not allocated to a server within T timesteps as specified by the property in Section 7.3.4.1. As the number of users increase, there is greater contention for access to server resources, thus the probability of the property being satisfied decreases. Hence, as depicted by the model checking algorithm, the probabilities decrease with the number of users. With increase in T, the probability values increase as there are more requests being invoked with fixed number of servers. There is a greater variation in probabilities with the greedy algorithm as compared to MobMig. Such a variation depicts the algorithm's sensitivity to increase in number of users when there is greater contention from a high number of users. With the Delay-Aware algorithm, it is interesting to note that for $N = 512$, the waiting time probabilities are higher than those for $N = 768$. Such a scenario depicts the dependence of latencies with respect to user service request distribution patterns.



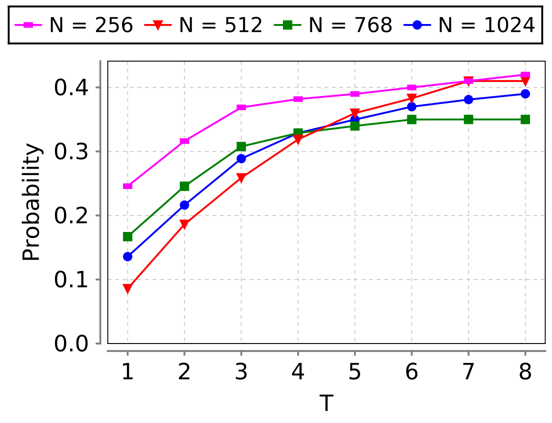
(a) MobMig



(b) Greedy



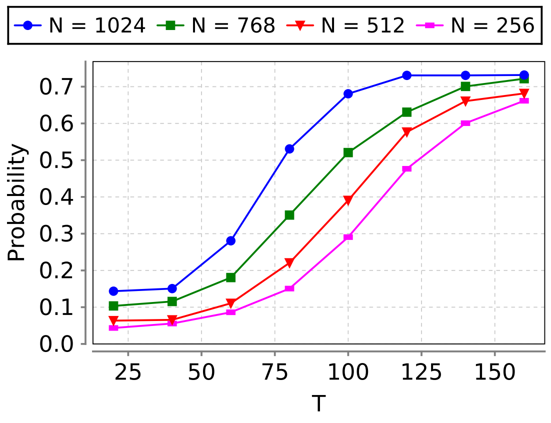
(c) Migration-RL



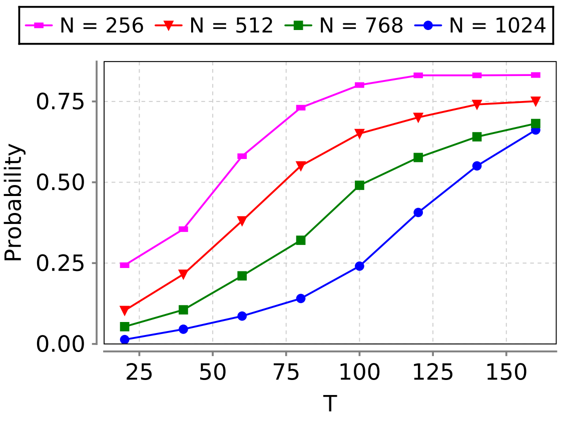
(d) Delay-RL

FIGURE 7.12: Probabilities of Request Waiting Time

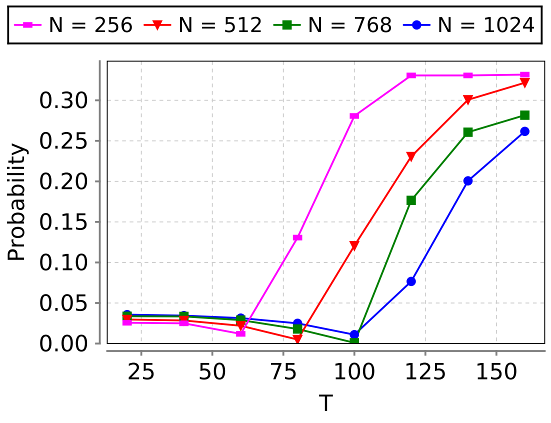
Figures 7.13 and 7.14 consider scenarios involving migration of service requests and depicts probability plots for the categories of properties outlined in Section 7.3.4.2. Differences between the two request traffic patterns account for the behaviour of the policies with the different mobility patterns. With increase in T, the probability of a migration *increases*. In case of vehicular mobility patterns, MobMig experiences a sharper increase in probability of migrations while eventually reaching a steady state value. The same is observed by varying the number of users where the probability of service migrations remains high. However, with pedestrian mobility patterns, the probability of service migrations does not increase as sharply. The fitness function of MobMig, being based on direction vectors between the moving direction and the server location can thus have an adverse affect on service migrations considering mobility patterns. This is because vehicular traffic which accounts for rapid changes in directions on the basis of road networks causes rapid fluctuations in fitness values involving direction vectors. A similar trend is observed with the greedy algorithm. This motivates the design of fitness functions which learn from and adapt to changing mobility and service invocation patterns. It is interesting to note that the probabilities *decrease* initially after which there is an increase, as opposed to



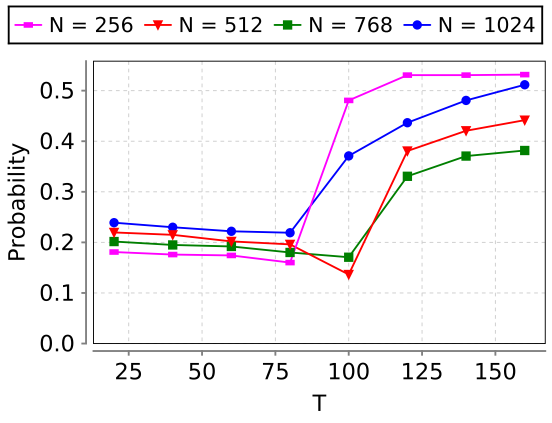
(a) MobMig



(b) Greedy



(c) Migration-RL

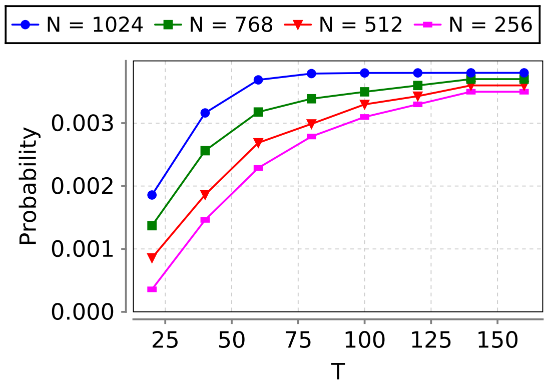


(d) Delay-RL

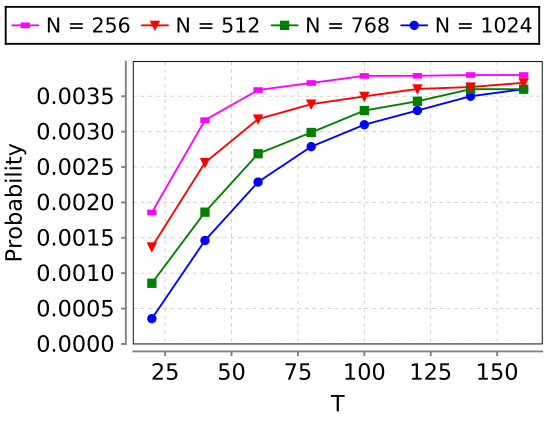
FIGURE 7.13: Probabilities of Migration for Vehicular Traffic Patterns

classical strategies where the probabilities remain nearly steady initially. Such a characteristic depicts the adaptive nature of RL policies towards learning request-server bindings according to the varied distributions of patterns in the service traffic dataset.

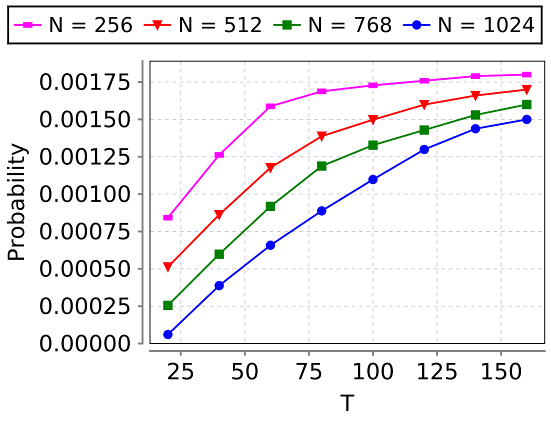
Figure 7.15 plots the reward values as discussed in Sections 7.3.4.3 and 7.3.4.4. With increasing number of users, the reward witnesses a general increasing pattern since more requests are invoked over time leading to a steady accumulation of reward values as inferred from Figure 7.15a. The Delay-Aware RL performs better than the Migration-Aware RL in all the scenarios under consideration since the fitness function takes into consideration the computation cost as well. We now analyze the impact of failures on the policies. Figure 7.15b depicts the obtained rewards for the different policies with varying number of users. The reward values are lower for MobMig and the greedy policy as compared to the two RL policies depicting their adaptive nature even in the presence of failures. For certain scenarios the reward obtained for MobMig is higher as compared to the greedy policy while for certain other scenarios the converse is true. This occurs since the fitness function of MobMig only considers the direction of movement of



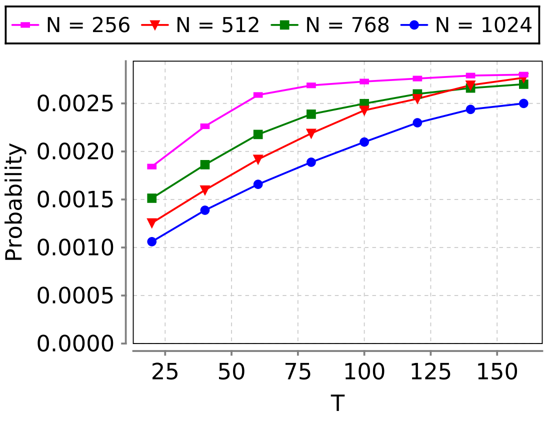
(a) MobMig



(b) Greedy



(c) Migration-RL

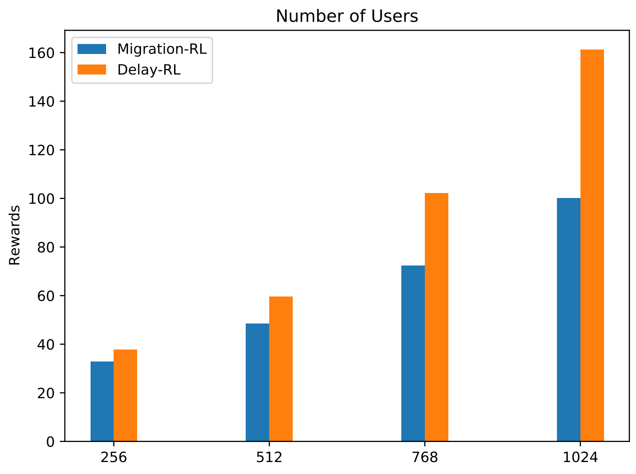


(d) Delay-RL

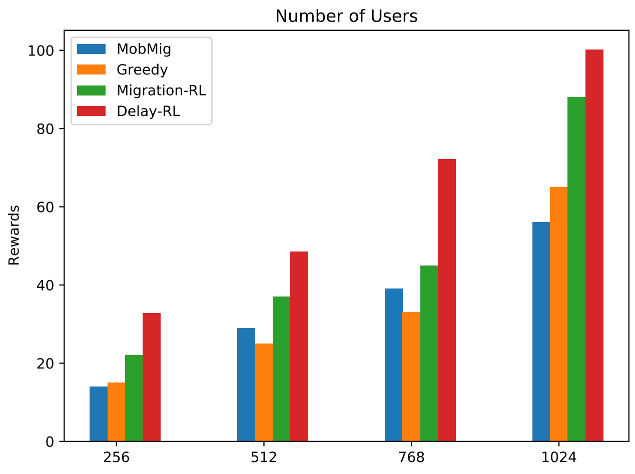
FIGURE 7.14: Probabilities of Migration Properties for Pedestrian Traffic Patterns

users while ignoring the current availability status of servers. In scenarios when the server is available, MobMig fares quite well, outperforming the greedy strategy.

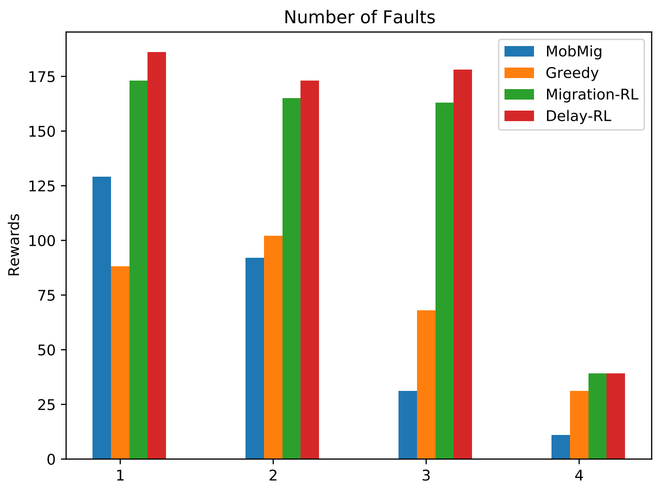
Figure 7.15c depicts the rewards obtained for the different policies varying the number of simultaneous failures. The rewards obtained for the RL based policies are significantly higher as compared to MobMig and the greedy policy especially when the number of failures is 1, 2 or 3. The Delay-Aware RL performs the best among all the four policies. However, when the number of failures is 4, the Delay-Aware RL performs similar to the Migration-Aware RL. The rewards obtained for the RL based policies depict the robustness and the adaptive nature of such policies to learn server allocations with an MEC fault distribution. However, an interesting note is that when the number of failures changes from 3 to 4, the robust performance of the RL based policies significantly drops off. This signifies a threshold, beyond which the RL based policies suffer significantly in terms of latency due to the non-availability of resources.



(a) Rewards for Allocated Requests



(b) Rewards for SMG



(c) Rewards for Multiple Failures in SMG

FIGURE 7.15: Reward Properties

7.5 Conclusion

In this chapter, we propose a novel trace driven learning framework to model MEC service allocation policies. Such an approach allows us to characterize and derive performance guarantees for properties of interest. As opposed to earlier work in modeling and characterization, we do not individually model policies, rather propose a generalized model which can be generated from execution logs of MEC allocation policies. Further, we use the logs to model interactions between MEC system components which are utilized to analyze how allocation policies behave in a failure sensitive MEC setup. We use real world traces to experimentally validate our approach and demonstrate the working on several allocation policies. We believe that our framework can guide an edge site designer in benchmarking and selecting the allocation policy to deploy in an MEC environment.

Chapter 8

Conclusion and Future Work

This thesis mainly focuses on the design and verification of policies for MEC. We propose several improvements in design of policies over traditional approaches. Additionally, we propose a verification framework to model and quantitatively verify performance metrics of policies. Further, we conduct extensive simulation experiments on real-world datasets to validate our approaches. In conclusion, the following issues have been addressed here:

- In our first work, we consider service allocation for microservice based applications. We consider scenarios where the correlation between the microservice workflow structure and the geo-spatial distribution of microservice container deployments, can have a critical impact on service allocation. We propose an abstraction refinement approach to provide speed-ups over a naive ILP approach. We consider scenarios where the locations of users are known and a latency estimate associated with each microservice is available.

- In our second work, we demonstrate the benefits of proactively prefetching microservices considering the workflow structure of microservice based applications. We utilize an MDP to model all possible proactive deployments induced by user-mobility and leverage Dyna-Q learning to determine the rewards associated with each prefetching action. Further, we design a heuristic to cater to resource constraints associated with each MEC server. In this work, we consider applications with linear workflows.

- In our third work, we propose a Safe Reinforcement Learning based auto-scaling policy agent that ensures load-balancing. We model the MEC environment using a Markov Decision Process and encode application service specific latency requirements in LTL. We leverage Safe Q-learning to determine the rewards associated with each auto-scaling action to maximize the probability of satisfying the latency requirements.

- In our fourth work, we consider the issue of fault tolerance. We propose a formal methods driven local recovery policy for high-priority applications. We use Stochastic Multi-Player Games as a formal model and specify objectives in temporal logic for verification with a Probabilistic Model Checker. For lower priority applications, we resort to a global recovery strategy by designing a greedy heuristic considering each server's failure probability.
- In our final work, we develop a framework for verification of service allocation policies. We propose a trace driven approach to derive a formal model of allocation policies and perform quantitative verification to produce probabilistic guarantees on performance metrics. We encode performance metrics of service allocation policies in temporal logic. Our framework allows characterization of service allocation policies without the necessity of modeling individual characteristics of service allocation policies.

In our work, we leverage on formal methods for the design and verification of MEC policies. The use of formal techniques enables a unique standpoint of the MEC problems and allows us to design and characterize policies with formal guarantees. The problems we study in this thesis are state of the art in the MEC context. We believe that our methods and standpoint will open up a lot of interesting avenues in MEC research going ahead. Some future directions building on our work are outlined below.

- When designing service allocation policies we assume full-observability, i.e, the locations of users and latency estimates are available apriori. In certain scenarios, full information about the state of the MEC system may not be available, for example, the current resource availability of all servers under consideration may not be available. In such scenarios, a possible future direction is to formally model the MEC system using a Partially Observable Markov Decision Process (POMDP) and utilize the model to synthesize allocation policies.
- While we demonstrate the benefits of proactively prefetching microservices to reduce user-perceivable latencies, we only cater to applications restricted to linear workflows. We plan to extend our approach to applications with graph based workflows while simultaneously incorporating server capacity constraints into our formal model.
- Our auto-scaling formulation assumes that the data required for each application such as models for machine learning, are co-located with the application at run-time and the number of data instances are identically scaled with the applications. As future work, we propose to design auto-scaling policies with segregated application and data instances in scenarios where application and data instances are not necessarily co-located.

- Finally, we plan to extend our trace-driven verification framework to analyze MEC resource allocation and computation offloading policies.

We believe that our work has immense potential in the MEC context especially in design and performance characterization through the formal lens. We also believe that our research will make inroads into large scale MEC policy design and analysis and pave the way for much wider MEC deployments going forward.

Chapter 9

Publications / Communications out of this work

Journals :

- *(Accepted)* Kaustabha Ray, Ansuman Banerjee and Swarup Kumar Mohalik, "Service Selection with Package Bundles and Compatibility Constraints", In IEEE Transactions on Services Computing, 2021, doi: 10.1109/TSC.2021.3075030.
- Kaustabha Ray and Ansuman Banerjee, "Modeling and Verification of Service Allocation Policies for Multi-Access Edge Computing Using Probabilistic Model Checking", In IEEE Transactions on Network and Service Management 18, no. 3 (2021): 3400-3414.
- Kaustabha Ray and Ansuman Banerjee, "Horizontal Auto-Scaling for Multi-Access Edge Computing Using Safe Reinforcement Learning", In ACM Transactions on Embedded Computing Systems 20, no. 6 (2021): 1-33.
- *(Accepted)* Kaustabha Ray and Ansuman Banerjee, "Prioritized Fault Recovery Strategies for Multi-Access Edge Computing Using Probabilistic Model Checking", In IEEE Transactions on Dependable and Secure Computing, 2022, doi: 10.1109/TDSC.2022.3143877.

Conferences :

- Kaustabha Ray and Ansuman Banerjee, "Trace-driven Modeling and Verification of a Mobility-Aware Service Allocation and Migration Policy for Mobile Edge Computing", In Proceedings of IEEE International Conference on Web Services (ICWS), pp. 310-317, 2020.

- Kaustabha Ray, Ansuman Banerjee, and Nanjangud C. Narendra, “Proactive Microservice Placement and Migration for Mobile Edge Computing”, In Proceedings of IEEE/ACM Symposium on Edge Computing (SEC), pp. 28-41, 2020.

Appendix A

Proofs of Theorems

We now reproduce the proof of Theorem 5.1 for which we need the following definitions:

A.1 Memoryless Policy, Induced Markov Chain , Bottom Strongly Connected Component and Expected Return of a Path

Definition A.1 [**MDP Policy:**]

A policy π for an MDP M is a function $\pi : S^+ \to \Lambda$. A policy is memoryless if it only depends on the current state, i.e. $\pi(\sigma[: n]) = \pi(\sigma[n])$ for any σ, and thus can be defined as $\pi : S \to \Lambda$. A Markov chain (MC) of an MDP M induced by a memoryless policy π is a tuple $M_\pi = (S, T_\pi, AP, L)$ where $T_\pi(s, s') = T(s, \pi(s), s')$ for all $s, s' \in S$. A bottom strongly connected component $(BSCC)$ of an MC is a strongly connected component with no outgoing transitions.

Let $R : S \to \mathbb{R}$ be a reward function of the MDP M. Then, for a discount factor $\gamma \in (0,1)$, the K-step return ($K \in \mathbb{N}$ or $K = \infty$) of a path σ from time $t \in \mathbb{N}$ is

$$Ret_{t:K}(\sigma) = \sum_{i=0}^{K} \gamma^i R(\sigma[t+i]), Ret_t(\sigma) = \lim_{K\to\infty} Ret_{t:K}(\sigma)$$

Under a policy π, the value of a state s is defined as the expected return of a path $-$ i.e.,

$$v_\pi(s) = \mathbb{E}_\pi \left[Ret_t(\sigma) \mid \sigma[t] = s \right]$$

for any fixed $t \in \mathbb{N}$.

A.2 Proof of LTL satisfaction on an MDP

Henceforth, for simplicity, the product MDP M' is denoted by $M = (S, T, AP, L, R, B)$ and the superscript $'$ is omitted for simplicity. Theorem 5.1 is restated as below.

Theorem A.2.1. For a given MDP M with $B \subseteq S$, the value function v^π for policy π and discount factor $0 < \gamma < 1$ satisfies $\lim_{\gamma\to 1^-} v^\pi(s) = Pr_\pi(s \models GF\ (B))$ for all states $s \in S$, if the return of a path is defined as

$$Ret_t(\rho) = \sum_{i=0}^{\infty} R_B(\rho[t+i]) \prod_{j=0}^{i-1} \Gamma_B(\rho[t+j]) \tag{A.1}$$

where $\prod_{j=0}^{-1} = 1$, $R_B : S \to [0,1)$ and $\Gamma_B : S \to (0,1)$ are the reward and discount functions defined as:

$$R_B(s) = \begin{cases} 1 - \gamma_{B'} & s \in B \\ 0 & s' \notin B \end{cases}, \Gamma_B(s) = \begin{cases} \gamma_B & s \in B \\ \gamma & s \notin B \end{cases} \tag{A.2}$$

and $\gamma_B = \gamma_B(\gamma)$ is a function of γ such that $\lim_{\gamma\to 1^-} \frac{1-\gamma}{1-\gamma_B(\gamma)} = 0$ ♦

Before proving Theorem A.2.1 some bounds are developed on $Ret_t(\sigma)$.

Lemma A.2.1. For all paths and $Ret_t(\sigma)$ from Equation A.2 it holds that

$$0 \leq \gamma Ret_{t+1}(\sigma) \leq Ret_t(\sigma) \leq 1 - \gamma_B + \gamma_B Ret_{t+1}(\sigma) \leq 1$$

Proof: Since there is no negative reward, $Ret_t \geq 0$ holds. By the return definition, replacing γ with 1 yields a larger or equal return, which constitutes the following upper bound on the return: $Ret_t(\sigma) \leq 1 - \gamma_B^b \leq 1$, where b is the number of B states visited. Return $Ret_t(\sigma)$ from A.1 satisfies

$$Ret_t(\sigma) = \begin{cases} 1 + \gamma_B \left(Ret_{t+1}(\sigma) - 1\right) & \sigma[t] \in B \\ \gamma Ret_{t+1}(\sigma) & \sigma[t] \notin B \end{cases} \tag{A.3}$$

From $Ret_t(\sigma) \leq 1$ it follows that $1 + \gamma_B \left(Ret_{t+1}(\sigma) - 1\right) \geq \gamma Ret_{t+1}(\sigma)$, which with (A.3) proves the other inequalities. ♦

Lemma A.2.1 implies that replacing a prefix of a path with states belonging to B never decreases the return of a path and similarly replacing with states that do not belong to B never increases the return. The result is particularly useful for establishing the upper and lower bounds on the value of a state.

The next lemma shows that under a policy, the values of states in the accepting BSCCs of the induced Markov chain approach 1 in the limit; thus, is the key to proving Theorem A.2.1.

Lemma A.2.2. Let $BSCC\left(M_\pi\right)$ denote the set of all $BSCC$s of an induced Markov chain M_π and let B_π denote the set of B states that belong to a $BSCC$ of M_π– i.e.

$$B_\pi := \{s \mid s \in B, s \in T, T \in BSCC\left(M_\pi\right)\}$$

Then, for any state $s \in B_\pi$

$$\lim_{\gamma \to 1^-} v_\pi^\gamma(s) = 1$$

Proof. For any fixed $t \in \mathbb{N}$, let N_t be the stopping time of first returning to the state $s \in S$ after leaving it at t

$$N_t = \min\{\tau \mid \sigma[t+\tau] = s, \tau > 0\}$$

Since under a policy π, the value of a state s is defined as the expected return of a path, i.e., $v_\pi(s) = \mathbb{E}[Ret_t(\sigma)|\sigma(t) = s]$, the following holds:

$$\begin{aligned} v_\pi^\gamma(s) &= 1 - \gamma_B + \gamma_B \mathbb{E}_\pi \left[Ret_{t+1}(\sigma) \mid \sigma[t] = s\right] \\ &= 1 - \gamma_B + \gamma_B \mathbb{E}_\pi \Bigg[Ret_{t+1:t+N_t-1}(\sigma) \\ &+ \left(\prod_{i=1}^{N_t-1} \Gamma(\sigma[t+i])\right) \cdot Ret_{t+N_t}(\sigma) \mid \sigma[t] = s\Bigg] \end{aligned}$$

Since once a state $s \in B_\pi$ is visited, almost surely it is visited again. Using $Ret_t(\sigma) \geq \gamma Ret_{t+1}(\sigma)$, we obtain

$$\begin{aligned} v_\pi^\gamma(s) &\geq 1 - \gamma_B + \gamma_B \mathbb{E}_\pi \left[\gamma^{N_t-1} Ret_{t+N_t}(\sigma) \mid \sigma[t] = s\right] \\ &\geq 1 - \gamma_B + \gamma_B \mathbb{E}_\pi \left[\gamma^{N_t-1} \mid \sigma[t] = s\right] v_\pi(s) \quad \text{(By Markov Property [126])} \\ &\geq 1 - \gamma_B + \gamma_B \gamma^{\mathbb{E}_\pi[N_t-1|\sigma[t]=s]} v_\pi(s) \qquad \text{(By Jensen Inequality [126])} \\ &\geq 1 - \gamma_B + \gamma_B \gamma^n v_\pi(s) \end{aligned}$$

where $n \geq 1$ is a constant.

Now, since $v_\pi^\gamma(s) \geq 1 - \gamma_B + \gamma_B \gamma^n v_\pi(s)$ as proved above, substituting:

$$\begin{aligned} v_\pi^\gamma(s) &\geq \frac{1-\gamma_B}{1-\gamma_B\gamma^n} \\ &\geq \frac{1-\gamma_B}{1-\gamma_B(1-n(1-\gamma))} \\ &= \frac{1}{1+n\frac{1-\gamma}{1-\gamma_B} - n(1-\gamma)} \end{aligned}$$

where the second " $\geq$ " holds by $(1-(1-\gamma))^n \geq 1-n(1-\gamma)$ for $\gamma \in (0,1)$. Finally, since $v_\pi^\gamma(s) \leq 1$ by Lemma A.2.1 letting $\gamma, \gamma_B \to 1^-$ under the condition $\lim_{\gamma\to 1^-} \frac{1-\gamma}{1-\gamma_B(\gamma)} = 0$ results in $\lim_{\gamma \to 1^-} v_\pi^\gamma(s) = 1$ ♦

The proof of Theorem A.2.1 is now outlined below.

Proof of Theorem A.2.1. The expected return of a random path σ from a state $s \in S$ is determined by whether it visits the states $B \subseteq S$ infinitely often:

$$\begin{aligned} v_\pi^\gamma(s) \ &= \mathbb{E}_\pi[Ret_t(\sigma)|\sigma[t]=s, \sigma \models GF(B)]Pr_\pi(s \models GF(B)) \\ &\quad + \mathbb{E}_\pi[Ret_t(\sigma)|\sigma[t]=s, \sigma \not\models GF(B)]Pr_\pi(s \models GF(B)) \end{aligned} \tag{A.4}$$

for some fixed $t \in \mathbb{N}$. let M_t be the stopping time of first reaching a state in B_π after leaving s at t

$$M_t = \min\{\tau \mid \sigma[t+\tau] \in B_\pi, \tau > 0\}$$

where B_π is defined as in Lemma A.2.2. Then, it holds that

$$\begin{aligned} \mathbb{E}_\pi[Ret_t(\sigma)|\sigma[t]=s, \sigma \models GF(B)] &= \mathbb{E}_\pi[Ret_t(\sigma)|\sigma[t]=s, \sigma \models FB_\pi] \\ &\geq \mathbb{E}_\pi[\gamma^{M_t} Ret_{t+M_t}|\sigma[t]=s, \sigma \models GF(B)] \\ &\geq \mathbb{E}_\pi[\gamma^{M_t}|\sigma[t]=s, \sigma \models GF(B)] \ v_{\pi,min}^\gamma(B_\pi) \\ &\geq \gamma^{\mathbb{E}_\pi[M_t|\sigma[t]=s,\sigma\models GF(B)]} \ v_{\pi,min}^\gamma(B_\pi) \\ &\text{(above inequalities hold due to Lemma A.2.1,} \\ &\text{Markov property and Jensen's inequality)} \\ &= \gamma^m v_{\pi,min}^\gamma(B_\pi) \end{aligned}$$

where $v_{\pi,\min}^\gamma(B_\pi) = \min_{s\in B_\pi} v_\pi^\gamma(s)$ and m is a constant. Here, the first equality holds because a path $\sigma \models GF(B)$ almost surely eventually enters an accepting BSCC, it eventually reaches a state $s \in B_\pi$ almost surely. From Equation A.4:

$$v_\pi^\gamma(s) \geq \gamma^m v_\pi\left(B_\pi\right) \Pr_\pi(s \models GF(B)) \tag{A.5}$$

Similarly, let M'_t be the stopping time of first reaching a rejecting BSCC of M_π after leaving s at t. Then

$$\begin{aligned} M'_\pi = \min\{\tau \mid \sigma[t+\tau] \in T, T \cap B = \varnothing \\ T \in BSCC\left(M_\pi\right), \tau > 0\} \end{aligned}$$

denoting the number of time steps before a rejecting BSCC is reached. Thus, from Lemma 2 and the Markov property

$$\begin{aligned} \mathbb{E}_\pi[Ret_t(\sigma)|\sigma[t] = s, \sigma \not\models GF(B)] &\leq \mathbb{E}_\pi[1 - \gamma_B^{M'_\pi}|\sigma[t] = s, \sigma \not\models GF(B)] \\ &\leq 1 - \gamma_B^{\mathbb{E}_\pi[M'_\pi|\sigma[t]=s,\sigma\models GF(B)]} \\ &= 1 - \gamma_B^{m'} \end{aligned}$$

where m' is also constant. From this upper bound and Equation A.4,

$$v_\pi^\gamma(s) \leq Pr_\pi(s \models GF(B)) \; + 1 - \gamma_B^{m'} Pr_\pi(s \models GF(B)) \tag{A.6}$$

Both the above upper bounds in Equation A.6 and the lower bound from Equation A.5 approach the probability of satisfying the formula as the limit γ approaches 1 from below, thus concluding the proof. ♦

Appendix B

PRISM Models for Fault Recovery

This appendix details a PRISM implementation of the fault recovery SMG.

```
smg


//recovery time maximum number of steps

const int TM = 50;


//number of containers to recover

const int T = 5;


//goal boolean

global goal : bool init false;


//where the containers are deployed

global container1deploys2 : bool init false;

global container2deploys2 : bool init false;

global container3deploys2 : bool init false;
global container4deploys2 : bool init false;

global container5deploys2 : bool init false;


global container1deploys3 : bool init false;

global container2deploys3 : bool init false;

global container3deploys3 : bool init false;

global container4deploys3 : bool init false;

global container5deploys3 : bool init false;


global container1deploys4 : bool init false;

global container2deploys4 : bool init false;

global container3deploys4 : bool init false;

global container4deploys4 : bool init false;

global container5deploys4 : bool init false;


//memory requirement of each container

const int memcontainer1 = 3;

const int memcontainer2 = 4;

const int memcontainer3 = 4;

const int memcontainer4 = 3;

const int memcontainer5 = 2;


//users associated with each container

const int c1u = 56;

const int c2u = 12;

const int c3u = 5;

const int c4u = 68;

const int c5u = 70;
```

```
//maximum deployment time of a container

const int TMAX = 5;



//deployment time of each container

const int container1deploytime = 4;

const int container2deploytime = 3;

const int container3deploytime = 5;

const int container4deploytime = 2;

const int container5deploytime = 2;


//turn encodes the game turn based nature

global turn : [1..2] init 2;


//tick indicates the total recovery time

global tick : [-1..TM] init TM;


//trigger will be true

global timercontainer : [0..TMAX] init container1deploytime;


//server1 initialized with container

global requests1 : [1..T] init 1;


//server 2 initialized with capacity

global s2capacity : [0..20] init 20;


//server 3 initialized with capacity

global s3capacity : [0..20] init 20;


//server 3 initialized with capacity
```

```
global s4capacity : [0..20] init 20;


//player for servers failure

player p1 env endplayer


//player for error recovery protocol

player p2 erp, [ndchoice2], [ndchoice3], [ndchoice4] endplayer


//module of server failures

// 1 only S1 has failed

// 2 - 1 and 2 have failed

// 3 - 1 and 3 have failed

module env

    counter : [1..7] init 1;


    changetrigger : bool init false;

    //s1 error

    [f1]turn = 1 & counter = 1 -> 0.85 :(counter' = 1) & (turn' = 2) +

        0.05: (counter' = 2) & (turn' = 2) + 0.05 :(counter' = 3) & (turn' = 2) +

   0.05 : (counter' = 4) & (turn' = 2);


    //s1 and s2 error, choice remaining is only s3

    [f2]turn = 1 & counter = 2  & changetrigger = false -> (counter' = 2) & (turn'

   = 2)

        & (requests1' = 1) & (changetrigger' = true) & (s3capacity' = 20)

        & (container1deploys2' = false) & (container2deploys2' = false) &

   (container3deploys2' = false) & (container4deploys2' = false) &

   (container5deploys2' = false)
```

```
        & (container1deploys3' = false) & (container2deploys3' = false) &
    (container3deploys3' = false) & (container4deploys3' = false) &
    (container5deploys3' = false);

    //s1 and s3 error, choice remaining is only s2
    [f3]turn = 1 & counter = 3  & changetrigger = false -> (counter' = 3) & (turn'
    = 2)
        & (requests1' = 1) & (changetrigger' = true) & (s2capacity' = 20)
        & (container1deploys2' = false) & (container2deploys2' = false) &
    (container3deploys2' = false) & (container4deploys2' = false) &
    (container5deploys2' = false)
        & (container1deploys3' = false) & (container2deploys3' = false) &
    (container3deploys3' = false) & (container4deploys3' = false) &
    (container5deploys3' = false);

    //s1 and s2 error, choice remaining is only s3
    [f2]turn = 1 & counter = 2  & changetrigger = true -> (counter' = 2) & (turn' =
    2);



    //s1 and s3 error, choice remaining is only s4
    [f3]turn = 1 & counter = 3  & changetrigger = true -> (counter' = 3) & (turn' =
    2);
endmodule

module erp
    //trigger : bool init true;

    //when container has not yet been initialized, decrement counter value
```

```
 []turn = 2 & timercontainer > 0 & requests1 != T  -> (turn' = 1) &
(timercontainer'  = timercontainer - 1) & (tick' = tick - 1);


 []turn = 2 & requests1 = T & tick > -1 -> (goal'= true);


 //erp turn and choice is 2, move from request to 2 when counter = 0 (trigger)


 //for counter = 1 : following choices when s1 has failed and s2 and s3 are
active


 //for container 1
 [ndchoice2]turn = 2 & requests1 < T & (s2capacity - memcontainer1) > 0 &
     (s3capacity - memcontainer1) > 0 & (s4capacity - memcontainer1) > 0 &
counter = 1 & requests1 = 1 & timercontainer = 0 ->
     (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer1) &
(turn' = 1) & (tick' = tick - 1)
                 & (container1deploys2' = true) & (timercontainer' =
container2deploytime);


 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer1) > 0 &
     (s3capacity - memcontainer1) > 0 & (s4capacity - memcontainer1) > 0 &
counter = 1 & requests1 = 1 & timercontainer = 0 ->
     (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer1) &
(turn' = 1) & (tick' = tick - 1)
                 & (container1deploys3' = true) & (timercontainer' =
container2deploytime);


 [ndchoice4]turn = 2 & requests1 < T & (s2capacity - memcontainer1) > 0 &
     (s3capacity - memcontainer1) > 0 & (s4capacity - memcontainer1) > 0 &
counter = 1 & requests1 = 1 & timercontainer = 0 ->
```

```
     (requests1' = requests1 + 1) & (s4capacity' = s4capacity - memcontainer1) &
(turn' = 1) & (tick' = tick - 1)
                 & (container1deploys4' = true) & (timercontainer' =
container2deploytime);



 //for container 2
 [ndchoice2]turn = 2 & requests1 < T & (s2capacity - memcontainer2) > 0 &
         (s3capacity - memcontainer2) > 0 & (s4capacity - memcontainer2) > 0 &
counter = 1 & requests1 = 2 & timercontainer = 0->
     (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer2) &
(turn' = 1) & (tick' = tick - 1)
                 & (container2deploys2' = true) & (timercontainer' =
container3deploytime);

 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer2) > 0 &
     (s3capacity - memcontainer2) > 0 & (s4capacity - memcontainer2) > 0 &
counter = 1 & requests1 = 2 & timercontainer = 0->
     (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer2) &
(turn' = 1) & (tick' = tick - 1)
                 & (container2deploys3' = true) & (timercontainer' =
container3deploytime);

 [ndchoice4]turn = 2 & requests1 < T & (s2capacity - memcontainer2) > 0 &
     (s3capacity - memcontainer2) > 0 & (s4capacity - memcontainer2) > 0 &
counter = 1 & requests1 = 2 & timercontainer = 0->
     (requests1' = requests1 + 1) & (s4capacity' = s4capacity - memcontainer2) &
(turn' = 1) & (tick' = tick - 1)
                 & (container2deploys4' = true) & (timercontainer' =
container3deploytime);
```

```
 //for container 3
 [ndchoice2]turn = 2 & requests1 < T & (s2capacity - memcontainer3) > 0 &
        (s3capacity - memcontainer3) > 0 & (s4capacity - memcontainer3) > 0 &
counter = 1 & requests1 = 3 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer3) &
(turn' = 1) & (tick' = tick - 1)
                & (container3deploys2' = true) & (timercontainer' =
container4deploytime);

 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer3) > 0 &
    (s3capacity - memcontainer3) > 0 & (s4capacity - memcontainer3) > 0 &
counter = 1 & requests1 = 3 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer3) &
(turn' = 1) & (tick' = tick - 1)
                & (container3deploys3' = true) & (timercontainer' =
container4deploytime);

 [ndchoice4]turn = 2 & requests1 < T & (s2capacity - memcontainer3) > 0 &
    (s3capacity - memcontainer3) > 0 & (s4capacity - memcontainer3) > 0 &
counter = 1 & requests1 = 3 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s4capacity' = s4capacity - memcontainer3) &
(turn' = 1) & (tick' = tick - 1)
                & (container3deploys4' = true) & (timercontainer' =
container4deploytime);
```

```
 //for container 4
 [ndchoice2]turn = 2 & requests1 < T & (s2capacity - memcontainer4) > 0 &
        (s3capacity - memcontainer4) > 0 & (s4capacity - memcontainer4) > 0 &
counter = 1 & requests1 = 4 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer4) &
(turn' = 1) & (tick' = tick - 1)
                & (container4deploys2' = true) & (timercontainer' =
container5deploytime);

 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer4) > 0 &
    (s3capacity - memcontainer4) > 0 & (s4capacity - memcontainer4) > 0 &
counter = 1 & requests1 = 4 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer4) &
(turn' = 1) & (tick' = tick - 1)
                & (container4deploys3' = true) & (timercontainer' =
container5deploytime);

 [ndchoice4]turn = 2 & requests1 < T & (s2capacity - memcontainer4) > 0 &
    (s3capacity - memcontainer4) > 0 & (s4capacity - memcontainer4) > 0 &
counter = 1 & requests1 = 4 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s4capacity' = s4capacity - memcontainer4) &
(turn' = 1) & (tick' = tick - 1)
                & (container4deploys4' = true) & (timercontainer' =
container5deploytime);



 //for container 5
 [ndchoice2]turn = 2 & requests1 < T & (s2capacity - memcontainer5) > 0 &
```

```
        (s3capacity - memcontainer5) > 0 &  (s4capacity - memcontainer4) > 0 &
counter = 1 & requests1 = 5 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer5) &
(turn' = 1) & (tick' = tick - 1)
                & (container5deploys2' = true);

 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer5) > 0 &
    (s3capacity - memcontainer5) > 0 &  (s4capacity - memcontainer5) > 0 &
counter = 1 & requests1 = 5 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer5) &
(turn' = 1) & (tick' = tick - 1)
                & (container5deploys3' = true);

 [ndchoice4]turn = 2 & requests1 < T & (s2capacity - memcontainer5) > 0 &
    (s3capacity - memcontainer5) > 0 &  (s4capacity - memcontainer5) > 0 &
counter = 1 & requests1 = 5 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer5) &
(turn' = 1) & (tick' = tick - 1)
                & (container5deploys4' = true);





//--------------------------------------------------------------------------------
 //for counter = 2 : following choices when s1 and s2 have failed and s3 is
active

 //for container 1
 [ndchoice3]turn = 2 & requests1 < T & (s3capacity - memcontainer1) > 0 &
counter = 2 & requests1 = 1 & timercontainer = 0 ->
```

```
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer1) &
(turn' = 1) & (tick' = tick - 1)
                & (container1deploys3' = true) & (timercontainer' =
container2deploytime);

 //for container 2
 [ndchoice3]turn = 2 & requests1 < T & (s3capacity - memcontainer2) > 0 &
counter = 2 & requests1 = 2 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer2) &
(turn' = 1) & (tick' = tick - 1)
                & (container2deploys3' = true) & (timercontainer' =
container3deploytime);

 //for container 3
 [ndchoice3]turn = 2 & requests1 < T & (s3capacity - memcontainer3) > 0 &
counter = 2 & requests1 = 3 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer3) &
(turn' = 1) & (tick' = tick - 1)
                & (container3deploys3' = true) & (timercontainer' =
container4deploytime);

 //for container 4
 [ndchoice3]turn = 2 & requests1 < T & (s3capacity - memcontainer4) > 0 &
counter = 2 & requests1 = 4 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer4) &
(turn' = 1) & (tick' = tick - 1)
                & (container4deploys3' = true) & (timercontainer' =
container5deploytime);

 //for container 5
 [ndchoice3]turn = 2 & requests1 < T & (s3capacity - memcontainer5) > 0 &
counter = 2 & requests1 = 5 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s3capacity' = s3capacity - memcontainer5) &
(turn' = 1) & (tick' = tick - 1)
                & (container5deploys3' = true);



//--------------------------------------------------------------------------------
 //for counter = 3 : following choices when s1 and s3 have failed and s2 is
active


 //for container 1
 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer1) > 0 &
counter = 3 & requests1 = 1 & timercontainer = 0 ->
    (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer1) &
(turn' = 1) & (tick' = tick - 1)
                & (container1deploys3' = true) & (timercontainer' =
container2deploytime);


 //for container 2
 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer2) > 0 &
counter = 3 & requests1 = 2 & timercontainer = 0->
    (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer2) &
(turn' = 1) & (tick' = tick - 1)
                & (container2deploys3' = true) & (timercontainer' =
container3deploytime);


 //for container 3
 [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer3) > 0 &
counter = 3 & requests1 = 3 & timercontainer = 0->
```

```
        (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer3) &
    (turn' = 1) & (tick' = tick - 1)
                    & (container3deploys3' = true) & (timercontainer' =
    container4deploytime);

    //for container 4
    [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer4) > 0 &
    counter = 3 & requests1 = 4 & timercontainer = 0->
        (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer4) &
    (turn' = 1) & (tick' = tick - 1)
                    & (container4deploys3' = true) & (timercontainer' =
    container5deploytime);

    //for container 5
    [ndchoice3]turn = 2 & requests1 < T & (s2capacity - memcontainer5) > 0 &
    counter = 3 & requests1 = 5 & timercontainer = 0->
        (requests1' = requests1 + 1) & (s2capacity' = s2capacity - memcontainer5) &
    (turn' = 1) & (tick' = tick - 1)
                    & (container5deploys3' = true);
endmodule



rewards "value"
    //resource contention amongst competing containers
    //deployed on the same server
    container1deploys2 = true & container3deploys2 = true : 0;
    container1deploys3 = true & container3deploys3 = true : 0;
    container1deploys4 = true & container3deploys4 = true : 0;

    //not deployed on the same server
    container1deploys2 != true & container3deploys2 != true : c1u + c2u + c3u + c4u
   + c5u;
    container1deploys3 != true & container3deploys3 != true : c1u + c2u + c3u + c4u
   + c5u;
    container1deploys4 != true & container3deploys4 != true : c1u + c2u + c3u + c4u
   + c5u;
endrewards
```

Appendix C

PRISM Models for Verification of Service Allocation Policies

This appendix details a PRISM implementation of the SMG model for service allocation policies. The counter utilized for the number of game rounds is not depicted explicitly for brevity.

```
//turn based game
global turn : [0..2] init 0;

//define reward constants : as many as latency discretized intervals
const double rwd1;
const double rwd2;
const double rwd3;
const double rwdmax;

module failures
counter : [1..m] init m;
```

```
//generate failures with number of servers active
[genfailures] turn = 0 ->
        p1 : (counter' = 1) & (turn' = 1)
        + p2 : (counter' = 2) & (turn' = 1)
        + p3 : (counter' = 3) & (turn' = 1)
        + p4 : (counter' = 4) & (turn' = 1)
        + p5 : (counter' = 5) & (turn' = 1)


        ...
        ...


        + pm : (counter' = m) & (turn' = 1);
endmodule


module servicerequests
users : [1..k] init k;


//generate stochastic service requests
[genrequests] turn = 1 ->
            ps1 : (users' = 1) & (turn' = 2)
        + ps2 : (users' = 2) & (turn' = 2)
        + ps3 : (users' = 3) & (turn' = 2)
        + ps4 : (users' = 4) & (turn' = 2)
        + ps5 : (users' = 5) & (turn' = 2)
        ...
        ...
        + psk : (users' = k) & (turn' = 2);
endmodule


module serverlatency
```

```
latency : [1..ltmax] init ltmax;


//generate latencies in accordance with number of active servers and service
   requests generated in the previous turns


//pc1lt1 ... pc1ltmax forms a probability distribution over latency generated
[c1] turn = 2 & counter = 1 &  users = 1 ->
        pc1lt1 : (latency’ = 0) & (turn’ = 0)
        + pc1lt2 : (latency’ = 1) & (turn’ = 0)
        + pc1lt3 : (latency’ = 2) & (turn’ = 0)
        + pc1lt4 : (latency’ = 3) & (turn’ = 0)
        ...
        ...
        + pc1ltmax : (latency’ = 4) & (turn’ = 0);

[c2] turn = 2 & counter = 1 &  users = 2 ->
        pc2lt1 : (latency’ = 0) & (turn’ = 0)
        + pc2lt2 : (latency’ = 1) & (turn’ = 0)
        + pc2lt3 : (latency’ = 2) & (turn’ = 0)
        + pc2lt4 : (latency’ = 3) & (turn’ = 0)
        ...
        ...
        + pc2ltmax : (latency’ = 4) & (turn’ = 0);


        ...
        ...
        ...

[cX] turn = 2 & counter = 6 &  users = 6 ->
        pcXlt1 : (latency’ = 0) & (turn’ = 0)
        + pcXlt2 : (latency' = 1) & (turn' = 0)
        + pcXlt3 : (latency' = 2) & (turn' = 0)
        + pcXlt4 : (latency' = 3) & (turn' = 0)
        ...
        ...
        + pcXltmax : (latency' = 4) & (turn' = 0);
endmodule


//assign each latency value a specific reward
rewards "users"
    latency = ltmax : rwdmax;
    ...
    ...
    latency = 3 : rwd3;
    latency = 2 : rwd2;
    latency = 1 : rwd1;
endrewards
```

References

[1] Pasika Ranaweera, Anca Delia Jurcut, and Madhusanka Liyanage. Survey on multi-access edge computing security and privacy. *IEEE Communications Surveys & Tutorials*, 23(2): 1078–1124, 2021.

[2] Stephen Pasteris, Shiqiang Wang, Mark Herbster, and Ting He. Service placement with provable guarantees in heterogeneous edge computing systems. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 514–522. IEEE, 2019.

[3] Konstantinos Poularakis, Jaime Llorca, Antonia M Tulino, Ian Taylor, and Leandros Tassiulas. Service placement and request routing in mec networks with storage, computation, and communication constraints. *IEEE/ACM Transactions on Networking*, 28(3): 1047–1060, 2020.

[4] Vajiheh Farhadi, Fidan Mehmeti, Ting He, Tom La Porta, Hana Khamfroush, Shiqiang Wang, and Kevin S Chan. Service placement and request scheduling for data-intensive applications in edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1279–1287. IEEE, 2019.

[5] Konstantinos Poularakis, Jaime Llorca, Antonia M Tulino, Ian Taylor, and Leandros Tassiulas. Joint service placement and request routing in multi-cell mobile edge computing networks. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 10–18. IEEE, 2019.

[6] Qinglan Peng, Yunni Xia, Zeng Feng, Jia Lee, Chunrong Wu, Xin Luo, Wanbo Zheng, Shanchen Pang, Hui Liu, Yidan Qin, et al. Mobility-aware and migration-enabled online edge user allocation in mobile edge computing. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 91–98. IEEE, 2019.

[7] Phu Lai, Qiang He, Guangming Cui, Xiaoyu Xia, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. Edge user allocation with dynamic quality of service. In *International Conference on Service-Oriented Computing*, pages 86–101. Springer, 2019.

[8] Phu Lai, Qiang He, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. Optimal edge user allocation in edge computing with variable sized vector bin packing. In *International Conference on Service-Oriented Computing*, pages 230–245. Springer, 2018.

[9] Shangguang Wang, Yan Guo, Ning Zhang, Peng Yang, Ao Zhou, and Xuemin Sherman Shen. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. *IEEE Transactions on Mobile Computing*, 20(3):939–951, 2021.

[10] Alexandros Evangelidis, David Parker, and Rami Bahsoon. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems*, 87:629–638, 2018.

[11] Athanasios Naskos, Emmanouela Stachtiari, Anastasios Gounaris, Panagiotis Katsaros, Dimitrios Tsoumakos, Ioannis Konstantinou, and Spyros Sioutas. Dependable horizontal scaling based on probabilistic model checking. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 31–40. IEEE, 2015.

[12] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 329–338. IEEE, 2019.

[13] Jie Xu, Lixing Chen, and Shaolei Ren. Online learning for offloading and autoscaling in energy harvesting mobile edge computing. *IEEE Transactions on Cognitive Communications and Networking*, 3(3):361–373, 2017.

[14] Hao He, Jiang Hu, and Dilma Da Silva. Enhancing datacenter resource management through temporal logic constraints. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 133–142. IEEE, 2017.

[15] Atakan Aral and Ivona Brandic. Dependency mining for service resilience at the edge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 228–242. IEEE, 2018.

[16] Janick Edinger, Dominik Schäfer, Christian Krupitzer, Vaskar Raychoudhury, and Christian Becker. Fault-avoidance strategies for context-aware schedulers in pervasive computing systems. In *2017 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 79–88. IEEE, 2017.

[17] Jitendcr Grover and Rama Murthy Garimella. Reliable and fault-tolerant iot-edge architecture. In *2018 IEEE SENSORS*, pages 1–4. IEEE, 2018.

[18] Michele Sevegnani, Milan Kabác, Muffy Calder, and Julie McCann. Modelling and verification of large-scale sensor network infrastructures. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 71–81. IEEE, 2018.

[19] Muffy Calder, Simon Dobson, Michael Fisher, and Julie McCann. Making sense of the world: Framing models for trustworthy sensor-driven systems. *Computers*, 7(4):62, 2018.

[20] Matt Webster, Michael Breza, Clare Dixon, Michael Fisher, and Julie McCann. Formal verification of synchronisation, gossip and environmental effects for wireless sensor networks. *Electronic Communications of the EASST*, 76, 2019.

[21] Mingsong Chen, Saijie Huang, Xin Fu, Xiao Liu, and Jifeng He. Statistical model checking-based evaluation and optimization for cloud workflow resource allocation. *IEEE Transactions on Cloud Computing*, 8(2):443–458, 2016.

[22] Muffy Calder and Michele Sevegnani. Stochastic model checking for predicting component failures and service availability. *IEEE Transactions on Dependable and Secure Computing*, 16(1):174–187, 2017.

[23] Muffy Calder, Alexandros Koliousis, Michele Sevegnani, and Joseph Sventek. Real-time verification of wireless home networks using bigraphs with sharing. *Science of Computer Programming*, 80:288–310, 2014.

[24] Zhijin Qin, Yuanwei Liu, Geoffrey Ye Li, and Julie McCann. Modelling and analysis of low-power wide-area networks. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2017.

[25] Matt Webster, Michael Breza, Clare Dixon, Michael Fisher, and Julie McCann. Exploring the effects of environmental conditions and design choices on iot systems using formal methods. *Journal of Computational Science*, 45:101183, 2020.

[26] Oana Andrei and Muffy Calder. Data-driven modelling and probabilistic analysis of interactive software usage. *Journal of Logical and Algebraic Methods in Programming*, 100:195–214, 2018.

[27] Oana Andrei, Muffy Calder, Matthew Chalmers, Alistair Morrison, and Mattias Rost. Probabilistic formal analysis of app usage to inform redesign. In *International Conference on Integrated Formal Methods*, pages 115–129. Springer, 2016.

[28] Mingsong Chen, Daian Yue, Xiaoke Qin, Xin Fu, and Prabhat Mishra. Variation-aware evaluation of mpsoc task allocation and scheduling strategies using statistical model checking. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 199–204. IEEE, 2015.

[29] Gethin Norman, David Parker, Marta Kwiatkowska, Sandeep Shukla, and Rajesh Gupta. Using probabilistic model checking for dynamic power management. *Formal aspects of computing*, 17(2):160–176, 2005.

[30] Dongdong An, Jing Liu, Min Zhang, Xiaohong Chen, Mingsong Chen, and Haiying Sun. Uncertainty modeling and runtime verification for autonomous vehicles driving control: A machine learning-based approach. *Journal of Systems and Software*, 167:110617, 2020.

[31] Matthias Fruth. Probabilistic model checking of contention resolution in the ieee 802.15. 4 low-rate wireless personal area network protocol. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 290–297. IEEE, 2006.

[32] Athanassios Boulis, Ansgar Fehnker, Matthias Fruth, and Annabelle McIver. Cavi–simulation and model checking for wireless sensor networks. In *2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 37–38. IEEE, 2008.

[33] Jordi Navarrette, Subash Shankar, Xiaojie Zhang, and Saptarshi Debroy. Formal modeling and analysis of multi-rogue backoff manipulation attacks in unlicensed networks. In *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, pages 1–7. IEEE, 2020.

[34] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. The MIT Press, 2018.

[35] Zhixin Pan, Jennifer Sheldon, and Prabhat Mishra. Test generation using reinforcement learning for delay-based side-channel analysis. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–7. IEEE, 2020.

[36] Zhixin Pan and Prabhat Mishra. Automated test generation for hardware trojan detection using reinforcement learning. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 408–413, 2021.

[37] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[38] Taolue Chen, Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, and Aistis Simaitis. Automatic verification of competitive stochastic systems. *Formal Methods Syst. Des.*, 43 (1):61–92, 2013.

[39] He Li, Kaoru Ota, and Mianxiong Dong. Eccn: Orchestration of edge-centric computing and content-centric networking in the 5g radio access network. *IEEE Wireless Communications*, 25(3):88–93, 2018.

[40] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE Network*, 32(1):96–101, 2018.

[41] Zhenyu Zhou, Bingchen Wang, Mianxiong Dong, and Kaoru Ota. Secure and efficient vehicle-to-grid energy trading in cyber physical systems: Integration of blockchain and edge computing. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 50(1): 43–57, 2019.

[42] He Li, Kaoru Ota, and Mianxiong Dong. Deep reinforcement scheduling for mobile crowd-sensing in fog computing. *ACM Trans. Internet Technol.*, 19(2), 2019. ISSN 1533-5399.

[43] Jianwen Xu, Kaoru Ota, and Mianxiong Dong. Big data on the fly: Uav-mounted mobile edge computing for disaster management. *IEEE Trans. Netw. Sci. Eng.*, 7(4):2620–2630, 2020. doi: 10.1109/TNSE.2020.3016569. URL `https://doi.org/10.1109/TNSE.2020.3016569`.

[44] Fuli Qiao, Mianxiong Dong, Kaoru Ota, Siyi Liao, Jun Wu, and Jianhua Li. Making big data intelligent storable at the edge: Storage resource intelligent orchestration. In *2019 IEEE Global Communications Conference, GLOBECOM 2019, Waikoloa, HI, USA, December 9-13, 2019*, pages 1–6. IEEE, 2019. doi: 10.1109/GLOBECOM38437.2019.9013942. URL `https://doi.org/10.1109/GLOBECOM38437.2019.9013942`.

[45] Chaofeng Zhang, Mianxiong Dong, and Kaoru Ota. Deploying SDN control in internet of uavs: Q-learning-based edge scheduling. *IEEE Trans. Netw. Serv. Manag.*, 18(1):526–537, 2021. doi: 10.1109/TNSM.2021.3059159. URL `https://doi.org/10.1109/TNSM.2021.3059159`.

[46] Xiaoyu Xia, Feifei Chen, Qiang He, John Grundy, Mohamed Abdelrazek, and Hai Jin. Online collaborative data caching in edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):281–294, 2020.

[47] Qiang He, Guangming Cui, Xuyun Zhang, Feifei Chen, Shuiguang Deng, Hai Jin, Yanhui Li, and Yun Yang. A game-theoretical approach for user allocation in edge computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 31(3):515–529, 2019.

[48] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, pages 3–18, 2019.

[49] Docker hub : https://hub.docker.com/.

[50] Stephen Pasteris, Shiqiang Wang, Mark Herbster, and Ting He. Service placement with provable guarantees in heterogeneous edge computing systems. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 514–522. IEEE, 2019.

[51] Tao Ouyang, Rui Li, Xu Chen, Zhi Zhou, and Xin Tang. Adaptive user-managed service placement for mobile edge computing: An online learning approach. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1468–1476. IEEE, 2019.

[52] Andrea Tomassilli, Frédéric Giroire, Nicolas Huin, and Stéphane Pérennes. Provably efficient algorithms for placement of service function chains with ordering constraints. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 774–782. IEEE, 2018.

[53] Bin Gao, Zhi Zhou, Fangming Liu, and Fei Xu. Winning at the starting line: Joint network selection and service placement for mobile edge computing. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1459–1467. IEEE, 2019.

[54] Lin Wang, Lei Jiao, Ting He, Jun Li, and Max Mühlhäuser. Service entity placement for social virtual reality applications in edge computing. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 468–476. IEEE, 2018.

[55] Jie Xu, Lixing Chen, and Pan Zhou. Joint service caching and task offloading for mobile edge computing in dense networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 207–215. IEEE, 2018.

[56] Nuo Yu, Qingyuan Xie, Qiuyun Wang, Hongwei Du, Hejiao Huang, and Xiaohua Jia. Collaborative service placement for mobile edge computing applications. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

[57] Lixing Chen, Cong Shen, Pan Zhou, and Jie Xu. Collaborative service placement for edge computing in dense small cell networks. *IEEE Transactions on Mobile Computing*, 20(2): 377–390, 2021.

[58] Bin Gao, Zhi Zhou, Fangming Liu, Fei Xu, and Bo Li. An online framework for joint network selection and service placement in mobile edge computing. *IEEE Transactions on Mobile Computing*, 2021.

[59] Changsheng You, Kaibin Huang, Hyukjin Chae, and Byoung-Hoon Kim. Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Transactions on Wireless Communications*, 16(3):1397–1411, 2016.

[60] Xu Chen, Lei Jiao, Wenzhong Li, and Xiaoming Fu. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, 24 (5):2795–2808, 2015.

[61] Haoxin Wang and Jiang Xie. User preference based energy-aware mobile ar system with edge computing. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1379–1388. IEEE, 2020.

[62] Shan Cao, Yang Wang, and Chengzhong Xu. Service migrations in the cloud for mobile accesses: A reinforcement learning approach. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–10. IEEE, 2017.

[63] Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin Chan, and Kin K Leung. Dynamic service migration in mobile edge computing based on markov decision process. *IEEE/ACM Transactions on Networking*, 27(3):1272–1288, 2019.

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

[65] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[66] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future generation computer systems*, 29(1):84–106, 2013.

[67] Hong Yao, Changmin Bai, Muzhou Xiong, Deze Zeng, and Zhangjie Fu. Heterogeneous cloudlet deployment and user-cloudlet association toward cost effective fog computing. *Concurrency and Computation: Practice and Experience*, 29(16):e3975, 2017.

[68] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things*, 3(5):637–646, 2016.

[69] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.

[70] Weiwei Chen, Dong Wang, and Keqin Li. Multi-user multi-task computation offloading in green mobile edge cloud computing. *IEEE Transactions on Services Computing*, 12(5): 726–738, 2019.

[71] Zhixiong Chen, Zhengchuan Chen, and Yunjian Jia. Integrated task caching, computation offloading and resource allocation for mobile edge computing. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

[72] Changfeng Ding, Jun-Bo Wang, Ming Cheng, Chuanwen Chang, Jin-Yuan Wang, and Min Lin. Joint beamforming and computation offloading for multi-user mobile-edge computing. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[73] Zhe Hao, Yanhua Sun, Qing Li, and Yanhua Zhang. Delay - energy efficient computation offloading and resources allocation in heterogeneous network. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

[74] José Leal D Neto, Se-Young Yu, Daniel F Macedo, José Marcos S Nogueira, Rami Langar, and Stefano Secci. Uloof: a user level online offloading framework for mobile edge computing. *IEEE Transactions on Mobile Computing*, 17(11):2660–2674, 2018.

[75] Feng Wang, Jie Xu, Xin Wang, and Shuguang Cui. Joint offloading and computing optimization in wireless powered mobile-edge computing systems. *IEEE Transactions on Wireless Communications*, 17(3):1784–1797, 2017.

[76] Junhui Zhao, Qiuping Li, Yi Gong, and Ke Zhang. Computation offloading and resource allocation for cloud assisted mobile edge computing in vehicular networks. *IEEE Transactions on Vehicular Technology*, 68(8):7944–7956, 2019.

[77] Gongming Zhao, Hongli Xu, Yangming Zhao, Chunming Qiao, and Liusheng Huang. Offloading tasks with dependency and service caching in mobile edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(11):2777–2792, 2021.

[78] Zhe Wang, Sanjay Ranka, and Prabhat Mishra. Temperature-aware task partitioning for real-time scheduling in embedded systems. In *2012 25th International Conference on VLSI Design*, pages 161–166. IEEE, 2012.

[79] Weixun Wang, Xiaoke Qin, and Prabhat Mishra. Temperature-and energy-constrained scheduling in multitasking systems: A model checking approach. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 85–90. IEEE, 2010.

[80] Weixun Wang and Prabhat Mishra. System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(5):902–910, 2011.

[81] Rajesh Devaraj, Arnab Sarkar, and Santosh Biswas. Fault-tolerant preemptive aperiodic rt scheduling by supervisory control of tdes on multiprocessors. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):1–25, 2017.

[82] Sanjay Moulik, Rajesh Devaraj, and Arnab Sarkar. Cost: A cluster-oriented scheduling technique for heterogeneous multi-cores. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1951–1957. IEEE, 2018.

[83] Jianan Sun, Qing Gu, Tao Zheng, Ping Dong, Alvin Valera, and Yajuan Qin. Joint optimization of computation offloading and task scheduling in vehicular edge computing networks. *IEEE Access*, 8:10466–10477, 2020.

[84] Lingfang Gao and Melody Moh. Joint computation offloading and prioritized scheduling in mobile edge computing. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 1000–1007. IEEE, 2018.

[85] Tongxin Zhu, Tuo Shi, Jianzhong Li, Zhipeng Cai, and Xun Zhou. Task scheduling in deadline-aware mobile edge computing systems. *IEEE Internet of Things Journal*, 6(3): 4854–4866, 2018.

[86] Zhufang Kuang, Linfeng Li, Jie Gao, Lian Zhao, and Anfeng Liu. Partial offloading scheduling and power allocation for mobile edge computing systems. *IEEE Internet of Things Journal*, 6(4):6774–6785, 2019.

[87] Xiaoke Qin, Weixun Wang, and Prabhat Mishra. Tcec: Temperature and energy-constrained scheduling in real-time multitasking systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(8):1159–1168, 2012.

[88] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Leveraging deep learning to improve performance predictability in cloud microservices with seer. *ACM SIGOPS Operating Systems Review*, 53(1):34–39, 2019.

[89] Alper Kamil Bozkurt, Yu Wang, Michael M Zavlanos, and Miroslav Pajic. Control synthesis from linear temporal logic specifications using model-free reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10349–10355. IEEE, 2020.

[90] Mbarka Soualhia, Chunyan Fu, and Foutse Khomh. Infrastructure fault detection and prediction in edge cloud environments. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, page 222–235, 2019.

[91] Hussaini Adamu, Bashir Mohammed, Ali Bukar Maina, Andrea Cullen, Hassan Ugail, and Irfan Awan. An approach to failure prediction in a cloud based environment. In *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 191–197. IEEE, 2017.

[92] Min Huang, Zhen Liu, and Yang Tao. Mechanical fault diagnosis and prediction in iot based on multi-source sensing data fusion. *Simulation Modelling Practice and Theory*, 102:101981, 2020.

[93] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

[94] Costas Courcoubetis and Mihalis Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *29th Annual Symposium on Foundations of Computer Science*, pages 338–345, 1988.

[95] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *Journal of the ACM (JACM)*, 42(4):857–907, 1995.

[96] Zibin Zheng, Yilei Zhang, and Michael R Lyu. Investigating qos of real-world web services. *IEEE Transactions on Services Computing*, 7(1):32–39, 2012.

[97] Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin Chan, and Kin K Leung. Dynamic service migration in mobile edge computing based on markov decision process. *IEEE/ACM Transactions on Networking*, 27(3):1272–1288, 2019.

[98] Existing commercial wireless telecommunication services facilities in san francisco: Datasf: City and county of san francisco. https://data.sfgov.org/Geographic-Locations-and-Boundaries/Existing-Commercial-Wireless-Telecommunication-Ser/aa26-h926, May 2020.

[99] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 19–33, 2019.

[100] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

[101] Repository of availability traces. https://pbg.cs.illinois.edu/availability/.

[102] Martin Pielot. Crawdad dataset telefonica/mobilephoneuse (v. 2019-04-29), April 2019.

[103] Jiri Matousek and Bernd Gärtner. *Understanding and using linear programming*. Springer Science & Business Media, 2007.

[104] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169, 2000.

[105] Zoltán Füredi. Matchings and covers in hypergraphs. *Graphs and Combinatorics*, 4(1): 115–206, 1988. URL `https://doi.org/10.1007/BF01864160`.

[106] Petr Slavık. A tight analysis of the greedy algorithm for set cover. *Journal of Algorithms*, 25(2):237–254, 1997.

[107] Shuiguang Deng, Hongyue Wu, Daning Hu, and J Leon Zhao. Service selection for composition with qos correlations. *IEEE Transactions on Services Computing*, 9(2):291–303, 2014.

[108] http://www.gurobi.com. Gurobi. 2020.

[109] Tayebeh Bahreini and Daniel Grosu. Efficient placement of multi-component applications in edge computing systems. In *Proceedings of the Second ACM/IEEE SEC*, pages 1–11, 2017.

[110] Shiqiang Wang, Murtaza Zafer, and Kin K Leung. Online placement of multi-component applications in edge computing environments. *IEEE Access*, 5:2514–2533, 2017.

[111] Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. Split and migrate: Resource-driven placement and discovery of microservices at the edge. In *OPODIS*, volume 153, pages 9:1–9:16, 2019.

[112] Hossein Badri, Tayebeh Bahreini, Daniel Grosu, and Kai Yang. Energy-aware application placement in mobile edge computing: A stochastic optimization approach. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):909–922, 2020.

[113] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *ICLR*, 2019.

[114] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[115] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[116] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Cautious reinforcement learning with logical constraints. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS*, pages 483–491, 2020.

[117] Dorsa Sadigh, Eric S Kim, Samuel Coogan, S Shankar Sastry, and Sanjit A Seshia. A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications. In *53rd IEEE Conference on Decision and Control*, pages 1091–1096. IEEE, 2014.

[118] Martin Breitbach, Dominik Schäfer, Janick Edinger, and Christian Becker. Context-aware data and task placement in edge computing environments. In *PerCom*, pages 1–10. IEEE, 2019.

[119] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[120] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. Metis: Learning to schedule long-running applications in shared container clusters at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, 2020.

[121] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[122] 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.

[123] Li Liu, Sammy Chan, Guangjie Han, Mohsen Guizani, and Masaki Bandai. Performance modeling of representative load sharing schemes for clustered servers in multiaccess edge computing. *IEEE Internet of Things Journal*, 6(3):4880–4888, 2018.

[124] Hailiang Zhao, Shuiguang Deng, Cheng Zhang, Wei Du, Qiang He, and Jianwei Yin. A mobility-aware cross-edge computation offloading framework for partitionable applications. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 193–200. IEEE, 2019.

[125] Rui Zhu, Bang Liu, Di Niu, Zongpeng Li, and Hong Vicky Zhao. Network latency estimation for personal devices: A matrix completion approach. *IEEE/ACM Transactions on Networking*, 25(2):724–737, 2016.

[126] D Bertsekas and J Tsitsiklis. Introduction to probability, ser. *Athena Scientific optimization and computation series. Athena Scientific*, 2008.