Energy and Time Aware Scheduling of Applications in Edge and Fog Computing Environments

Mohammad Goudarzi

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

February 2022

ORCID: 0000-0002-7178-3386

Energy and Time Aware Scheduling of Applications in Edge and Fog Computing Environments

Mohammad Goudarzi
Principal Supervisor: Prof. Rajkumar Buyya
Co-Supervisor: Prof. Marimuthu Palaniswami

Abstract

The Internet of Things (IoT) paradigm is playing a principal role in the advancement of many application scenarios such as healthcare, smart city, transportation, entertainment, and agriculture, which significantly affect the daily life of humans. The smooth execution of these applications requires sufficient computing and storing resources to support the massive amount of data generated by IoT devices. However, IoT devices are resource-limited intrinsically and are not capable of efficient processing and storage of large volumes of data. Hence, IoT devices require surrogate available resources for the smooth execution of their heterogeneous applications, which can be either computation-intensive or latency-sensitive. Cloud datacenters are among the potential resource providers for IoT devices. However, as they reside at a multi-hop distance from IoT devices, they cannot efficiently execute IoT applications, especially latency-sensitive ones. *Fog computing* paradigm, which extends Cloud services to the edge of the network within the proximity of IoT devices, offers low latency execution of IoT applications. Hence, it can improve the response time of IoT applications, service startup time, and network congestion. Also, it can reduce the energy consumption of IoT devices by minimizing their active time. However, Fog servers are resource-limited compared to Cloud servers, preventing them from the execution of all types of IoT applications, especially extremely computation-intensive applications. Hence, Cloud servers are used to support Fog servers to create a robust computing environment with heterogeneous types of resources. Consequently, the Fog computing paradigm is highly dynamic, distributed, and heterogeneous. Thus, without efficient scheduling techniques for the management of IoT applications, it is difficult to harness the full potential of this computing paradigm for different IoT-driven application scenarios.

This thesis focuses on different scheduling techniques for the management of IoT

applications in Fog computing environments while considering *a.* IoT devices' characteristics, *b.* the structure of IoT applications, *c.* the context of resource providers, *d.* the networking characteristics of the Fog servers, *e.* the execution cost of running IoT applications, and *f.* the dynamics of computing environment. This thesis advances the state-of-the-art by making the following contributions:

1. A comprehensive taxonomy and literature review on the scheduling of IoT applications from different perspectives, namely application structure, environmental architecture, optimization properties, decision engine characteristics, and performance evaluation, in Fog computing environments.

2. A distributed Fog-driven scheduling technique for network resource allocation in dense and ultra-dense Fog computing environments to optimize throughput and satisfy users' heterogeneous demands.

3. A distributed scheduling technique for the batch placement of concurrent IoT applications to optimize the execution time of IoT applications and energy consumption of IoT devices.

4. A distributed application placement and migration management technique to optimize the execution time of IoT applications, the energy consumption of IoT devices, and the migration downtime in hierarchical Fog computing environments.

5. A Distributed Deep Reinforcement Learning (DDRL) technique for scheduling IoT applications in highly dynamic Fog computing environments to optimize the execution time of IoT applications and energy consumption of IoT devices.

6. A system software for scheduling IoT applications in multi-Cloud Fog computing environments.

7. A detailed study outlining challenges and new research directions for the scheduling of IoT applications in Fog computing environments.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Mohammad Goudarzi, February 2022

Preface

Main Contributions

This thesis research has been carried out in Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2-8 and are based on the following publications:

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "Scheduling IoT Applications in Edge and Fog Computing Environments: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, USA, 2022 (Revision).

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Fog-driven Dynamic Resource Allocation Technique in Ultra Dense Femtocell Networks", *Journal of Network and Computer Applications (JNCA)*, Volume 145, ISSN: 1084-8045, Elsevier Press, Amsterdam, Netherlands, November 2019.

- **Mohammad Goudarzi**, Huaming Wu, Marimuthu Palaniswami, and Rajkumar Buyya, "An Application Placement Technique for Concurrent IoT Applications in Edge and Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC)*, Volume 20, Number 4, Pages: 1298-1311, ISSN: 1536-1233, IEEE Press, New York, USA, January 2020.

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Distributed Application Placement and Migration Management Techniques for Edge and Fog Computing Environments", *Proceedings of the 16th Conference on Computer*

Science and Intelligent Systems (FedCSIS), IEEE Press, Pages: 37-56, Online, Poland, September 2-5, 2021.

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Distributed Deep Reinforcement Learning Technique for Application Placement in Edge and Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC)*, (in press, DOI: 10.1109/TMC.2021.3123165, accepted on 23 October 2021).

- **Mohammad Goudarzi**, Qifan Deng, and Rajkumar Buyya, "Resource Management in Edge and Fog Computing using FogBus2 Framework", *Managing Internet of Things Applications across Edge and Cloud Data Centres*, Rajiv Ranjan, Karan Mitra, Prem Prakash Jayaraman, Albert Y. Zomaya (eds), ISBN: 978-1785617799, IET Press, Hertfordshire, UK, June 2022.

Supplementary Contributions

During the Ph.D. candidature, I have also contributed to the following collaborative works (this thesis does not claim them as its contributions):

- **Mohammad Goudarzi**, Shashikant Ilager, and Rajkumar Buyya, "Cloud Computing and Internet of Things: Recent Trends and Directions", *New Frontiers in Cloud Computing and Internet of Things*, Rajkumar Buyya, Lalit Garg, Giancarlo Fortino, Sanjay Misra (eds), 2022 (Accepted, in press).

- Qifan Deng, **Mohammad Goudarzi**, and Rajkumar Buyya, "FogBus2: A Lightweight and Distributed Container-based Framework for Integration of IoT-enabled Systems with Edge and Cloud Computing", *Proceedings of the Big Data in Emergent Distributed Environments (BiDEDE'21) in conjunction with the 2021 ACM SIGMOD-/PODS Conference*, ACM Press, Pages: 1–8, Shaanxi, China, June 20-25, 2021.

- Redowan Mahmud, Samodha Pallewatta, **Mohammad Goudarzi**, and Rajkumar Buyya, "IFogSim2: An Extended iFogSim Simulator for Mobility, Clustering, and Microservice Management in Edge and Fog Computing Environments", *Journal of Systems and Software (JSS)* (Accepted, in press).

Acknowledgements

Ph.D. is a rewarding journey full of wonderful experiences that is not possible without the support and encouragement of many people. Now, since my journey is near the end, I would like to take the opportunity to express my sincere thanks to all the amazing people who helped me along the way.

First and foremost, I offer my deepest gratitude to my principal supervisor, Professor Rajkumar Buyya, who awarded me the opportunity to pursue my studies under his guidance. I would like to thank him for his continuous support, encouragement, hard work, constructive comments, advice, and guidance throughout all rough and enjoyable moments of my Ph.D. endeavor. I also thank my co-supervisor, Professor Marimuthu Palaniswami, for his insightful comments, support, and motivation during my Ph.D. career. I would also like to express my gratitude to the chair of my Ph.D. advisory committee, Professor James Bailey, for his support, constructive comments, and suggestions on my work.

I would also like to thank all the past and current members of the CLOUDS Laboratory, at the University of Melbourne. In particular, I thank Dr. Adel Nadjaran Toosi, Dr. Amir Vahid Dastjerdi, Prof. Vlado Stankovski, Dr. Huaming Wu, Prof. Mohsen Kahani, Dr. Maria Rodriguez, Dr. Rami Bahsoon, Dr. Sukhpal Singh Gill, Dr. Xunyun Liu, Dr. Jungmin Jay Son, Dr. Safiollah Heidari, Dr. Minxian Xu, Dr. Sara Kardani, Dr. Shashikant Ilager, Dr. Farzad Khodadadi, Dr. Caesar Wu, Dr. Muhammad Hilman, Dr. Redowan Mahmud, Dr. Muhammed Tawfiqul Islam, Dr. TianZhang He, Zhiheng Zhong, Samodha Pallewatta, Amanda Jayanetti, Rajeev Muralidhar, Kwangsuk Song, Anupama Mampage, Jie Zhao, Ming Chen, Siddharth Agarwal, Tharindu Bandara, Thanh-Hoa Nguyen, Qifan Deng, Zhiyu Wang, Kalyani Pendyala, Linna Ruan, Shreshth Tuli, Riccardo Mancini, Dr. Amin Shahraki, and Dr. Mohammad Reza Razian for their support.

I also thank the School of Computing and Information System's admin staff, especially Rhonda Smith, who continuously supported and responded to many queries throughout my Ph.D. candidature.

I wish to acknowledge the Australian Federal Government and its funding agencies, the University of Melbourne, Australian Research Council (ARC), Oracle, and CLOUDS laboratory for granting scholarships and supports that enabled me to do the research.

I would like to give heartfelt thanks to my parents and my parents-in-law for their endless help, support, and love.

Lastly and most importantly, I am thankful to my wife for her endless love, devotion, appreciation, and understanding.

Mohammad Goudarzi
Melbourne, Australia
February 2022

Contents

List of Figures

List of Tables

Chapter 1
Introduction

The Internet of Things (IoT) has become an integral basis of the digital world, thanks to the continuous development of super-cheap and tiny-sized computer chips and ubiquitous access to the Internet. In the context of IoT, the term "Things" refers to any entity (e.g., smart devices, sensors, human beings) that are context-aware and able to communicate with other entities without any temporal and spatial constraints [2]. Small devices and sensors act as distributed data aggregators with Internet access that forward collected data to the larger computer platforms for processing and/or permanent storage [3]. Thus, it has shaped a new interaction type among different real-world entities. This distributed paradigm draws a promising future and provides a great opportunity for developers and businesses to transform their work into a smarter version.

IoT applications span across almost all vital aspects of modern living, such as healthcare, security, entertainment, transportation, and industrial systems [2, 4], as depicted in Fig. 1.1. According to the Cisco [5], Norton [6], and Business Insider [7], 15 billion IoT devices will be connected to the Internet by 2023, 21 billion by 2025, and 41 billion by 2027. Bain & Company [8] calculated the size of the IoT market in 2021 (including hardware, software, systems integration, and data services) to be around 520 billion U.S dollars, while Statista [9] and Business Insider [7] expect the size of IoT market will reach to 1 trillion U.S dollars by the end of 2022 and over 2 trillion U.S dollars by 2027, accordingly.

Considering the ever-increasing number of IoT devices and IoT applications, a tremendous amount of data is being generated. IoT devices may produce data either constantly or periodically. Statistics depict that 18.3 ZB of data was produced by IoT devices in 2019 while International Data Corporation (IDC) [10] predicts about a 400% increase in upcoming years, which hits 73 ZB of data by 2025. The real power of IoT resides in col-

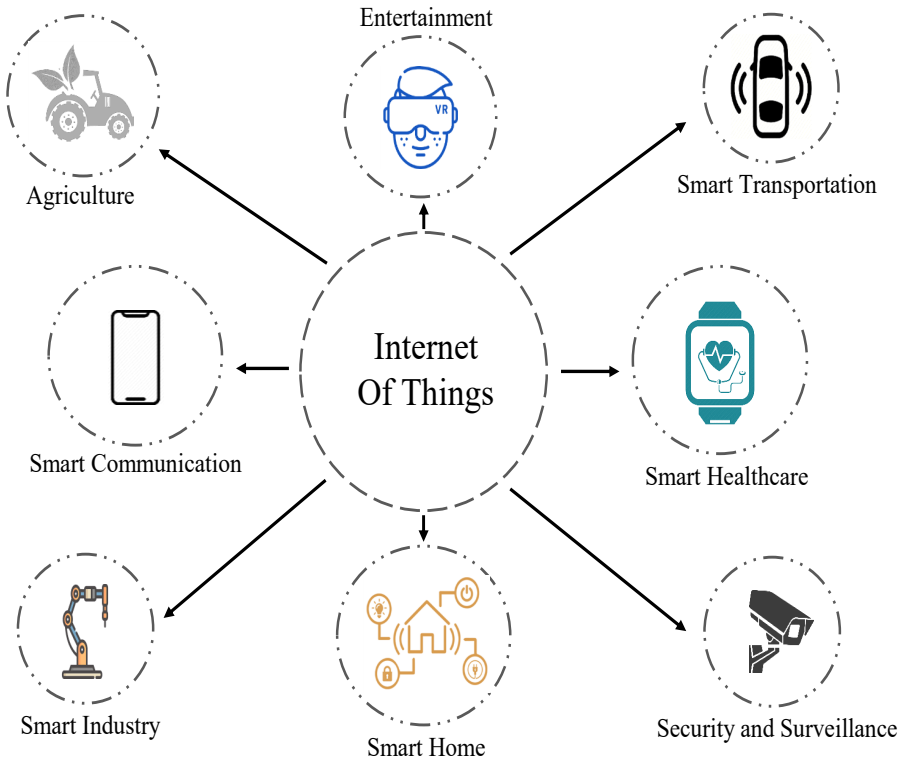


Figure 1.1: IoT applications

lecting and analyzing the data circulating in the environment [2]. However, the majority of the IoT devices are equipped with a constrained battery, computing units, and storage capacity, which prevent the efficient execution of IoT applications and data analysis in a timely manner. Hence, data should be forwarded to surrogate servers for processing and storage. The processing, storage, and transmission of this gigantic amount of IoT data require special attention when considering different IoT applications.

1.1 Edge and Fog Computing Paradigms

The development in computing and networking technologies over the last many decades is attributed to underlying technologies of Cloud computing and IoT. Cloud computing is one of the main enablers of IoT that offers on-demand services to process, analyze, and store the data generated from IoT devices in a simplified, scalable, and affordable manner [2, 11]. Recent advances in Cloud computing services such as serverless plat-

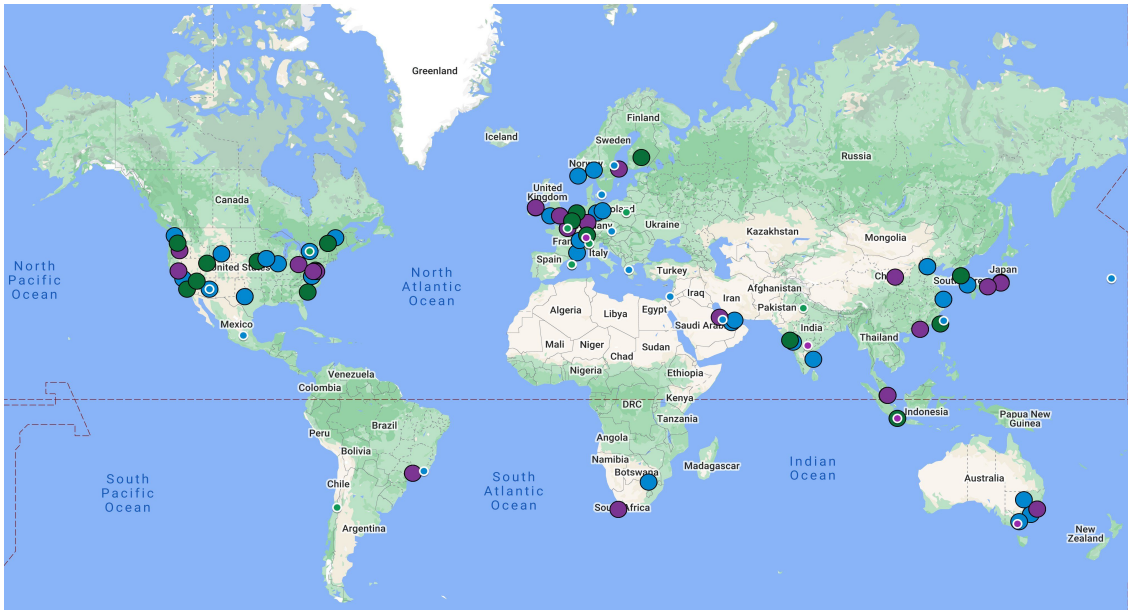


Figure 1.2: Geographical coverage of different Cloud service providers (Microsoft Azure: Blue, Amazon Web Services: Purple, Google Cloud Platform: Green)

forms, FaaS, transactional databases, Machine Learning (ML), and autonomous data warehouses open new horizons in the field of data acquisition and analysis of IoT data [12]. Besides, IoT businesses and companies that deploy their applications and systems on the Cloud can reduce their expenses (e.g., infrastructural and operational costs), which leads to more affordable services and products for the end-users.

Cloud datacenters are located at a multi-hop distance from IoT devices, as shown in Fig. 1.2. Thus, IoT applications running on these datacenters face an extended delay, which is required to transfer IoT data and instructions between IoT devices and Cloud datacenters. It can negatively affect the quality of service delivery in several ways. First, the service startup time may increase due to the expanded transmission time for sending instructions to distant Cloud instances. Besides, the communication latency to distant Cloud datacenters is high, which is an important barrier for the efficient service delivery of real-time and latency-sensitive IoT applications. Furthermore, the extended period of transmission time and higher latency lead to higher energy consumption for IoT devices. Moreover, when a huge number of IoT devices initiate data-driven interactions with applications deployed on the remote datacenters, it incurs substantial loads on the network

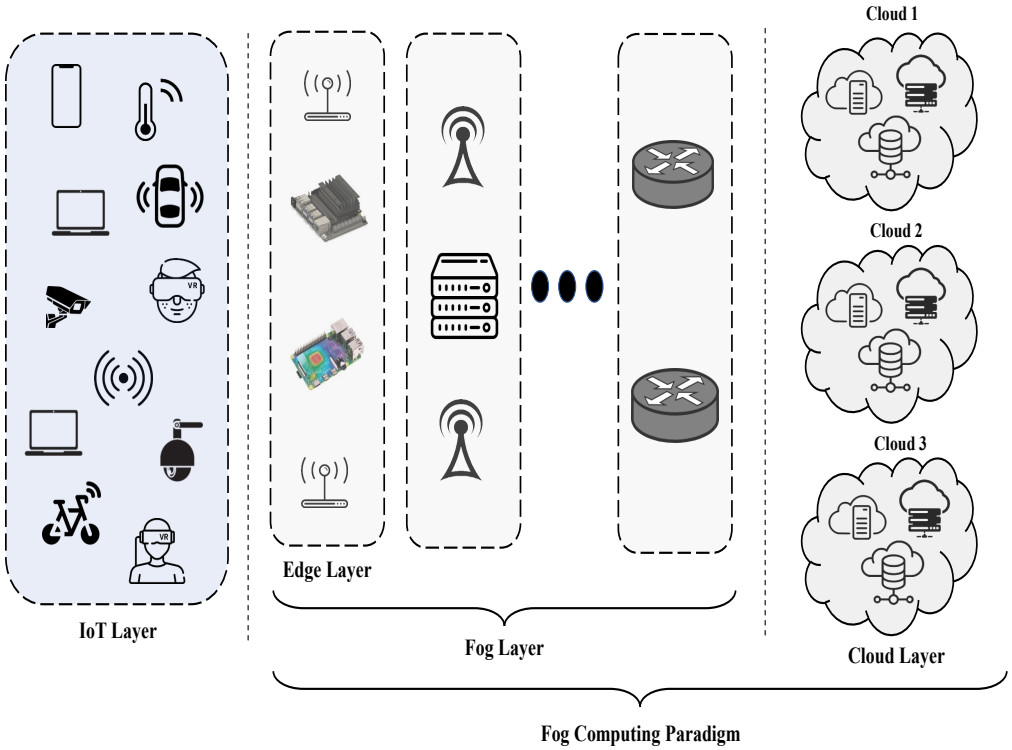


Figure 1.3: An illustration of Fog computing environments

and may lead to severe congestion. Last but not the least, it increases the computational overhead on Cloud datacenters and may reduce computing efficiency [13]. Therefore, the Cloud-centric execution of IoT applications may fail to satisfy the diverse range of IoT applications' requirements, especially for real-time and latency-sensitive IoT applications. To address these limitations, Edge and Fog computing paradigms have been introduced to bring Cloud-like services at the edge of the network and closer to IoT devices.

1.1.1 Properties of Edge and Fog Computing

Fog servers are geographically distributed in an intermediate layer between Cloud servers and IoT devices. The platform of Fog computing encompasses a large number of heterogeneous distributed Fog servers (e.g., Raspberry pi, Nvidia Jetson platform, small-cell base stations, nano servers, femtocells, regional servers, core routers, or switches) which

offers computational and storage resources for IoT devices running various applications, as depicted in Fig. 1.3. Since Fog servers are located in the proximity of IoT devices compared to Cloud datacenters, they can offer Cloud-like services with better latency, which effectively address the requirements of real-time and latency-sensitive IoT applications [14]. Moreover, these resources can be accessed with higher bandwidth (i.e., data rate) which reduces the required transmission time. Furthermore, Fog computing can help reduce the energy consumption of IoT devices, which is an important metric, especially for battery-constrained IoT devices. Also, it conserves network bandwidth that reduces the scope of network congestion [15]. Besides, Fog computing can help to better distribute the computational load, which reduces the massive load on Cloud datacenters. Additionally, Fog computing supports robust location-awareness and connectivity features to simplify the communication with mobile and energy-constrained IoT devices [16].

Compared to Cloud computing, Fog servers usually have limited resources (e.g., CPU, RAM) while these resources can be accessed more efficiently. Hence, Edge/Fog computing does not compete with Cloud computing, but they complement each other to satisfy the diverse requirements of heterogeneous IoT applications and systems. In our view, Edge computing harnesses only distributed Edge resources at the closest layer to IoT devices, and Fog computing harnesses distributed resources located in different computing layers while it also uses Cloud resources (although some works use these terms interchangeably), as shown in Fig. 1.3.

1.1.2 Initiatives for Realizing Fog Computing

Taking the benefits of Fog computing into cognizance, technology giants such as Amazon, Microsoft, Alphabet, and Oracle have already commenced integrating Fog services with their Cloud infrastructure [17, 18]. Moreover, Industrial Internet Consortium has been formed to standardize the theory of Fog computing [19]. Besides, several hardware manufacturers such as Cisco, Intel, Dell, and Nvidia are building compatible machines for Fog Computing [20–22]. There are several other software systems, frameworks, and IoT applications developed by SONM, NEC Laboratories, FogHorn systems, and

Drofika Labs. Considering these fast-paced advancements, Fog computing is expected to add 5.7 billion U.S dollars to the global market of utility computing by 2025 [23].

1.1.3 Challenges of Fog Computing

There are several challenges for the scheduling of IoT applications in Edge and Fog computing environments, which are discussed below.

• *Scheduling of networking resources*: According to the Ericsson Mobility Report [24] by 2023, the number of 5G-only mobile devices will reach more than 1 billion active devices, generating 20 percent of mobile data traffic. Such abundant data traffic will cause congestion, leading to decreased total network throughput. Hence, the resources such as radio resource blocks should be efficiently managed to support latency-sensitive applications.

Alongside an abundant amount of data that decreases the throughput, interference among Edge/Fog servers can further degrade the total throughput of the network. Although the Fog computing paradigm brings the computation and storage capabilities closer to the end-users, accessing those servers still requires the network resources that should be managed carefully to reach the highest performance of Edge/Fog computing. Besides, the number of interfered network resources should be minimized to guarantee a high data rate.

• *Distributed and heterogeneous Edge/Fog servers*: There are different types of Edge and Fog servers, ranging from on-premises servers (e.g., Raspberry Pi) to small-scale datacenters. These servers are heterogeneous in terms of their architecture, communication standards, supported operating systems, price-performance, etc. Moreover, compared to Cloud servers, the majority of these servers have resource limitations in terms of processing capability, networking characteristics, and storage. Hence, efficient scheduling of IoT applications in these computing environments is an important challenge to achieve the best-targeted performance.

• *Heterogeneous IoT applications*: IoT applications can be modeled as a set of tasks (either dependent or independent) with heterogeneous resource requirements. Scheduling these IoT applications on heterogeneous computing environments to optimize the

targeted optimization problem is an important challenge.

Moreover, there are different optimization metrics/parameters for different IoT applications. Two of the most important optimization metrics for IoT applications are their execution time and the energy consumption of IoT devices. Although several works consider these parameters separately, jointly optimizing these metrics in highly heterogeneous Edge and Fog computing environments is still in its infancy.

• *Batch scheduling*: Due to the ever-increasing number of IoT devices, the number of requests to obtain computational and storage resources has been significantly increased. Hence, the scheduler (i.e., decision engine) can receive several concurrent requests in its scheduling time slot with a higher probability. Although there are several policies to prioritize concurrently received IoT requests, prioritization is not always the solution, especially in scenarios where the number of concurrent IoT requests is large. So, the batch scheduling/placement of concurrent IoT applications is an important problem affecting the execution cost of all IoT applications.

• *Uncertain failures*: Although Fog computing reduces application service delivery time, Fog nodes are highly prone to get affected by anomalies, power failures, and out-of-capacity faults. It obstructs the proper execution of applications assigned to them. Hence, providing failure recovery mechanisms and policies for resource scheduling is a key factor for the smooth execution of different IoT applications

• *Cooperative execution of IoT applications*: Due to limited resources of Edge/Fog servers, one IoT application may not be able to be completely and efficiently executed on one server. Therefore, the resources of different Edge/Fog servers should be augmented for the efficient execution of IoT applications. However, heterogeneity of servers' architectures and resources put some constraints on such collaborative execution of IoT applications.

• *Mobility and migration management of IoT applications*: Mobile devices are an important part of IoT devices that run a wide range of IoT applications such as patient monitoring, entertainment, and transportation, just to mention a few. Considering user mobility, some modules of each IoT application may require migration to other servers for execution, leading to service interruption and extra execution costs.

• *Security*: The outcomes of Fog-based applications can be used by different parties

simultaneously. For example, the services of a Fog-based healthcare application are relevant to hospitals, insurance companies, and employer organizations. In such cases, Fog environments require to ensure on-demand and secure access to application outcomes as a part of application management. However, due to the resource scarcity and dynamics of Fog environments, it is hard to apply compute-intensive and complex security measures on Fog nodes.

In this thesis, we address the challenges of executing applications through resource-constrained, heterogeneous, and distributed Edge/Fog servers by identifying their suitable placement options in Edge/Fog computing environments. Here, we propose a taxonomy on energy and time-aware scheduling of IoT applications in Edge and Fog computing environments and review the existing scheduling strategies and their limitations. Furthermore, we develop a resource allocation policy for efficient allocation of networking resources in Edge and Fog computing environments. Afterward, a set of scheduling policies for proper and efficient management of IoT applications in Edge and Fog computing environments are proposed.

1.2 Methodologies

We follow the systematic research methodology as shown in Fig. 1.4 in our research works.

- Qualitative Comparison: For each research problem, we identify the key parameters of the problem, study current techniques in the literature, and compare them with our proposed technique.

- Modeling: For each research problem: 1) We provide the system model to represent key architectural elements involved in our system, 2) We define the targeted workflow and workloads, 3) We formulate the problem by focusing on specific optimization objectives of interests, including throughput, time, energy, and weighted cost.

- Algorithms: We propose different scheduling algorithms for the management of IoT applications in Edge and Fog computing environments. These algorithms are

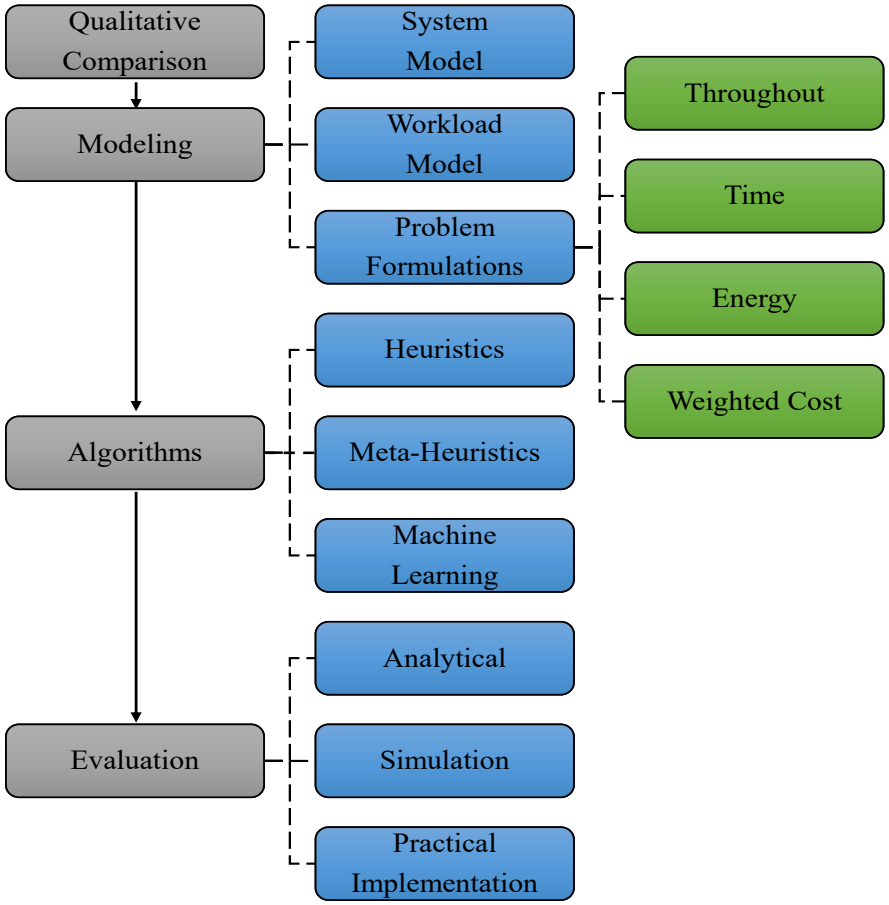


Figure 1.4: The research methodologies used in this research

designed based on heuristics, meta-heuristics, and machine learning-based approaches to solve optimization problems.

- Evaluation: The proposed techniques in this thesis have been evaluated using three methodologies, namely analytical, discrete event-driven simulation, and practical implementation. Due to limited accessibility and management costs, simulation is a common evaluation methodology to evaluate the proposed algorithms in complex and large-scale systems. In this thesis, we used MATLAB as an analytical tool, and OpenAI Gym [25] and iFogSim simulation toolkit [26, 27] for simulation. Also, we extend the iFogSim simulation toolkit with several new features. Besides, we build our prototype systems and evaluate respective metrics in real small-scale Edge/Fog computing environments.

Our research methodology has produced innovative algorithms, methods, open-sourced datasets, and software systems.

1.3 Research Questions and Objectives

In smart systems, an ever-increasing number of IoT devices closely interact with Fog and Cloud servers over the network for the smooth execution of IoT applications. The two most important costs related to the execution of applications in IoT-enabled systems are the execution time of IoT applications and the energy consumption of IoT devices. Hence, to satisfy the resource requirements of different IoT applications and optimize their execution cost in heterogeneous computing environments, scheduling policies play a key role. This thesis investigates the scheduling of IoT applications from the perspectives of different entities interacting with the IoT-enabled smart systems. The objective of this thesis is to improve the users' Quality of Experience (QoE) and the execution cost of running IoT applications in Edge and Fog Computing environments. To achieve these objectives, we solve important resource management problems by addressing the following research questions:

- Q1. *How to efficiently schedule network resources to satisfy different requirements of heterogeneous IoT applications in Fog computing environments?* Rapid increase of data-streaming IoT applications such as video streaming leads to a significant amount of data to be transferred over cellular networks, as one of the main communication mediums for IoT-enabled systems [2, 28, 29]. Besides, to address the requirements of a large number of IoT devices, an increasing number of Edge devices with computing and cellular communication modules are added to the network. Since the number of cellular network resources is restricted, the requested Quality of Service (QoS) of IoT applications can be satisfied for only a limited number of users. Moreover, the network resources of these devices can interfere together, which significantly reduces the throughput of the network. Thus, IoT devices cannot efficiently communicate with Fog servers for the smooth execution of IoT applications. Hence, it is necessary to design network resource allocation techniques for densely-deployed Fog computing environments to improve the throughput of the

network and interference on the network resources of Edge servers.

- Q2. *How to perform batch scheduling for concurrent IoT applications consisting of dependent tasks?* IoT applications, consisting of dependent tasks, can be modeled as Directed Acyclic Graph (DAG). To schedule each DAG-based IoT application, the constraints among dependent tasks should be satisfied before the execution of each task. Thus, for the batch scheduling of several concurrent DAG-based IoT applications, such dependency constraints for all applications should be considered together, which makes the scheduling problem of concurrent IoT applications more complex. Hence, it is necessary to design batch placement techniques for concurrent IoT applications to solve the dependency constraints of each application while trying to optimize the execution cost of a batch of DAG-based IoT applications.

- Q3. *How to perform placement and mobility management for real-time IoT applications?* The execution of real-time IoT applications exclusively on one Edge/Fog server may not be always feasible because of the limited resources of these servers. Hence, the resources of different servers should be aggregated together to satisfy the requirements of each IoT application. To obtain this, different Edge/Fog servers should be able to communicate and coordinate together for the execution of IoT applications. Moreover, IoT applications for mobile users are among the most popular type of IoT applications. As a result, as the user moves along the path, different modules/tasks of IoT the application currently placed on different servers should be migrated to servers in the proximity of users. It usually leads to service interruptions for a specific amount of time, which is a big problem, specifically for real-time IoT applications. Hence, it is necessary to design resource and mobility management policies to provide real-time service for IoT users while minimizing the service interruption time in the migration process.

- Q4. *How to automatically learn the scheduling policies in highly dynamic and stochastic computing environments to optimize complex objectives?* Numerous parameters are affecting the execution of IoT applications in Edge and Fog computing environments with non-linear relationships. These parameters include the Fog environment and IoT application characteristics. The Fog environment characteristics

include the number of servers, specification of each server, and networking specifications. The IoT application characteristics include the dependency model of IoT application, the requirement of each task, and data-flow among different tasks. These contributing parameters can be further extended based on different IoT scenarios. The majority of existing scheduling algorithms in Edge and Fog computing environments are based on static rules or manually fine-tuned heuristics that fail to capture these intricacies in the environment. Therefore, it is essential to build an adaptive scheduling algorithm based on Deep Reinforcement Learning (DRL) to deal with such complexity and learn adaptive scheduling policies.

1.4 Thesis Contributions

This thesis makes the following contributions to address the research problems mentioned above:

1. Proposes different taxonomies on the scheduling of IoT applications in Edge and Fog computing environments and reviews the existing scheduling approaches.

2. Investigates efficient scheduling policy for network resources to optimize the total throughput of the network while mitigating the interference in dense and ultra-dense Edge and Fog computing environments (addresses the Q1).

    - A mathematical model to optimize total throughput of the network in hierarchical Edge and Fog computing environments.
    - A distributed dynamic clustering algorithm to solve intra-cluster interference, by which Cluster Heads (CHs) adaptively control their cluster size based on the requested demands of their end-users.
    - A Fog-driven graph formation strategy to model the inter-cluster interference.
    - A Fog-driven graph coloring technique to define and assign a set of network resources to each specific cluster based on the average resource demands of each cluster.

- A Fog-driven graph relaxation technique to reduce the constraints of graph coloring problem.
- A policy-aware scheduling technique to distribute network resources for Edge servers.

3. Puts forward a distributed batch scheduling technique for concurrent DAG-based IoT applications to optimize the execution cost of IoT applications (addresses the Q2).

   - A mathematical model to minimize the weighted cost of running concurrent IoT applications in terms of the execution time of IoT applications and the energy consumption of IoT devices.
   - An innovative approach to combine tasks of different concurrent IoT applications for batch placement.
   - An optimized meta-heuristic technique for batch placement of concurrent IoT applications.
   - A fast failure recovery technique to assign failed tasks to appropriate servers in a timely manner.

4. Proposes distributed scheduling and migration management techniques for real-time IoT applications in hierarchical Fog computing environments (addresses the Q3).

   - A mathematical model to minimize the weighted cost of running real-time applications in terms of the execution time of IoT applications and the energy consumption of IoT devices in hierarchical Fog computing environments.
   - A mathematical model to minimize the weighted cost of the migration process in terms of the execution time of IoT applications and the energy consumption of IoT devices in hierarchical Fog computing environments.
   - A ranking-based distributed scheduling policy for real-time IoT applications.
   - A distributed migration management technique to minimize the downtime and service interruption in the pre-copy migration model.

- Failure recovery techniques for clustering of resources, scheduling, and migration of IoT applications.

5. Proposes distributed DRL-based scheduling framework to learn and optimize complex scheduling of DAG-based IoT applications in Edge and Fog computing environments (addresses the Q4).

   - A mathematical model to minimize the weighted cost of running IoT applications in terms of the execution time of IoT applications and the energy consumption of IoT devices.
   - A DRL model for the scheduling of DAG-based IoT applications.
   - An action, reward, and state management methods for DRL framework.
   - A distributed actor-critic DRL-based model for off-policy learning of optimal policy for Edge and Fog computing environment.
   - Evaluation and validation using simulation and testbed experiments, consisting of heterogeneous Fog and Cloud servers.

6. Develops a software system for scheduling IoT applications in Edge and Fog computing environments.

   - A system configuration consisting of multiple Cloud datacenters, multiple Edge/Fog servers, and different IoT applications.
   - Software system's modules for the scheduling of IoT applications.
   - Several containerized IoT applications implemented and integrated with the software system.
   - Implementation and integration of scheduling algorithm with software system using practical implementation.
   - Evaluation and validation of scheduling algorithm's performance in real Fog computing environments.

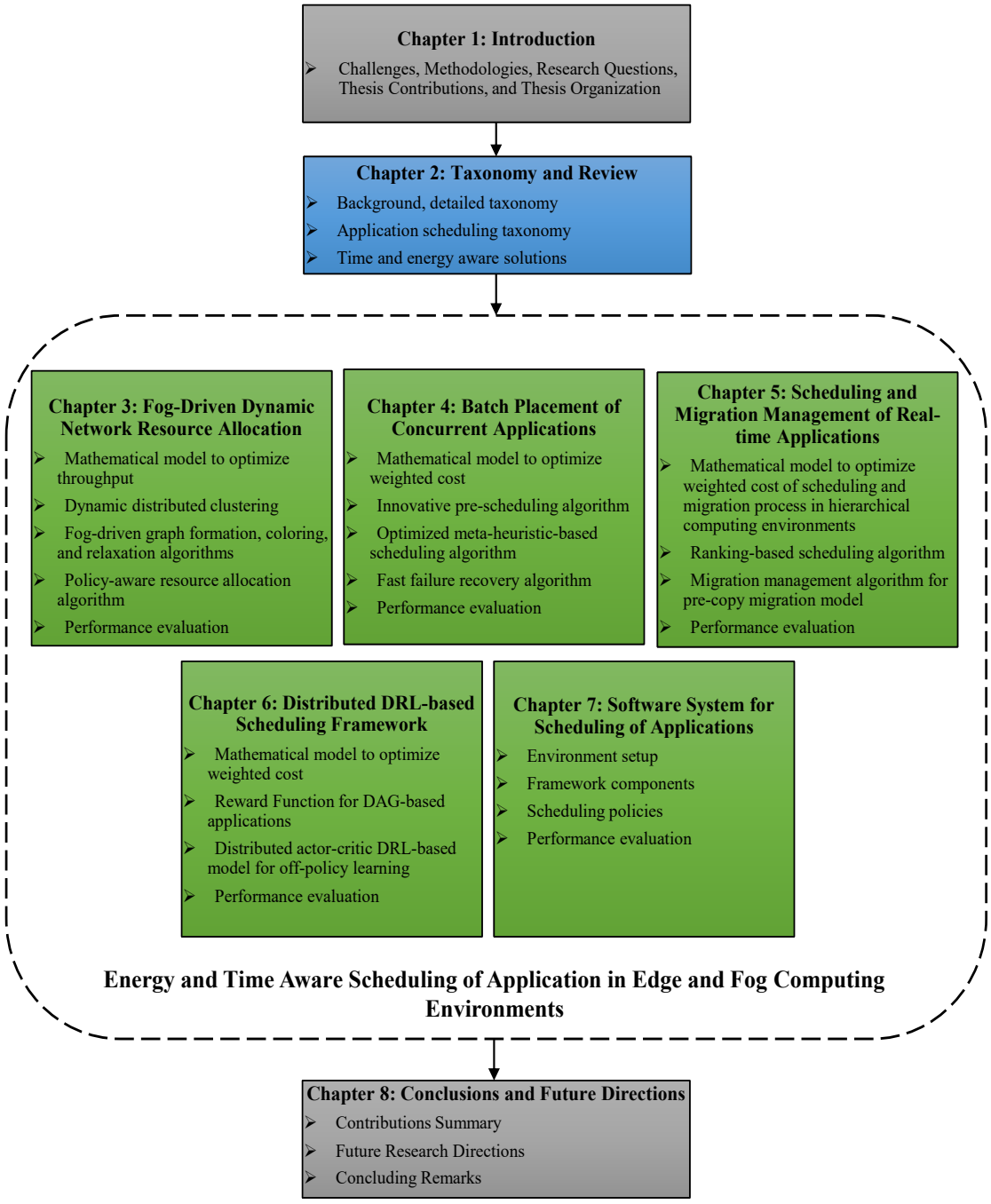


Figure 1.5: The thesis structure

1.5 Thesis Organization

The structure of this thesis is shown in Fig. 1.5. The remaining part of this thesis is organized as follows:

- Chapter 2 presents a taxonomy and literature review on scheduling IoT applications in Edge and Fog computing environments. This chapter is derived from:

  - **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "Scheduling IoT Applications in Edge and Fog Computing Environments: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, USA, 2022 (revision).

- Chapter 3 presents efficient Fog-driven scheduling policies for network resources to optimize the total throughput of the network while mitigating the interference in dense and ultra-dense Edge and Fog computing environments. This chapter is derived from:

  - **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Fog-driven Dynamic Resource Allocation Technique in Ultra-Dense Femtocell Networks", *Journal of Network and Computer Applications (JNCA)*, Volume 145, ISSN: 1084-8045, Elsevier Press, Amsterdam, Netherlands, November 2019.

- Chapter 4 presents a batch placement scheduling technique for concurrent IoT applications to optimize the execution cost of IoT applications. This chapter is derived from:

  - **Mohammad Goudarzi**, Huaming Wu, Marimuthu Palaniswami, and Rajkumar Buyya, "An Application Placement Technique for Concurrent IoT Applications in Edge and Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC)*, Volume 20, Number 4, Pages: 1298-1311, ISSN: 1536-1233, IEEE Press, New York, USA, January 2020.

- Chapter 5 presents techniques for scheduling and migration management of real-time IoT applications in hierarchical Edge and Fog computing environments. This chapter is derived from:

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Distributed Application Placement and Migration Management Techniques for Edge and Fog Computing Environments", *Proceedings of the 16th Conference on Computer Science and Intelligent Systems (FedCSIS)*, IEEE Press, Pages: 37-56, Online, Poland, September 2-5, 2021.

- Chapter 6 proposes distributed DRL-based scheduling framework to learn and optimize complex scheduling of DAG-based IoT applications in Edge and Fog computing environments. This chapter is derived from:

  - **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Distributed Deep Reinforcement Learning Technique for Application Placement in Edge and Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC)*, (in press, DOI: 10.1109/TMC.2021.3123165, accepted on 23 October, 2021).

- Chapter 7 proposes a software system for scheduling IoT applications in Edge and Fog computing environments. This chapter is derived from:

  - **Mohammad Goudarzi**, Qifan Deng, and Rajkumar Buyya, "Resource Management in Edge and Fog Computing using FogBus2 Framework", *Managing Internet of Things Applications across Edge and Cloud Data Centres*, Rajiv Ranjan, Karan Mitra, Prem Prakash Jayaraman, Albert Y. Zomaya (eds), ISBN: 978-1785617799, IET Press, Hertfordshire, UK, June 2022 (in press).

- Chapter 8 concludes the thesis by summarizing the findings and offers new directions for future research.

Chapter 2

A Taxonomy and Review on Scheduling IoT Applications

This chapter investigates the existing scheduling techniques in Fog computing and reviews them in terms of different perspectives, namely, application structure, environmental architecture, optimization characteristics, decision engine properties, and performance evaluation. Based on an in-depth analysis of the literature, separate taxonomies for each perspective on scheduling IoT applications in Fog computing environments are proposed. A detailed survey of existing approaches is conducted according to the taxonomy. Finally, the research gaps for further improvement of the Fog computing paradigm are identified and discussed.

2.1 Introduction

Fog computing paradigm provides a scalable solution for integrating a diverse range of hardware and software technologies to offer a wide variety of services for end-users. The Fog computing environment is highly heterogeneous in terms of end-users' devices, IoT applications, infrastructures, communication protocols, and deployed frameworks. Hence, the smooth execution of IoT applications in this highly heterogeneous computing environment depends on a large number of contributing parameters, making the efficient scheduling of IoT applications an important and yet a challenging problem in Fog computing environments.

This chapter is derived from:

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "Scheduling IoT Applications in Edge and Fog Computing Environments: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, USA, 2022 (revision).

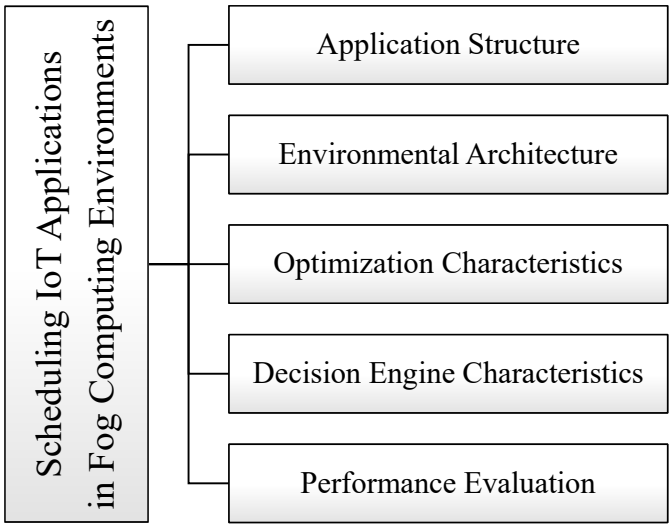


Figure 2.1: Different perspectives of scheduling IoT applications in Fog computing

Several techniques for scheduling IoT applications in Fog computing environments have been proposed. Several works focused on the structure of IoT applications and how these parameters affect the scheduling [30–32] while some other techniques mainly focus on environmental parameters of Fog computing, such as the effect of hierarchical Fog layers on the scheduling of IoT applications [33, 34]. Besides, several techniques focus on defining specific optimization models to formulate the effect of different parameters such as FSs' computing resources, networking protocols, and IoT devices characteristics, just to mention a few [35]. Moreover, several works have proposed different placement techniques to find an acceptable solution for the optimization problem [36–38] while some other techniques consider mobility management [34, 39, 40] and failure recovery [3, 41]. All these perspectives directly affect the scheduling problem, especially when designing decision engines. These perspectives should be simultaneously considered when studying and evaluating each proposal. However, very few initiatives have been found that simultaneously categorize scheduling techniques used for IoT applications in Fog computing environments from different perspectives. Hence, in this chapter, We identify five important perspectives regarding scheduling IoT applications in Fog computing environments, namely application structure, environmental architecture, optimization models, decision engines' characteristics, and performance

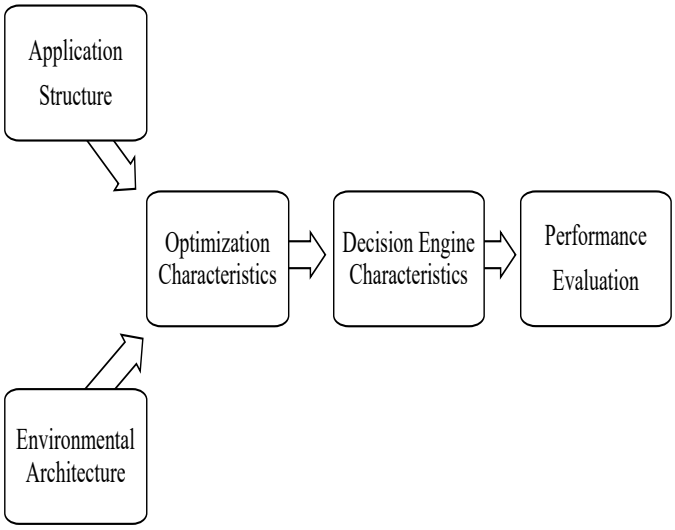


Figure 2.2: The relation among different identified perspectives

evaluation, as shown in Fig. 2.1. Considering each perspective, we present a taxonomy and review the existing proposals. Next, based on the studied works, we identify the research gaps in each perspective.

Fig. 2.2 depicts the relationships among identified perspectives. The features of application structure and environmental architecture help define the optimization characteristic and formulate the problem. Then, an efficient decision engine is required to effectively solve the optimization problem. Besides, the performance of the decision engine should be monitored and evaluated based on the main goal of optimization for the target applications and environment. Considering each perspective, we present a taxonomy and review the existing proposals. Finally, based on the studied works, we identify the research gaps in each perspective and discuss possible solutions.

The rest of this chapter is organized as follows. The existing related surveys and taxonomies on scheduling IoT applications in Fog computing environments are studied and compared with ours in Section 2.2. Section 2.3 presents a taxonomy and overview of the IoT applications' structure. Section 2.4 introduces a taxonomy on environmental properties of resources in Fog computing environments and studies the existing works accordingly. In Section 2.5, a taxonomy of optimization characteristics of problems in Fog computing environments is introduced. Section 2.6 identifies the important as-

pects of decision engines and presents a taxonomy of decision engines for scheduling IoT applications. Section 2.7 demonstrates the approaches and metrics used to evaluate scheduling strategies in Fog computing. Finally, Section 2.8 describes important design options for scheduling IoT applications, and Section 2.9 concludes this chapter.

2.2 Related Surveys

In the context of Fog computing, surveys targeted different aspects of Fog computing, such as security [42–45], smart cities [46], live migration techniques [47], existing software and hardware [16], deep learning applications [48], healthcare [49], and general surveys studied the Fog computing paradigm, its scope, architectures, and recent trends [50–57]. Also, some surveys mainly discussed resource management, application management, and scheduling in the context of Fog computing, such as [30, 35, 58–64], that we discuss and compare them with ours.

Aazam et al. [58] reviewed enabling technologies and research opportunities in Fog computing environments alongside studying computation offloading techniques in different domains such as Fog, Cloud, and IoT. Hong et al. [65] and Ghobaei-Arani [63] studied resource management approaches in Fog computing environments and discussed the main challenges for resource management. Yousefpour et al. [59] discussed the main features of the Fog computing paradigm and compared it with other related computing paradigms such as Edge and Cloud computing. Besides, it studied the foundations, frameworks, resource management, software, and tools proposed in Fog computing. Mahmud et al.[30] mainly discussed the application management and maintenance in Fog computing and proposed a taxonomy accordingly. Salaht et al. [60] presented a survey of current research conducted on service placement problems in Fog Computing and categorized these techniques. Shakarami et al. [62] studied machine learning-based computation offloading approaches while Adhikari et al. presented the type and applications of nature-inspired algorithms in the Edge computing paradigm. Martinez et al. [61] mainly focused on designing and evaluating Fog computing systems and frameworks. Lin et al. [35] and Sonkoly et al. [66] mainly studied and categorized different approaches for modeling the resources and communication types for compu-

Table 2.1: Summary of literature surveys on scheduling in Edge and Fog computing

Survey	Application Structure		Environmental Architecture		Optimization Characteristics		Decision Engine		Performance Evaluation		Conceptualize Scheduling Framework	Research Gap (in Years)
	Taxonomy	Research Gaps	Taxonomy	Research Gaps	Taxonomy	Research Gaps	Taxonomy	Research Gaps	Taxonomy	Research Gaps		
[58]	○	◐	○	○	○	○	○	○	○	○	●	4
[65]	○	○	◐	◐	◐	○	◐	○	○	○	○	3.5
[59]	○	○	◐	◐	○	○	○	◐	○	○	○	3
[63]	○	○	○	○	○	○	◐	◐	○	○	○	2.5
[60]	○	○	○	○	◐	●	◐	◐	●	◐	○	2
[30]	●	●	○	◐	○	○	◐	◐	○	○	●	1.5
[62]	○	○	○	○	○	○	◐	◐	○	○	○	1.5
[61]	○	○	○	○	○	○	○	◐	◐	◐	○	1.5
[35]	○	○	◐	○	●	◐	○	◐	○	○	○	1.5
[64]	○	◐	◐	○	○	○	○	◐	○	○	○	1
[66]	◐	○	○	○	◐	◐	○	○	○	○	○	0.5
[67]	○	○	○	○	○	◐	○	◐	○	○	○	0.5
This Survey	●	●	●	●	●	●	●	●	●	●	●	Current

●: Full Cover, ◐: Partial Cover, ○ : Does Not Cover

tation offloading in Edge computing. Finally, Islam et al. [64] proposed a taxonomy for context-aware scheduling in Fog computing and surveyed the related techniques in terms of contextual information such as user and networking characteristics.

Table 2.1 summarizes the characteristics of related surveys and provides a qualitative comparison with our work. The proper scheduling of IoT applications in Fog computing environments can be viewed from different perspectives, such as application structure, environmental architecture, optimization models, and the features of decision engines. Besides, the performance of scheduling techniques should be continuously evaluated to offer the best performance. As depicted in Table 2.1, the existing surveys barely study and provide comprehensive taxonomy for the above-mentioned perspectives. In this work, we identify the main parameters of each perspective and present a taxonomy accordingly. Moreover, we identify related research gaps and provide future directions to improve the Fog computing paradigm.

2.3 Application Structure

The primary goal of Fog computing is to offer efficient and high-quality service to users with heterogeneous applications and requirements. Hence, service providers require a comprehensive understanding of each IoT application structure (e.g., workload model and latency requirements) to better capture its complexities, perform efficient scheduling and resource management, and offer high-quality service to the users. Also, when designing the architecture of each IoT application, dynamics, constraints, and complex-

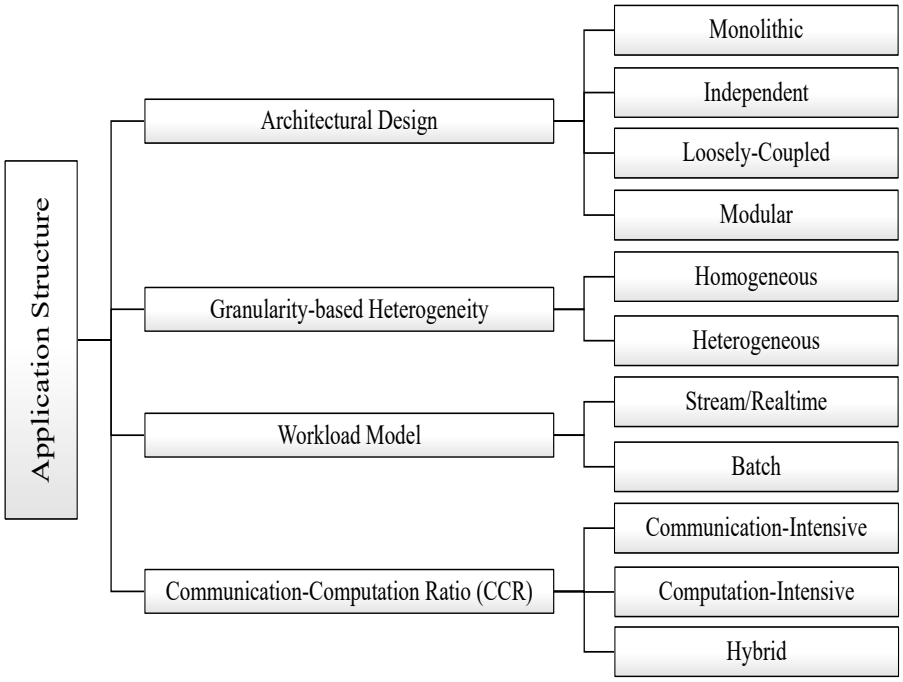


Figure 2.3: Application structure taxonomy

ities of resources in Fog computing should be carefully considered to exploit the potential of this paradigm. Fig. 2.3 presents a taxonomy and main elements of IoT application structure, described below.

2.3.1 Architectural Design

The logic of IoT applications can be implemented in different ways. To illustrate, operations of a Video Optical Character Recognition (VideoOCR) such as capturing frames, similarity check, and text extraction can be implemented as a single encapsulated program or as a set of interdependent components [1]. Hence, according to the granularity level of applications, their distribution, and coupling intensity, the architectural design of applications can be classified into four types:

Monolithic

It encapsulates the complete logic of an application as a single component or program. The parallel execution of these applications can be obtained using multi processing approaches [30]. In the context of Fog computing, several works such as [68–71] have considered monolithic applications.

Independent

These applications require the execution of a set of independent tasks or components for the complete execution of the application. The constituent parts of these applications can be simultaneously executed on different FSs or CSs. Several works such as [72–75] discuss applications with independent components or tasks in the Fog literature.

Modular

Each modular application is composed of a set of dependent tasks or components. While constituent parts of each application can be distributed over several FSs or CSs for parallel execution, there are some constraints for the execution of tasks based on their data-flow dependency model. Several works in the literature such as [76–79] discuss modular applications.

Loosely-coupled

Components of loosely-coupled applications (i.e., microservices) can be distributed over several CSs or FSs. Besides, due to service-level isolation, components of applications can be shared among different applications, providing high application extendability. Several works such as [1, 34, 80, 81] have considered loosely-coupled applications.

2.3.2 Granularity-based Heterogeneity

Tasks within an IoT application have different properties such as computation size, input size, output size, and deadline. These features affect the scheduling complexity, where

identifying the dynamics of applications with heterogeneous task properties requires further effort. Accordingly, we categorize IoT applications based on their granularity-level specifications to 1) **Heterogeneous** such as [77, 82] or 2) **Homogeneous** such as [73, 83].

2.3.3 Workload Model

The workload model specifies the data architecture of an application, which can be broadly divided into two categories for IoT applications:

Stream/Real-time

In this category, the data should be processed by the servers as soon as it was generated (i.e., real-time), and hence, the data usually require relatively simple transformation or computation. Several works such as [84–86] discuss stream workload for IoT applications.

Batch

In batch processing, the input data of an application is usually bundled for processing. However, contrary to heavy batch processing models, IoT applications often use micro-batches to provide a near-realtime experience. In the literature, several works such as [87–89] consider batch workload for the applications.

2.3.4 Communication-Computation Ratio (CCR)

Each IoT application, regardless of its architecture, contains some amount of input data for transmission and computational load for processing. These properties can significantly affect the scheduling decision to find proper FSs or CSs for an application. The CCR defines whether an application on average is more 1) **Computation-intensive** [32, 90, 91] or 2) **Communication-intensive** [68, 92, 93]. Besides, some works consider a

range of applications to cover both computation-intensive and communication-intensive applications, to which we refer as 3) **Hybrid** [3, 94].

2.3.5 Discussion

In this section, we discuss the effects of identified application structure's elements on the decision engine and describe the lessons that we have learned. Besides, we identify several research gaps accordingly. Table 2.2 summarizes the characteristics related to IoT application structure in Fog computing.

Effects on the decision engine

The application structure properties affect the decision engine in various aspects, as briefly described below.

1. Architectural design: It defines the number of tasks/modules and their respective dependency within a single application. Hence, as the number of tasks/modules per application increases, the problem space significantly increases. Considering an application with n number of tasks and m possible candidate configuration per task, the Time Complexity (TC) of finding the optimal solution is $O(m^n)$. Besides, the dependency of tasks within an application imposes hard constraints on the problem, which further increases the complexity. Thus, finding the optimal solution for the scheduling of applications becomes very time-consuming, and the design of an efficient placement technique to serve applications in a timely manner remain an important yet challenging problem.

2. Granularity-based heterogeneity: It shows the corresponding properties of each task/module within an application and plays a principal role in identifying the dynamics of applications. One of the most important features of decision engines is their adaptability and their capability to extract the complex dynamics of applications so that the decision engine can receive diverse types of applications' requests. Since applications with heterogeneous granularity-based properties have higher dynamics' complexity, the

decision engines designed for this application category should support high adaptability.

3. Workload model and CCR: These elements provide insightful information regarding the input data architecture of the application and its behavior in the runtime. Accordingly, the decision engine may define different priority queues for incoming requests based on their workload model and CCR to provide higher QoS for the users. For example, applications with real-time workload types and communication-intensive CCR may have higher priority for the placement on servers closer to the IoT devices than computation-intensive applications that are not real-time.

Lessons learned

Our findings regarding the IoT application structure in the surveyed works are briefly described in what follows:

1. Almost 70% of the surveyed works have overlooked studying the dependency model of tasks within an application and selected either the independent or monolithic design. The rest of the works consider dependency among tasks of an application in different models (i.e., sequential, parallel, or hybrid dependency). Moreover, only about 10% of the studied works consider microservices in their application design.

2. The most realistic assumption for the granular properties of each task/module is heterogeneous (i.e., heterogeneous input size, output size, and computation size). Almost 85% of the studied works consider heterogeneous properties for each task/module, while around 15% of the works consider the homogeneous properties for the tasks/-modules.

3. The workload model and CCR in each proposal depend on the targeted application scenarios. Almost 55% of the works did not study the CCR, or the required information to obtain the CCR (i.e., computation size of tasks, average data size) was not mentioned.

Among the rest of the works, computation-intensive, communication-intensive, and hybrid CCR form roughly 25%, 15%, and 5% of proposals respectively.

Research Gaps

We have identified several open issues for further investigation, that are discussed below:

1. According to Alibaba's data of 4 million applications, more than 75% of the applications consist of dependent tasks [95]. However, only around 30% of the recent works surveyed in this study consider applications with dependent tasks (i.e., modular or loosely-coupled), showing further investigation is required to identify the dynamics of these types of complex applications.

2. Although the microservice-based applications can significantly benefit the IoT scenarios, only a few works such as [34, 81] have studied the scheduling and migration of microservices in Edge/Fog computing environments. So, further investigation is required to study the behavior of microservice-based applications with different resource management techniques.

3. Modular or loosely-coupled IoT applications can be distributed over different FSs or CSs for parallel execution. However, several works such as [36] statically assign components of an application on pre-defined FSs or CSs and only schedule one or two remaining components. Hence, the best placement configuration of applications based on the current dynamics of the system cannot be investigated, leading to diminished performance gain.

4. When the number of IoT applications increases, there is a high probability that application requests with different workload models are submitted to the system. However, none of the studied works in the literature consider applications with different workload models and how they may mutually affect each other in terms of performance.

5. Due to the high heterogeneity of IoT applications in Fog, applications with diverse CCR may be submitted for processing, requiring special consideration such as networking and prioritization. Although there are only a few recent works such as [76, 84] that consider hybrid CCR, most of the recent works target one of the computation-intensive or communication-intensive applications.

Table 2.2: Summary of existing works considering application structure taxonomy

Ref	Application Structure			
	Design	Granularity Heterogeneity	Workload Model	CCR
[72]	Independent	Heterogeneous	Batch	Computation Intensive
[68]	Monolithic	Heterogeneous	Batch	Communication Intensive
[96]	Independent	Heterogeneous	Batch	Communication Intensive
[80]	Loosely-coupled	Heterogeneous	Batch	Not Defined
[76]	Modular	Heterogeneous	Batch	Hybrid
[3]	Modular	Heterogeneous	Batch	Hybrid
[70]	Monolithic	Homogeneous	Batch	Not Defined
[77]	Modular	Heterogeneous	Batch	Computation Intensive
[97]	Independent	Heterogeneous	Stream	Computation Intensive
[98]	Independent	Heterogeneous	Stream	Not Defined
[82]	Monolithic	Heterogeneous	Batch	Communication Intensive
[99]	Monolithic	Not Defined	Not Defined	Not Defined
[100]	Monolithic	Homogeneous	Batch	Computation Intensive
[81]	Loosely-coupled	Heterogeneous	Stream	Communication Intensive
[101]	Independent	Heterogeneous	Stream	Communication Intensive
[102]	Modular	Heterogeneous	Batch	Computation Intensive

[103]	Independent	Heterogeneous	Stream	Communication Intensive
[104]	Not Defined	Not Defined	Batch	Not Defined
[105]	Independent	Heterogeneous	Stream	Not Defined
[106]	Independent	Heterogeneous	Not Defined	Computation Intensive
[107]	Monolithic	Homogeneous	Not Defined	Computation Intensive
[108]	Modular	Heterogeneous	Batch	Not Defined
[90]	Loosely-coupled	Heterogeneous	Stream	Computation Intensive
[87]	Independent	Heterogeneous	Batch	Not Defined
[91]	Independent	Heterogeneous	Batch	Computation Intensive
[88]	Independent	Heterogeneous	Batch	Not Defined
[109]	Loosely-coupled	Heterogeneous	Stream	Computation Intensive
[110]	Independent	Heterogeneous	Stream	Computation Intensive
[111]	Loosely-coupled	Heterogeneous	Batch	Computation Intensive
[112]	Monolithic	Heterogeneous	Stream	Computation Intensive
[113]	Independent	Heterogeneous	Batch	Computation Intensive
[89]	Independent	Heterogeneous	Batch	Not Defined
[114]	Monolithic	Homogeneous	Batch	Not Defined
[115]	Independent	Heterogeneous	Not Defined	Not Defined
[116]	Monolithic	Homogeneous	Batch	Not Defined
[117]	Monolithic	Homogeneous	Batch	Computation Intensive
[118]	Loosely-coupled	Heterogeneous	Not Defined	Communication Intensive
[95]	Modular	Heterogeneous	Batch	Communication Intensive
[119]	Modular	Heterogeneous	Batch	Computation Intensive
[120]	Independent	Heterogeneous	Stream	Not Defined

[121]	Independent	Heterogeneous	Stream	Not Defined
[122]	Modular	Heterogeneous	Batch	Communication Intensive
[1]	Loosely-coupled	Heterogeneous	Stream	Not Defined
[92]	Monolithic	Homogeneous	Batch	Communication Intensive
[123]	Modular	Homogeneous	Batch	Not Defined
[124]	Independent	Heterogeneous	Not Defined	Computation Intensive
[125]	Independent	Heterogeneous	Stream	Not Defined
[126]	Independent	Heterogeneous	Batch	Not Defined
[127]	Modular	Heterogeneous	Batch	Hybrid
[36]	Modular	Heterogeneous	Not Defined	Not Defined
[128]	Modular	Heterogeneous	Stream	Not Defined
[129]	Monolithic	Heterogeneous	Batch	Computation Intensive
[73]	Independent	Homogeneous	Batch	Not Defined
[69]	Monolithic	Homogeneous	Not Defined	Not Defined
[71]	Independent	Heterogeneous	Batch	Communication Intensive
[130]	Independent	Heterogeneous	Batch	Not Defined
[131]	Independent	Heterogeneous	Batch	Communication Intensive
[132]	Independent	Heterogeneous	Not Defined	Not Defined
[133]	Independent	Heterogeneous	Batch	Not Defined
[78]	modular	Heterogeneous	Stream	Not Defined
[34]	Loosely-coupled	Heterogeneous	Stream	Computation Intensive
[134]	Independent	Heterogeneous	Batch	Computation Intensive
[135]	Monolithic	Heterogeneous	Not Defined	Not Defined
[136]	Loosely-coupled	Heterogeneous	Stream	Not Defined

[137]	Monolithic	Homogeneous	Batch	Not Defined
[138]	Monolithic	Homogeneous	Batch	Computation Intensive
[139]	Independent	Heterogeneous	Batch	Not Defined
[140]	Modular	Heterogeneous	Batch	Not Defined
[141]	Independent	Heterogeneous	Batch	Not Defined
[83]	Monolithic	Homogeneous	Not Defined	Not Defined
[142]	Independent	Heterogeneous	Batch	Not Defined
[143]	Monolithic	Heterogeneous	Batch	Computation Intensive
[144]	Independent	Heterogeneous	Batch	Not Defined
[145]	Independent	Heterogeneous	Not Defined	Not Defined
[146]	Loosely-coupled	Heterogeneous	Not Defined	Not Defined
[147]	Modular	Heterogeneous	Batch	Not Defined
[148]	Independent	Not Defined	Stream	Not Defined
[149]	Independent	Heterogeneous	Stream	Not Defined
[150]	Loosely-coupled	Heterogeneous	Batch	Not Defined
[151]	Independent	Heterogeneous	Batch	Not Defined
[94]	Modular	Heterogeneous	Batch	Hybrid
[84]	Independent	Heterogeneous	Stream	Hybrid
[152]	Independent	Heterogeneous	Batch	Computation Intensive
[153]	Independent	Heterogeneous	Batch	Not Defined
[154]	Modular	Heterogeneous	Batch	Not Defined
[155]	Independent	Heterogeneous	Batch	Not Defined
[85]	Independent	Heterogeneous	Stream	Not Defined
[156]	Modular	Heterogeneous	Batch	Not Defined

[86]	Independent	Heterogeneous	Stream	Not Defined
[157]	Independent	Heterogeneous	Stream	Not Defined
[32]	Independent	Heterogeneous	Stream	Computation Intensive
[158]	Independent	Heterogeneous	Stream	Not Defined
[79]	Modular	Heterogeneous	Stream	Not Defined
[75]	Independent	Heterogeneous	Batch	Not Defined
[159]	Loosely-coupled	Heterogeneous	Stream	Not Defined
[160]	Independent	Heterogeneous	Batch	Not Defined
[93]	Independent	Heterogeneous	Batch	Communication Intensive
[161]	Modular	Heterogeneous	Batch	Computation Intensive
[74]	Independent	Heterogeneous	Batch	Not Defined
[162]	Modular	Heterogeneous	Batch	Computation Intensive

2.4 Environmental Architecture

The configuration and properties of IoT devices and resource providers directly affect the complexity and dynamics of scheduling IoT applications. To illustrate, as the number of resource providers increases, heterogeneity in the system also grows as a positive factor, while the complexity of making a decision also increases that may negatively affect the process of making decisions. In this section, we classify the environmental architecture properties, as depicted in Fig. 2.4, into the following categories:

2.4.1 Tiering Model

IoT devices and resource providers can be conceptually organized in different tiers based on their proximity to users and resources, described below:

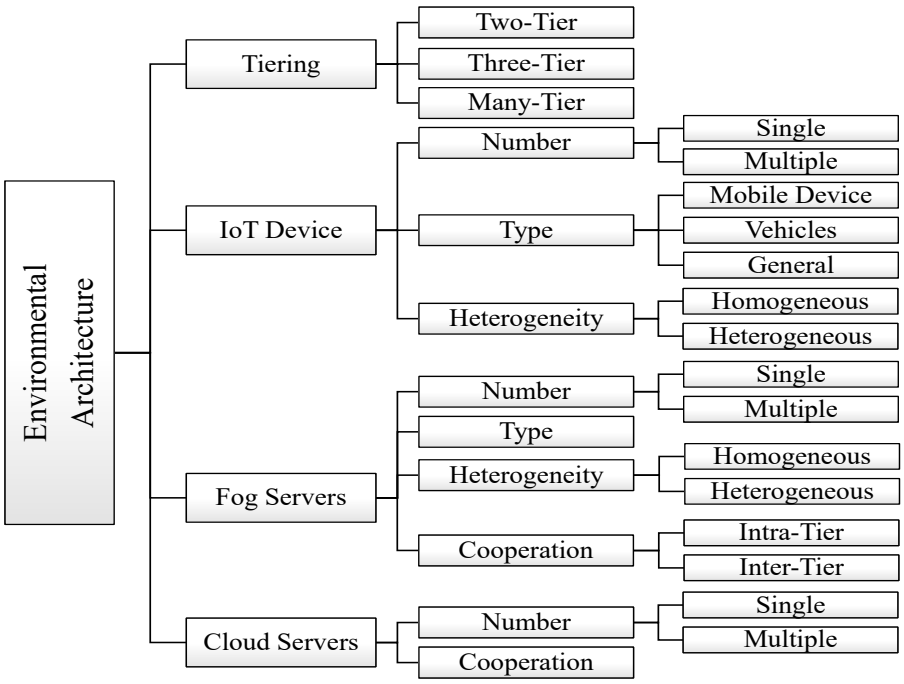


Figure 2.4: Environmental architecture taxonomy

Two-Tier

In this resource organization, IoT devices are situated at the bottom-most layer and resource providers are placed at the edge of the network in the proximity of IoT devices (i.e., Edge computing). Several works use two-tier resource organization such as [70, 73, 103, 156].

Three-Tier

Compared to two-tier model, this model also uses CSs at the highest-most layer to support edge resources (i.e., Fog computing). Several works considered three-tier model in the literature such as [85, 93, 154, 159].

Many-Tier

In many-tier resource organizations, IoT devices and CSs are situated at the bottom-most and highest-most tiers, while FSs are placed in between through several tiers (i.e.,

hierarchical Fog computing). In the literature, several works have considered many-tier model such as [34, 133, 142, 145].

2.4.2 IoT Devices

IoT devices can play two roles; as service requester and/or resource provider. When they act as service requesters, still they can execute a portion of their tasks or components based on their available resources. Moreover, IoT devices can simultaneously play these different roles. We study the properties of IoT devices from the following perspectives:

Number

The higher number of IoT devices (either as service requester or service provider), the higher complexity of the scheduling problem. Some works only consider single IoT device in the environment such as [77, 87, 102] while other works consider multiple IoT devices simultaneously such as [135, 137, 153].

Type

The type of IoT devices helps us understand the amount of resources, capabilities, and constraints of these devices. The IoT devices used in the current literature can be broadly classified into three categories, namely 1) **Mobile Devices (MD)** which are mostly considered as smartphones or tablets [80, 136, 162], 2) **Vehicles** [81, 101, 139], and 3) **General** devices containing a set of IoT devices, ranging from small sensors to drones [70, 76, 143].

Heterogeneity

We also study the resources of IoT devices and their request types, and classify proposals into 1) **Heterogeneous** where IoT devices have various resources and different request types and sizes such as [78, 123, 133] or 2) **Homogeneous** where the resources of IoT devices are the same or they have the same request type and size such as [77, 87, 109].

2.4.3 Fog Servers (FSs)

FSs usually act as resource providers for IoT devices. The environmental properties of FSs can be classified based on the following criteria:

Number

Similar to IoT devices, we classify the number of FSs in the environment into 1) **Single** and 2) **Multiple**. The complexity and dynamics of systems in works that have considered only single FS such as [99, 110, 120] is simpler than the works that have considered multiple FSs such as [104, 106, 108].

Type

The type of FSs acting as service provider in the Fog computing environment ranges from IoT devices with additional resources to resource-rich data centers. Several works have considered a specific type of FS and discussed their properties in their works such as 1) **Base Station (BS)** and **Macro-cell Station (MS)** [87, 96, 104], 2) **Femtocells** [89, 105, 163], 3) **Rpi** [118] and 4) **Access Points (AP)** [121, 129]. Moreover, several works consider 5) **General** FSs containing a set of FSs with different types such as [126, 133].

Heterogeneity

We study the FSs' resources and classified works based on their heterogeneity into 1) **Heterogeneous** and 2) **Homogeneous** accordingly. The majority of works have considered heterogeneous resources for FSs [72, 74, 76, 162] while some works consider homogeneous resources for FSs [99, 131, 161].

Cooperation

Compared to CSs, each FS has fewer resources and it may not be able to satisfy the requirements of IoT applications. Cooperation among FSs helps augmenting their resources and providing service for demanding IoT applications. We classify proposals

based on their cooperation among FSs into 1) **Intra-tier** where FSs of same tier collaborate to satisfy users' requests [1, 80, 80, 140] and 2) **Inter-tier** where FSs of different layers also collaborate for the execution of IoT one application [34, 76].

2.4.4 Cloud Servers (CSs)

The environmental properties of CSs can be classified based on the following criteria:

Number

The current literature based on the CSs' number can be divided into 1) **Single** and 2) **Multiple** categories. The majority of works only consider one CDC as resource provider (either as a central entity with aggregated resources or different number of VMs) to support FSs such as [85, 148, 153, 154]. In real-world environment, different CDCs are available which can provide services with different QoS for multiple applications. Some works such as [3, 76, 90, 164] have considered multiple CDCs with heterogeneous CSs in the literature.

Cooperation

Among the works considered multi CDCs, we study whether CSs from different CDCs are configured to collaboratively execute an IoT application or not. In the literature, some works such as [3, 75, 164] have considered collaborative multi CDCs scenarios.

2.4.5 Discussion

In this section, we discuss the effects of identified environmental architecture's elements on the decision engine and describe the lessons that we have learned. Besides, we identify several research gaps accordingly. Table 2.3 provides a summary of properties related to environmental architecture in Fog computing.

Effects on the decision engine

The elements of environmental architecture affect the decision engine in various aspects, as briefly described below.

1. Tiering: It represents the organization of end-users' devices and resources in the computing environment. Considering the properties of resources in different tiers, it helps find the most suitable deployment layer for the decision engine to efficiently serve IoT applications' requests with a wide variety of requirements. For example, to serve real-time IoT applications with low startup time requirements, the most suitable deployment layer in the three-tier model is the lowest-level Fog layer.

2. IoT devices: The number of IoT devices directly relates to the number of incoming requests to decision engines. It affects the admission control of decision engines. The type of IoT devices provides contextual information about the number of resources and intrinsic properties of the IoT devices that are important for the decision engine. For example, the MD type not only states that the IoT device does not have significant computing resources, but also presents that the device has mobility features. Thus, the IoT device type affects the advanced features of the decision engine, such as mobility, and also specifies whether the IoT devices can serve one or several tasks/modules of IoT applications or not.

3. Fog and Cloud servers: The number of available servers directly affects the TC of the scheduling problem. Considering an application with n number of tasks and m possible candidate configuration per task, the TC of finding the optimal solution is $O(m^n)$. Hence, it directly affects the choice of placement technique and scalability feature of the decision engine. As the problem space increases, a suitable decision engine should be selected to solve the scheduling problem. Moreover, the type and heterogeneity of resources provide further contextual information for the decision-making, such as the number of resources, networking characteristics, and resource constraints, just to mention a few.

Lessons learned

Our findings regarding the environmental architecture in the surveyed works are briefly described in what follows:

1. Almost 60% of works consider the three-tier model and many-tier models for the organization of end-users and resources. Not only do these works consider real-time applications, but also some of them assume both real-time and computation-intensive applications, such as [76, 94, 127]. This is mainly because these works use CSs as a backup plan for more computation-intensive applications or when the number of incoming IoT requests increases and the FSs cannot solely manage the incoming requests. Moreover, nearly 40% of surveyed works assume a two-tier model for the organization of end-users and resources. These works mostly assume real-time workload type and communication-intensive applications for the deployment on Edge servers, such as [68, 84, 110, 120].

2. In the surveyed works, almost 90% of the works considered an environment with multiple IoT devices, while 10% of works only focused on a single IoT device. When the number of IoT devices increases, the diversity of IoT applications and heterogeneity of their tasks also increase accordingly. Moreover, the greater number of works assume IoT devices as general devices with sensors, actuators, and diverse application requests. In contrast, some works targeted a specific IoT devices such as mobile devices and vehicles with almost 30% and 10% of proposals, respectively. These proposals studied other contextual information of targeted IoT devices in detail such as mobility [87], energy consumption [101], and networking characteristics [81, 140]. Finally, about 90% of works have studied IoT devices with heterogeneous properties and diverse application request types, which are the closest scenario to real-world computing environments.

3. Regarding Fog resources, almost 90% of the proposals consider multiple FSs in the environment. However, only 40% of the current literature has considered any cooperation model among FSs. There is a high probability that a single FS cannot solely manage the execution of several incoming resources due to its limited resources. Also, sending

partial/complete applications' tasks to the Cloud may negatively affect IoT devices' response time and energy consumption, especially for real-time IoT applications. Thus, cooperation among FSs is of paramount importance that can lead to the execution of IoT applications with better performance and QoS. Considering the type of the FSs, about 60% of the studied literature considered general FSs. The rest of the works studied a specific type of FSs and tried to involve their contextual information in the scheduling process of IoT applications, such as networking characteristics [72]. Moreover, some works considered IoT devices can simultaneously play different roles in the computing environments (i.e., service requester and service provider) such as [90, 101, 141].

4. In the current literature, around 60% of the works consider CSs as computing resources in the environment. However, only in 8% of these works multiple Cloud service providers (i.e., multi-Cloud), their communication, and interactions are studied, such as [3, 75, 76, 90].

Research Gaps

We have identified several open issues for further investigation, as discussed below:

1. Hierarchical Fog computing (i.e., multi-tier) has not been thoroughly considered by researchers. Only a few works (almost 5%) consider the organization of resources in the multi-tier environment, and most have focused on the heterogeneity of resources among different tiers. However, these works have not considered the heterogeneity of communication protocols and latency in multi-tier environments.

2. In the literature, several works have considered abstract CDC as a central unit with huge computing capacity [125], while in reality, CDCs contain several CSs hosting computing instances. Such assumptions affect the computing and communication time in simulation studies.

3. One of the main advantages of Fog computing is providing heterogeneous FSs in IoT devices' vicinity to collaboratively serve applications. However, many works have

not considered cooperation among FSs. In this case, due to the limited computing and communication resources of each FS and a large number of IoT requests, the serving FS may become a bottleneck which negatively affects the response time and QoS [163]. Besides, in uncooperative scenarios, the overloaded FS forwards requests to CSs, incurring higher latency. Hence, cooperative Fog computing, associated protocols, and constraints require further investigation for different IoT application scenarios.

Table 2.3: Summary of existing works considering environmental architecture taxonomy

Ref	Environmental Architecture									
	Tiering	IoT Device			Fog Servers				Cloud Servers	
		Number	Type	Hetero	Number	Type	Hetero	Coop	Number	Coop
[72]	Two-Tier	Multiple	MD	Hetero	Multiple	BS, MS	Hetero	○	○	○
[68]	Two-Tier	Multiple	Vehicle	Hetero	Multiple	RSU	ND	○	○	○
[96]	Three-Tier	Multiple	MD	ND	Multiple	BS	Hetero	Intra	Single	○
[80]	Three-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	Intra	Single	○
[76]	Many-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra, Inter	Multiple	●
[3]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	Multiple	●
[70]	Two-Tier	Multiple	General	Hetero	Multiple	Cloudet	Hetero	○	○	○
[77]	Three-Tier	Single	General	Homo	Multiple	General	Hetero	Intra	Single	○
[97]	Three-Tier	Multiple	General	Homo	Multiple	General	Hetero	ND	Single	○
[98]	Two-Tier	Multiple	General	ND	Multiple	BS, MS	Hetero	○	○	○
[82]	Three-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	Intra	Single	○
[99]	Two-Tier	Multiple	MD	Hetero	Single	BS	Homo	○	○	○
[100]	Two-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	Intra	○	○
[81]	Three-Tier	Multiple	Vehicle	Hetero	Multiple	Hybrid	Hetero	Intra	Single	○
[101]	Three-Tier	Multiple	Vehicle	Hetero	Multiple	BS, Vehicle	Homo	○	Single	○

[102]	Three-Tier	Single	MD	Hetero	Single	General	Homo	○	Single	○
[103]	Two-Tier	Multiple	MD	Hetero	Multiple	General	Hetero	○	○	○
[104]	Three-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	Intra	Single	○
[105]	Three-Tier	Multiple	MD	Hetero	Multiple	Femto	Hetero	Intra	Single	○
[106]	Three-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	Intra	Single	○
[107]	Two-Tier	Multiple	MD	Hetero	Multiple	BS, MD	Hetero	Intra	○	○
[108]	Three-Tier	Multiple	ND	ND	Multiple	General	Hetero	Intra	Single	○
[90]	Three-Tier	Multiple	MD	Hetero	Multiple	General, MD	Hetero	Intra	Multiple	●
[87]	Two-Tier	Single	MD	Homo	Single	BS	Homo	○	○	○
[91]	Two-Tier	Multiple	MD	Hetero	Multiple	BS, MS	Hetero	○	○	○
[88]	Two-Tier	Multiple	MD	Hetero	Multiple	General	Hetero	○	○	○
[109]	Two-Tier	Multiple	MD	Homo	Single	General	Homo	○	○	○
[110]	Two-Tier	Multiple	General	Hetero	Single	General	Homo	○	○	○
[111]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	Single	○
[112]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	Single	○
[113]	Three-Tier	Multiple	General	Hetero	Multiple	General	Homo	Intra	Single	○
[89]	Two-Tier	Multiple	MD	Hetero	Multiple	BS, Femto	Hetero	○	○	○
[114]	Two-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	○	○	○
[115]	Three-Tier	Multiple	General	ND	Multiple	General	Hetero	○	Single	○
[116]	Two-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	○	○

[117]	Two-Tier	Multiple	General	Hetero	Multiple	BS	Hetero	○	○	○
[118]	Three-Tier	Single	General	Hetero	Single	Rpi	Homo	Intra	Single	○
[95]	Two-Tier	Multiple	General	Hetero	Multiple	Hybrid	Homo	ND	○	○
[119]	Two-Tier	Multiple	General	Hetero	Single	General	Homo	○	○	○
[120]	Two-Tier	Multiple	General	Hetero	Single	General	Homo	○	○	○
[121]	Two-Tier	Multiple	General	Hetero	Multiple	AP	Hetero	○	Single	○
[122]	Three-Tier	Single	ND	ND	Single	General	Homo	○	Single	○
[1]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	Single	○
[92]	Three-Tier	Multiple	General	Hetero	Multiple	BS	Hetero	Intra	Single	○
[123]	Three-Tier	Multiple	General	Hetero	Single	Cloudlet	Homo	○	Single	○
[124]	Three-Tier	Multiple	MD	Hetero	Multiple	General	Homo	○	Single	○
[125]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[126]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[127]	Three-Tier	Single	General	Hetero	Multiple	General	Hetero	Intra	Single	○
[36]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	Single	○
[128]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	○	○
[129]	Three-Tier	Multiple	MD	Hetero	Multiple	AP	Hetero	○	Single	○
[73]	Two-Tier	Multiple	MD	Homo	Single	AP	Homo	○	○	○
[69]	Two-Tier	Single	General	Homo	Multiple	General	Hetero	○	○	○
[71]	Two-Tier	Single	MD	Hetero	Multiple	BS	Homo	○	○	○

[130]	Two-Tier	Multiple	MD	Hetero	Single	General	Homo	○	○	○
[131]	Two-Tier	Multiple	MD	Hetero	Single	General	Homo	○	○	○
[132]	Three-Tier	Multiple	MD	Hetero	Multiple	General	Hetero	○	Single	○
[133]	Many-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[78]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	Single	○
[34]	Many-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra, Inter	Single	○
[134]	Two-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	○	○	○
[135]	Two-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	Intra	○	○
[136]	Three-Tier	Single	MD	Homo	Multiple	Hybrid	Hetero	○	Single	○
[137]	Three-Tier	Multiple	MD	Hetero	Multiple	AP	Hetero	○	Single	○
[138]	Two-Tier	Multiple	MD	Hetero	Multiple	BS	Hetero	Intra	○	○
[139]	Two-Tier	Multiple	Vehicle	ND	Multiple	AP	Hetero	Intra	○	○
[140]	Two-Tier	Multiple	Vehicle	Hetero	Multiple	BS	Hetero	Intra	○	○
[141]	Two-Tier	Multiple	Vehicle	Hetero	Multiple	BS, Vehicle	Hetero	Intra	○	○
[83]	Two-Tier	Single	MD	Homo	Multiple	General	Hetero	ND	○	○
[142]	Many-Tier	Multiple	MD	Hetero	Multiple	General	Hetero	ND	Single	○
[143]	Two-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	○	○
[144]	Two-Tier	Single	Vehicle	Hetero	Multiple	AP, Vehicle	Hetero	Intra	○	○
[145]	Many-Tier	Multiple	General	Hetero	Multiple	General	Hetero	ND	Single	○

[146]	Two-Tier	Multiple	General	Hetero	Multiple	General	Hetero	ND	ND	○
[147]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	ND	Multiple	○
[148]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[149]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[150]	Two-Tier	Multiple	Vehicle	Hetero	Multiple	General, Vehicle	Hetero	Intra	○	○
[151]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[94]	Three-Tier	Multiple	General	ND	Multiple	General	Hetero	Intra	Single	○
[84]	Two-Tier	Multiple	General	Homo	Multiple	General	Hetero	Intra	○	○
[152]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[153]	Three-Tier	Multiple	General	ND	Multiple	General	Hetero	○	Single	○
[154]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[155]	Two-Tier	Multiple	General	ND	Multiple	General	Hetero	Intra	○	○
[85]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	○	Single	○
[156]	Two-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	○	○
[86]	Three-Tier	Multiple	General	Hetero	Multiple	BS	Hetero	○	Single	○
[157]	Three-Tier	Multiple	General	Hetero	Multipe	General	Hetero	○	Single	○
[32]	Many-Tier	Multiple	General	Hetero	Multipe	General	Hetero	Intra	Single	○
[158]	Three-Tier	Multiple	General	ND	Multipe	General	Hetero	○	Single	○
[79]	Many-Tier	Multiple	General	Hetero	Multipe	General	Hetero	Intra	Single	○

[75]	Three-Tier	Multiple	General	Hetero	Multipe	General	Hetero	Intra	Multiple	●
[159]	Three-Tier	Multiple	General	ND	Multipe	General	Hetero	Intra	single	○
[160]	Three-Tier	Multiple	ND	ND	Multipe	General	Hetero	ND	single	○
[93]	Three-Tier	Multiple	General	Hetero	Multipe	General	Hetero	ND	single	○
[161]	Three-Tier	Multiple	MD	Hetero	Single	General	Homo	○	Single	○
[74]	Three-Tier	Multiple	General	Hetero	Multiple	General	Hetero	Intra	Single	○
[162]	Three-Tier	Single	MD	Homo	Multiple	General	Hetero	Intra	Single	○

Hetero: Heterogeneity/Heterogeneous, Homo: Homogeneous, Coop: Cooperation, MD: Mobile Device, BS: Base Station,

MS: Macrocell Station, AP: Access Point, RSU: Roadside Units, Femto: Femtocell, ● : Yes, ○ : No

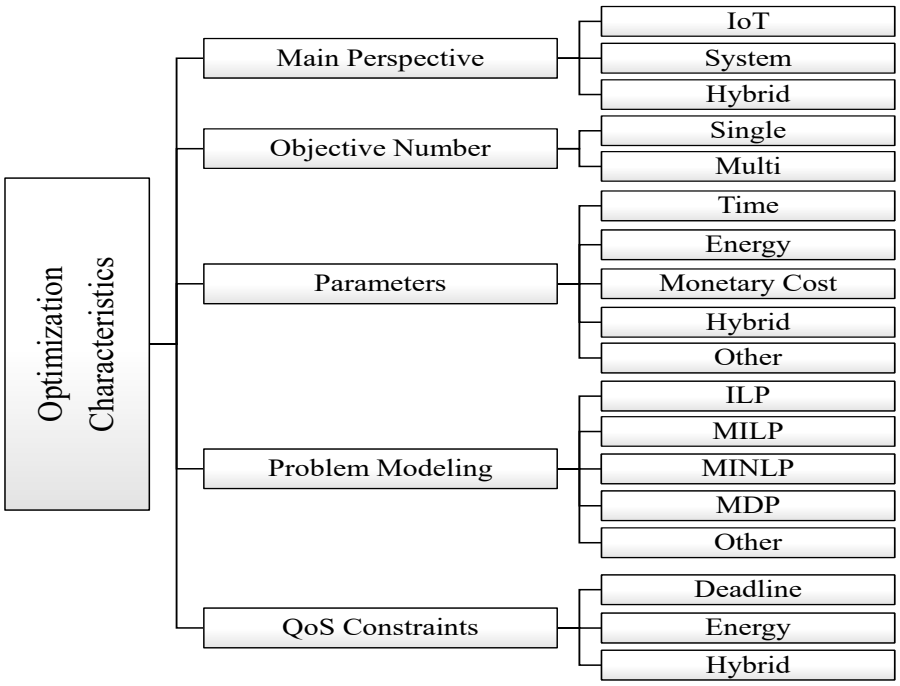


Figure 2.5: Optimization characteristics taxonomy

2.5 Optimization Characteristics

Considering the application structure, environmental parameters, and the target objectives, each proposal formulates the problem of scheduling IoT applications in Fog computing. Optimization parameters directly affect the selection process and properties of suitable decision engines. Fig. 2.5 presents the principal elements in optimization characteristics, as described in what follows:

2.5.1 Main Perspective

The proposals in the literature can be divided into three categories based on their main optimization goal, namely IoT devices/users, system, and hybrid, which are described in the following:

IoT devices/users

Main perspective of several proposals is to satisfy the requirements of IoT applications such as minimizing their execution time and energy consumption of IoT devices, or improving user experience in terms of QoS and Quality of Experience (QoE). Several works have considered IoT perspective for optimization such as [68, 127, 131, 150, 150, 161].

System

The main perspective of this category is to improve the efficiency of resource providers such as minimizing their energy consumption, improving resource utilization, and maximizing the monetary profit [89, 108, 115, 153]. Hence, these works often assume IoT devices with very limited limited computational resources that transfer sensed data to the surrogate servers for processing and storage.

Hybrid

Some proposals targeted optimizing the parameters of both IoT devices/users and resource providers, referred to as hybrid optimization [73, 86, 132, 158]. In these works, IoT devices have some computational resources to serve their partial/complete tasks. However, they may send their requests to other surrogate servers if overall global parameters of IoT devices and systems can be optimized.

2.5.2 Objective Number

According to the number of main optimization objectives of each proposal, we classify the current literature into 1) **Single objective** and 2) **Multi objective** proposals. Multi objective proposals consider several parameters to simultaneously optimize them, incurring higher complexity. In the literature, several proposals targeted single objective optimization such as [130, 141, 150, 156], while other proposals try to optimize several parameters such as [34, 74, 123, 157].

2.5.3 Parameters

According to the main objectives and the nature of optimization parameters in the literature, we categorize these parameters into the following categories:

Time

One of the most important optimization parameters is the execution time of IoT applications. Minimizing the execution time of IoT applications provides users with a better QoS and QoE. This category contains any metrics related to time such as response time, execution time, and makespan used in the literature such as [1, 79, 88, 119].

Energy

IoT devices are usually considered as battery-limited devices. Hence, minimizing their energy consumption is one of the most important optimization parameters. Besides, energy consumption from FSs' perspective is two-fold. First, some FSs, similar to IoT devices, are battery-constrained, making optimizing the energy consumption of FSs an important challenge. Second, from the system perspective, the energy consumption of FSs should be minimized to reduce carbon emissions. This category contains any proposals considered energy consumption as an optimization parameter either from IoT devices or system perspectives such as [68, 99, 120, 154].

Monetary Cost

This category studies the proposals that have considered the monetary aspects either from IoT users (i.e., minimizing monetary cost) or system perspectives (i.e., increasing monetary profit) [115, 117, 126, 126, 159].

Other

Some works have considered other optimization parameters such as the number of served requests, system utility, and resource utilization, just to mention a few, such as

[89, 108, 160, 163].

Hybrid

Several works also have considered a set of optimization parameters, referred as hybrid. These works use any combination of above-mentioned parameters simultaneously [3, 74, 76, 161].

2.5.4 Problem Modeling

Considering the main goal and optimization parameters, the optimization problem can be modeled/formulated. Considering surveyed literature in terms of the problem modeling approach, we classify the works into the following categories:

Integer Linear Programming (ILP)

It is a problem type where the variables and constraints are all integer values, and the objective function and equations are linear. Several works have used ILP for problem modeling such as [102, 107, 117, 147].

Mixed Integer Linear Programming (MILP)

In these problems, only some of the variables are constrained to be integers, while other variables are allowed to be non-integers. Also, the objective function and equations are linear. Several works have modelled their problem as a MILP such as [73, 75, 77, 123].

Mixed Integer Non-Linear Programming (MINLP)

It refers to problems with integer and non-integer variables and non-linear functions in the objective function and/or the constraints. Several works such as [70, 84, 98, 100] have used MINLP to present their optimization problems.

Markov Decision Process (MDP)

It provides a mathematical framework to model and analyzes problems with stochastic and dynamic systems. Several works have used MDP to model scheduling problem in Fog computing such as [72, 76, 160, 161].

Other

There are also some other optimization modeling approaches in the literature of scheduling applications in Fog computing such as game theory [92], lyapunov [89, 135], and mixed integer programming [93, 151].

2.5.5 QoS Constraints

The formulated optimization problem usually contains several constraints, incurring higher complexity compared to unconstrained problems. In this work, we classify techniques based on the QoS-related constraints applied to the main formulated problem into 1) **Deadline** such as [101, 105, 112], 2) **Energy** such as [106, 114], and 3) **Hybrid** (i.e., any combination of deadline, energy, and cost) such as [74, 84, 135].

2.5.6 Discussion

In this section, we discuss the effects of identified optimization characteristics' elements on the decision engine and describe the lessons that we have learned. Besides, we identify several research gaps accordingly. Table 2.4 provides a summary of characteristics related to optimization problems in Fog computing.

Effects on the decision engine

The optimization characteristics affect the decision engine in various aspects, as briefly described below.

1. Objective number and parameters: Simultaneous optimization of multi-objective problems usually incur higher complexity for the decision engine. Also, when the number of key parameters in a multi-objective scheduling problem increases, finding the best parameters' coefficients becomes a critical yet challenging step.

2. Problem modeling: It can affect the choice of placement technique as some specific algorithms and techniques can be used to solve the scheduling problem. For example, several traditional LP and ILP tools and libraries exist to solve LP and ILP scheduling problems.

3. QoS constraints: They incur hard or soft constraints and limitations on the main objective/objectives of the scheduling problem, which intensify the complexity of the scheduling problem. The decision engine should satisfy these constraints either using classic Constraint Satisfaction Problem (CSP) techniques or using customized approaches.

Lessons learned

Our findings regarding the optimization characteristics in the surveyed works are briefly described in what follows:

1. The main perspective of optimization for almost 75% of works is IoT, while for the rest of the works is hybrid and system by 15% and 10%, respectively. The main perspective element affects how some metrics are evaluated. For example, when evaluating the energy consumption in the IoT perspective, the energy consumed by the surrogate servers for the execution of tasks is overlooked. However, in the system and hybrid perspectives, the energy consumption of all resource providers and all entities in the systems are evaluated, respectively.

2. Considering objective numbers in the optimization problem, the works are almost equally divided into single and multiple objective numbers. Overall, the majority of works studied time and/or energy as their main optimization metrics. While the works

with an IoT perspective follow the same trend for the optimization metrics, the proposals with a system perspective almost consider the cost as their main optimization parameter. Also, the hybrid perspective proposals often consider a combination of time, energy, and/or cost as their main optimization metrics.

3. In problem modeling, the greater number of works have used either MDP or MINLP (each with roughly 25% of proposals) to formulate their problem. Also, some works initially had modeled their work as MINLP and then defined the MDP accordingly, such as [84, 130]. The rest of the works have used MILP (almost 15%), ILP (almost 15 %), and other optimization modeling approaches.

4. Almost 25% of works defined single or multiple QoS constraints for their problem, among which 90% have considered a single constraint, and the rest went for two QoS constraints. Among the QoS constraints, the deadline by 90% is the most used constraint in all works.

Research Gaps

We have identified several open issues for further investigation that are discussed below:

1. The main part of works in the literature either consider optimization problems from IoT devices/users. However, only a few works have considered IoT and system perspectives simultaneously (i.e., hybrid). Optimizing either of these perspectives can negatively affect other perspectives. To illustrate, when the principal target is minimizing the energy consumption of IoT devices, the majority of components or tasks are placed at FSs or CSs. However, it may negatively affect the energy consumption of resource providers and even increase the aggregated energy consumption in the environment. Hence, further investigation on hybrid optimization perspectives and mutual effects of different perspectives is required.

2. The cooperation among the resource providers (i.e., FSs, CSs) is an essential factor in offering higher-quality services. Proposals in the system and hybrid perspectives can

also consider other metrics such as trust and privacy index for resource providers and study how they affect the overall performance.

3. QoS constraints are set to guarantee a minimum service level for end-users. In current literature, most of proposals have focused on the deadline as the constraint. However, several other parameters such as privacy, security, and monetary cost and their combination as hybrid QoS constraints are not studied.

Table 2.4: Summary of existing works considering optimization characteristics taxonomy

Ref	Optimization Characteristics				
	Main Perspective	Objective Number	Metrics	Problem Model	QoS Constraints
[72]	System	Single	Cost	MDP	○
[68]	IoT	Single	Energy	MINLP	○
[96]	IoT	Multiple	Time, Energy, Cost	IP	○
[80]	IoT	Single	Cost	MINLP	Deadline
[76]	IoT	Multiple	Time, Energy, Weighted Cost	MDP	○
[3]	IoT	Multiple	Time, Energy, Weighted Cost	MIP	○
[70]	IoT	Single	Time	MINLP	○
[77]	IoT	Single	Time	MILP	○
[97]	Hybrid	Multiple	Time, Energy, Cost	MDP	○
[98]	IoT	Single	Time	MINLP	○
[82]	IoT	Single	Time	MDP	○
[99]	IoT	Multiple	Time, Computation Ratio	MDP	○
[100]	IoT	Multiple	Time, Energy, Weighted Cost	MINLP	Deadline

[81]	IoT	Single	Time	MDP	○
[101]	Hybrid	Single	Energy	MINLP	Deadline
[102]	IoT	Single	Energy	ILP	○
[103]	IoT	Single	Time	MDP	○
[104]	IoT	Multiple	Time, Cost	Not Defined	○
[105]	Hybrid	Single	Time/Enegy	MILP	Deadline
[106]	IoT	Multiple	Time, Energy	MINLP	Energy
[107]	IoT	Multiple	Cost, Time	ILP	○
[108]	System	Single	Maximize Served Requests	MILP	○
[90]	IoT	Single	Time	MINLP	○
[87]	IoT	Single	Time	MDP	○
[91]	IoT	Single	Energy	MINLP	Time
[88]	IoT	Single	Time	LP	○
[109]	IoT	Single	Time	MINLP	○
[110]	IoT	Multiple	Time, Energy	MINLP, MDP	○
[111]	Hybrid	Multiple	Time, Enery, Cost	Not Defined	○
[112]	IoT	Single	Maximize Offloaded Task	MILP	Deadline
[113]	Hybrid	Multiple	Time, Energy	MILP	○
[89]	System	Single	Stystem Utility	Lyaponuv	○
[114]	Hybrid	Single	QoS, eg. Delay	IP	Energy
[115]	System	Single	Cost	MILP	○
[116]	IoT	Single	Cost	MILP	Deadline
[117]	System	Single	Cost	ILP	○
[118]	IoT	Single	Time	Not Defined	○
[95]	IoT	Single	Time	Not Defined	○
[119]	IoT	Single	Time	MDP	○

[120]	IoT	Single	Energy	MDP	Deadline
[121]	IoT	Single	Time	Not Defined	Deadline
[122]	IoT	Multiple	Time, Energy	ILP	○
[1]	IoT	Single	Time	Not Defined	○
[92]	IoT	Multiple	Time, Energy	Game Theory	○
[123]	IoT	Multiple	Time, Energy, Weighted Cost	MILP	○
[124]	IoT	Multiple	Time,Energy	MINLP	○
[125]	IoT	Single	QoS	ILP	Cost, Deadline
[126]	IoT	Single	Cost	Lyaponuv	Deadline
[127]	IoT	Single	Deadline Satisfaction	Not Defined	○
[36]	System	Multiple	Time, Resource	ILP	○
[128]	IoT	Single	Time	ILP	Deadline
[129]	IoT	Multiple	Time, Cost	MINLP	Deadline
[73]	Hybrid	Multiple	Energy, Time	MILP	○
[69]	IoT	Multiple	Enery, Time	MDP	○
[71]	IoT	Multiple	Energy, Time	MDP	○
[130]	IoT	Single	Energy	MINLP, MDP	○
[131]	IoT	Multiple	Energy, Time	MDP	○
[132]	Hybrid	Multiple	Time, Resource Utilization	MDP	○
[133]	Hybrid	Multiple	Time, Cost	MDP	Deadline
[78]	Hybrid	Multiple	Energy, Cost	MDP	○
[34]	IoT	Multiple	Time, Eneegy	ILP	○
[134]	IoT	Single	Time	Not Defined	○
[135]	Hybrid	Single	Time	Lyapunov	Cost, Deadline
[136]	IoT	Single	Time	Not Defined	○
[137]	Hybrid	Multiple	Time, Energy	MINLP	Deadline

[138]	IoT	Multiple	Time, Energy	MINLP	Deadline
[139]	IoT	Single	Time	MINLP	Deadline
[140]	IoT	Single	Time	MDP	○
[141]	IoT	Single	Time	MILP	Quality Loss
[83]	IoT	Single	QoS	Not Defined	○
[142]	IoT	Multiple	Not Defined	Not Defined	○
[143]	IoT	Multiple	Time, Energy	Not Defined	○
[144]	IoT	Single	Time	MDP	○
[145]	IoT	Single	Time	Not Defined	○
[146]	IoT	Single	Bandwidth	Not Defined	○
[147]	IoT	Multiple	Time, Cost, Weighted Cost	ILP	○
[148]	IoT	Not Defined	Time	Not Defined	○
[149]	IoT	Single	Time	Not Defined	○
[150]	IoT	Single	Time	Not Defined	○
[151]	IoT	Single	Time	MIP	○
[94]	IoT	Single	Time	Not Defined	○
[84]	IoT	Multiple	Time, Energy	MINLP, MDP	Deadline,Max Energy
[152]	IoT	Single	Time	Not Defined	○
[153]	System	Multiple	Time, Energy	ILP	○
[154]	System	Single	Energy	Not Defined	Deadline
[155]	IoT	Single	Time	Not Defined	○
[85]	IoT	Single	Time	Not Defined	○
[156]	IoT	Single	Time	MINLP	○
[86]	Hybrid	Multiple	Time, Cost	MINLP	○
[157]	Hybrid	Multiple	Time, Energy, Cost	Not Defined	Deadline
[32]	IoT	Single	Time	Not Defined	○

[158]	hybrid	Multiple	Time, Energy, Cost	Not Defined	Deadline
[79]	IoT	Single	Time	Not Defined	○
[75]	IoT	Multiple	Time, Energy	MILP	Deadline
[159]	IoT	single	Cost	ILP	Deadline
[160]	Hybrid	Multiple	Maximize Served Requests, Minimize Fog Number	MDP	○
[93]	IoT	Single	Time	MIP, QCQP	○
[161]	IoT	Multiple	Time, Energy, Weighted Cost	MDP	○
[74]	IoT	Multiple	Time, Energy, Weighted Cost	MDP	Deadline, Max Energy
[162]	IoT	Multiple	Cost, Energy	Not Defined	Deadline

Cost: Monetary Cost, MDP: Markov Decision Process, ILP: Integer Linear Programming, MINLP: Mixed Integer Non-Linear Programming, MIP: Mixed Integer Programming, MILP: Mixed Integer Linear Programming, ○: No

2.6 Decision Engine Characteristics

The requirements of IoT applications in Fog computing can be satisfied if incoming IoT requests can be accurately scheduled based on the characteristics of application structure, environmental architecture, and optimization problems by the decision engine. The main responsibilities of the decision engine are organizing received IoT requests and solving the optimization problem through a placement decision while considering contextual information. Fig. 2.6 presents the main elements in decision engine, as described in what follows:

2.6.1 Deployment Layer

Decision engines can be deployed on servers at different layers unless the servers do not have sufficient resources to host them. Based on the deployment layer of the decision engine, the current literature can be classified into:

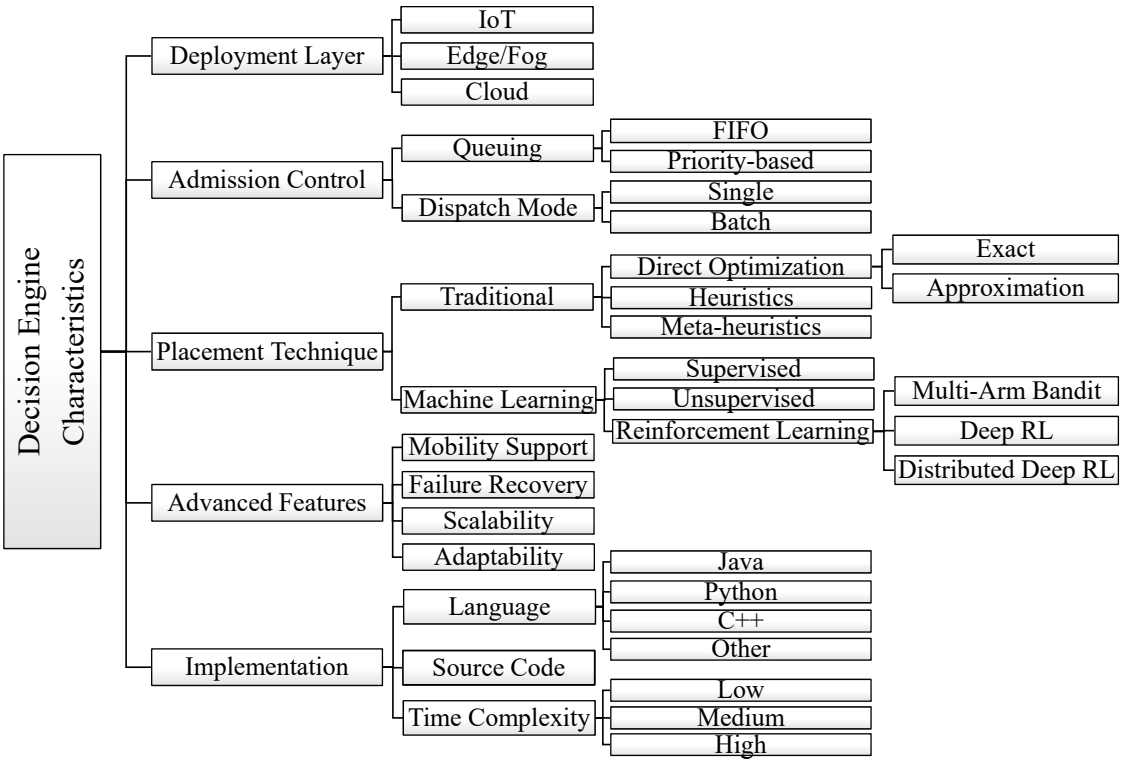


Figure 2.6: Decision engine taxonomy

IoT Layer

The IoT devices usually are considered as resource-limited and battery-constrained devices. Hence, decision engines running on IoT devices should be very lightweight even with compromising the accuracy. In the literature, several works such as [102, 103, 121, 130] deployed decision engines at the IoT layer.

Fog/Edge Layer

Distributed FSs with sufficient resources situated in the proximity of IoT devices are the main deployment targets for the decision engines. They provide low-latency and high-bandwidth access to decision engines for IoT devices. Majority of works such as [69, 76, 111, 162] deployed the decision engines in Edge/Fog Layer.

Cloud Layer

CSs are potential targets for the deployment of decision engines. Although the access latency to CSs is higher, they provide high availability, making them a suitable deployment target where FSs are not available or when IoT applications are insensitive to higher startup time. Some works such as [129, 136] considered Cloud layer for the deployment of decision engines.

2.6.2 Admission Control

The admission control presents the behavior of decision engines when new requests arrive. It denotes how the new requests are queued and organized by the dispatching module for placement.

Queuing

Decision engine may use different queuing policies when incoming IoT requests arrive. Based on queuing policy, we classify works into 1) **First-in-First-Out (FIFO)** such as [76, 103, 105] and 2) **Priority-based** where incoming requests are sorted based on their priority (e.g., deadline) [88, 112, 153].

Dispatching Mode

The dispatching module forwards requests from input queue to the placement module. Based on the selection policy of dispatching module, current literature can be classified to 1) **Single** model where only one task at a time is dispatched for placement [82, 84, 144] and 2) **Batch** model where a set of tasks are forwarded to placement module [3, 152, 158].

2.6.3 Placement Technique

Placement technique is the actual algorithm used to solve the optimization problem. Each placement algorithm has its advantages and disadvantages. Hence, it should be

carefully selected based on the properties and dynamics of applications, users, environment, and deployment layer. We classify placement techniques based on their approach to find the solution into two broad categories:

Traditional

In this approach, the programmer/designer defines the required logic of policies for the placement technique. The traditional placement technique can be further divided into three subcategories:

1. Direct Optimization: In this category, the optimization problem will be solved using classical optimization tools either using 1) **Exact** approach to find the optimal solution such as [80, 96] or 2) **Approximation** approach to find a near optimal solution such as [70, 107, 114].

2. Heuristics: These algorithms are a set of typically problem-dependent algorithmic steps to find a feasible solution for the problem. Heuristics usually scale well as their Time Complexity (TC) is low while they do not guarantee finding the optimal solution of the problem [68, 77, 108, 148].

3. Meta-heuristics: Meta-heuristics are composed of several advanced heuristics and typically are problem-independent, such as Genetic Algorithm (GA) and Simulated Annealing (SA). Although these algorithms usually perform better than heuristics, similarly they cannot guarantee to find the optimal solution. Several works such as [3, 90, 113, 147] used meta-heuristics.

Machine Learning (ML)

ML is a family of algorithms that can learn the required policies for placement techniques from historic data. ML algorithms scale reasonably well, however, they require accurate and ideally large samples of historic data. The ML-based placement techniques can be further divided into three subcategories:

1. Supervised Learning: These algorithms learn by using labeled data as their input. Type of problems are regression and classification, and some of the algorithms are linear regression and logistic regression. Some works in the current literature used supervised ML for the placement technique such as [100, 106, 124].

2. Unsupervised Learning: These algorithms are trained using unlabelled data, contrary to supervised ML, without any guidance. Some unsupervised algorithms are K–Means and fuzzy C–Means. Some works in the current literature used unsupervised ML for the placement technique such as [143, 144, 146].

3. Reinforcement Learning (RL): In these algorithms, agent/agents learn the required policy for placement technique by interaction with an uncertain and potentially complex environment. It does not require pre-defined data, and type of problems are exploitation or exploration. The current RL-based literature in scheduling IoT applications can be divided into 1) **Multi-Armed Bandit (MAB)** which are among the simplest RL problems such as [98, 104], 2) **Deep RL (DRL)** where deep learning is used in RL such as [69, 72, 103], and 3) **Distributed DRL** where several agents work collaboratively in a distributed manner for efficient learning such as [76, 84, 97].

2.6.4 Advanced Features

To fully utilize the potential of the Fog computing paradigm, several advanced features can be augmented with decision engines to capture high dynamics of this paradigm, described below:

Mobility Support

A significant number of IoT devices are moving entities (e.g., vehicles), requiring connected service through their path. So, decision engines should manage the migration process of application components and find suitable surrogate servers accordingly. Several works such as [34, 87, 114, 147] address mobility and migration management challenges alongside scheduling IoT applications.

Failure Recovery

In highly dynamic systems such as Fog computing, failure may happen due to software or hardware-related issues. So, application components faced with failure should be re-executed. Some works consider failure recovery mechanisms in their decision engines such as [3, 34, 94].

High Scalability

As a large number of IoT devices and servers exist in the Fog environment, mechanisms and algorithms used in the decision engine should be highly scalable and provide well-suited performance when the system size grows. Several works have studied the scalability feature of their techniques when the number of IoT applications and servers increases or discuss how their distributed techniques work efficiently in large systems such as [110, 118, 131].

High Adaptability

This feature ensures that the decision engine dynamically captures the contextual information (i.e., application, environment, etc), and updates the policies of placement techniques accordingly. In Fog literature, several works such as [76, 83, 133, 159] offer solutions with high adaptability.

2.6.5 Implementation

The implementation characteristics of decision engines are studied based on the following criteria:

Language

Different programming languages are used for the implementation of decision engines, while the majority have used Python [76, 117], Java [3, 136], and C++ [115, 127].

Source Code

Open-source decision engines help researchers to understand the detailed implementation specifications of each work, and minimize the reproducibility effort of decision engines, especially for comparison purposes. Some works such as [1, 120, 161] have provided the source code repository of their decision engines.

Time Complexity (TC)

TC of each placement technique presents the required time to solve the optimization problem in the worst-case scenario. It directly affects the service startup time and the decision overhead of each technique. Based on the current literature, we classify the TC into 1) **Low** the solution of optimization problem can be obtained in polynomial time where the maximum power of variable is equal or less than two (i.e., $O(n^2)$) [72, 76, 94], 2) **Medium** where the time complexity is polynomial and the maximum power is less than or equal to 3 (i.e., $O(n^3)$) [68, 75, 77], and 3) **High** for exponential TC and polynomials with high maximum power [80, 96, 112].

2.6.6 Discussion

In this section, we describe the lessons that we have learned regarding identified elements in decision engine characteristics of the current literature. Besides, we identify several research gaps accordingly. Table 2.5 provides a summary of decision engines-related characteristics in Fog computing.

Lessons learned

Our findings regarding the decision engine characteristics in the surveyed works are briefly described in what follows:

1. Almost 85% of surveyed works deployed the decision engine at the Edge layer in the proximity of IoT devices. Since the Edge servers can be accessed with lower latency and higher access bandwidth, deployment of decision engines at the Edge can reduce the

startup of IoT applications. However, the Edge devices should have sufficient resources to run the decision engine. Some proposals (about 10%) also deployed the decision engine on IoT devices. Deployment of a decision engine on IoT devices provides more control for IoT devices, especially mobile ones. It eliminates the extra overhead of communication with surrogate servers for making a decision. However, IoT devices often have very limited resources that are incapable of running powerful decision engines.

2. The queuing is an important element in the admission control that almost 80% of the works have not studied. Since most of works have considered several IoT devices in the environment, several IoT requests may arrive in each decision time-slot with a high probability. Hence, different queuing models can dramatically affect the decision engine performance and the QoS of end-users. FIFO and priority queue share the same proportion of proposals among the works that mentioned their queuing policy. Also, in priority-based queuing, almost all works have considered the deadline of applications or tasks as their main priority metric. Moreover, for the policy of dispatching module, about 75% of works selected single dispatching while 25% of works studied batch dispatching policy. Since different IoT requests may arrive in the same decision time-slot, the batch dispatching policy helps study the mutual effects of IoT applications with diverse resource requirements in the placement decision.

3. The traditional placement techniques are used in almost 60% of the proposals, while the ML-based placement techniques are studied in the rest of the works (almost 40%). However, the number of ML-based placement techniques has significantly increased in recent years. In traditional placement techniques, direct optimization, heuristics, and meta-heuristics share the same proportion of proposals. Also, in meta-heuristics techniques, the majority of works used population-based meta-heuristics, especially different variations of the GA. In the ML-based techniques, the majority of proposals have used RL-based techniques (almost 70%), specially DRL. Moreover, in the DRL techniques, the larger number of works used centralized DRL techniques such as DQN. However, the exploration and convergence rate of centralized DRL techniques are very slow. Thus, several studies have recently been conducted to adapt distributed DRL (i.e.,

DDRL) techniques for resource management in Edge/Fog computing environments, such as [76, 110, 131], to improve the exploration cost and convergence rate of the DRL techniques.

4. In advanced features, almost 25% of proposals embedded different mechanisms (i.e., traditional or ML-based) for the mobility management of IoT devices and migration of applications' constituent parts. Also, about 25% of studied works, mostly ML-based techniques, offer high adaptability features in their decision engine. However, traditional works often neglect to provide different mechanisms to support high adaptability. This is mainly because the scheduling policies are not statically defined in ML-based techniques. Hence, as the environmental or application properties change, the policies can be learned and updated accordingly. However, in the traditional scheduling techniques, updating the scheduling policies according to dynamic changes in environmental or application properties is very costly and time-consuming. Almost 20% of proposals studied different mechanisms to support high scalability feature, either using ML-based techniques or traditional approaches. In advanced features, the failure recovery mechanisms and techniques in scheduling are not well-studied and only a few works embedded these mechanisms in their scheduling techniques.

5. Considering the implementation of the techniques, almost 50% of the works mentioned their employed programming language. Java and python programming language are the most-employed programming language and are almost equally used in different proposals. However, Python is mainly used for ML-based techniques and direct optimization techniques, while Java is mostly used to implement traditional decision engines. Moreover, only about 10% of proposals shared their open-source repositories with researchers and developers among the surveyed works. Finally, about 65% of proposals discussed the TC of their works, among which almost 80% proposed decision engines with low TC while some proposals (almost 10%) went for medium TC and few works (almost 10%) proposed decision engines with high TC. The high TC proposals are among the direct optimization category of traditional approaches. While these high TC proposals cannot be currently adapted to large-scale Edge and Fog computing environ-

ments, they can find the optimal solution in small-scale problems. Hence, they can be used as a reference for the evaluation of other proposals.

Research Gaps

We have identified several open issues for further investigation that are discussed below:

1. The admission control concept in terms of different queuing, dispatching, and their mutual effect is not well studied in the current literature. Also, the greater number of works consider a single task dispatching model and overlook batch placement of applications, especially for applications with dependent tasks.

2. While traditional placement techniques (e.g., heuristics, meta-heuristics) are studied well in the literature, ML-based techniques are still in their infancy. Due to the lack of a large number of datasets, supervised and unsupervised ML have not been thoroughly considered. Also, the majority of employed RL techniques are centralized approaches, neglecting collaborative learning of multiple distributed agents for better efficiency and lower exploration costs.

3. Although all servers and devices are prone to failures, among advanced features, failure recovery mechanisms, algorithms, and their integration with the placement technique is the least-studied concept. Even the best placement techniques cannot complete their process in real-world scenarios unless a suitable failure recovery mechanism is embedded.

4. In the surveyed works, there is no proposal to study all the four identified elements in the advanced features (i.e., mobility, failure recovery, scalability, and adaptability) and describe the behavior and mutual effects of these elements on each other and decision engine.

5. Among the studied literature, none of the works has studied the privacy problem from different perspectives, such as end-users' data privacy, resource providers' privacy,

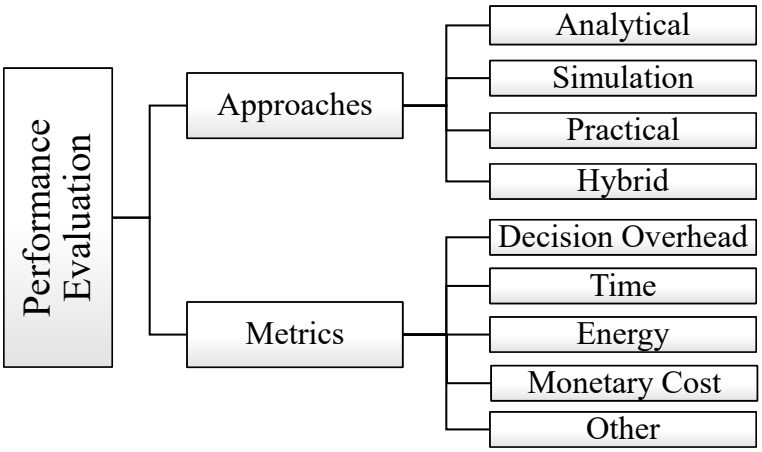


Figure 2.7: Performance evaluation taxonomy

and the decision engine's mechanisms for improving privacy.

2.7 Performance Evaluation

Different approaches and metrics have been used by the research community to evaluate the performance of their techniques. Identifying and studying these parameters helps to select the best approach and metrics for the implementation of new proposals and fair comparisons with other techniques in the literature. Fig. 2.7 presents a taxonomy and the main elements of performance evaluation, described below:

2.7.1 Approaches

The performance evaluation approaches can be divided into four categories, namely analytical, simulation, practical, and hybrid. There are different important aspects to consider when selecting an approach for the evaluation of proposals, such as credibility, implementation time, monetary cost, reproducibility time, and scalability. Fig. 2.8 presents a qualitative comparison of different approaches used in performance evaluation.

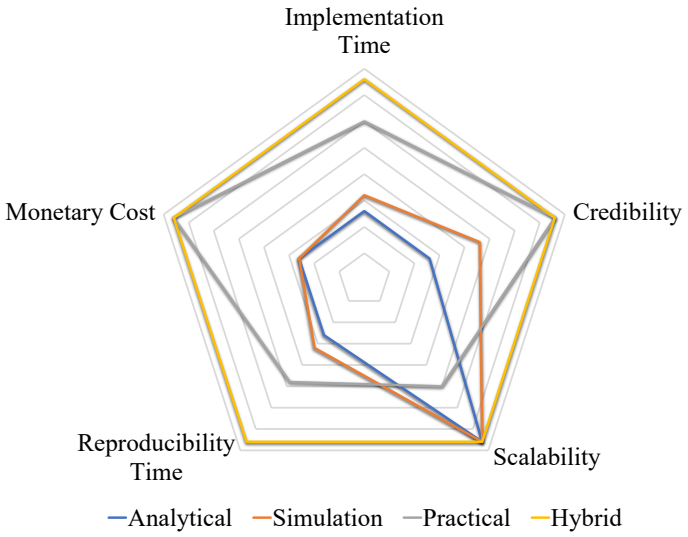


Figure 2.8: Performance evaluation approaches

Analytical

One of the popular approaches for the evaluation of different proposals is analytical tools. The implementation time, reproducibility time, and monetary cost of analytical tools are low, and scalable experiments can be executed. However, the credibility of such experiments is low since the dynamics of resources, applications, and environment cannot be fully captured and tested. Matlab is among the most popular tools that are either used directly [100, 100, 108] or integrated with some other libraries such as Sedumi[1] [93]. Also, C++ based analytical tools have been used in the literature, such as [70, 115].

[1] https://github.com/sqlp/sedumi

Table 2.5: Summary of existing works considering decision engine taxonomy

Ref	Decision Engine Characteristics										
	Deploy Layer	Admission Control		Placement Technique	Advanced Features				Implementation		
		Queuing	Dispatch		Mobility	Failure Recovery	High Scalability	High Adaptability	Language	Source Code	Time Complexity
[72]	Edge	ND	Single	ML, RL, DRL, (DQN)	○	○	○	●	ND	○	Low (MP 2)
[68]	Edge	ND	ND	Tr, H, Greedy	●	○	○	○	ND	○	Medium (MP 3)
[96]	Edge	ND	ND	Tr, DO, Exact	●	○	○	○	ND	○	High (Exp)
[80]	Edge	FIFO	Single	Tr, DO, Exact, (BB)	○	○	○	○	Java	○	High (Exp)
[76]	Edge	FIFO	Single	ML, RL, DDRL, (IMPALA)	○	○	●	●	Python	○	Low (MP 2)
[3]	Edge	FIFO	Batch	Tr, MetaH, (MA)	○	●	●	○	Java	○	Low (MP 2)

[70]	Edge	ND	Batch	Tr, DO, Approx	●	○	○	○	C++	○	ND
[77]	Edge	FIFO	Single	Tr, H	○	○	○	○	Java	○	Medium (MP 3)
[97]	Edge	ND	ND	ML, RL, DDRL, (A3C)	●	○	●	●	Java	○	Low
[98]	Edge	ND	Batch	ML, RL, MAB	○	○	○	●	ND	○	Low
[82]	Edge	ND	Single	ML, RL, DRL, (DQN)	○	○	○	●	ND	○	Low
[99]	Edge	ND	ND	ML, RL, DRL	○	○	○	●	ND	●	Low
[100]	Edge	ND	Single	ML, Sup, (DeepL)	○	○	○	○	ND	○	Low
[81]	Edge	ND	Single	ML, RL, (Q-Learning)	●	○	○	●	ND	○	Low
[101]	Edge	ND	Single	ML, Sup, (Imitation)	●	○	○	○	ND	○	Medium
[102]	IoT	ND	Single	ND	○	○	○	○	Android/Java	●	ND

[103]	IoT	FIFO	Single	ML, RL, DRL (DoubleDQN)	○	○	●	●	ND	○	Low
[104]	IoT	ND	Single	ML, RL, MAB	●	○	●	●	ND	○	Low
[105]	Edge	FIFO	Single	Tr, H	○	○	○	○	Android/Java	○	Low
[106]	Edge	ND	Single	ML, Sup, (Imitation)	○	○	○	○	ND	○	Low
[107]	Edge	ND	Single	Tr, DO, Approx	●	○	○	○	ND	○	Low (MP 2)
[108]	Edge	ND	single	Tr, H, Greedy	○	○	○	○	ND	○	High
[90]	Edge	ND	Batch	Tr, MetaH, (SA)	○	○	○	○	ND	○	ND
[87]	Edge	ND	ND	Tr, DO, Approx	●	○	○	○	ND	○	Medium
[91]	Edge	ND	Single	Tr, MetaH, (GA-PSO)	○	○	○	○	ND	○	Low (MP 2)
[88]	Edge	Priority	single	Tr, DO, Approx	○	○	●	○	ND	○	ND

[109]	Edge	ND	Single	Tr, H	○	○	○	●	ND	○	ND
[110]	Edge	ND	Single	ML, RL, DRL, DDRL	○	○	●	●	ND	○	Low
[111]	Edge	ND	Single	Tr, MetaH, (NSGA2)	●	○	○	○	Java	○	ND
[112]	Edge	Priority	Single	Tr, H	○	○	○	○	ND	○	High (MP 5)
[113]	Edge	ND	Batch	Tr, MetaH, (Ant Mating)	○	○	○	○	ND	○	ND
[89]	Edge	ND	ND	Tr, H	○	○	○	○	ND	○	ND
[114]	Edge	ND	Single	Tr, DO, Approx, (SAA)	●	○	○	○	ND	○	ND
[115]	ND	ND	Batch	Tr, H	○	○	○	○	C++	○	Low
[116]	Edge	ND	Single	Tr, DO, Approx	○	○	○	○	ND	○	Low (MP 2)
[117]	Edge	ND	Batch	Tr, DO, Approx	○	○	○	○	Python	○	Low (MP 2)
[118]	Edge	ND	Single	Tr, DO, Approx	○	○	○	●	Python	○	ND

[95]	Edge	ND	Single	Tr, DO, Approx	○	○	○	○	ND	○	ND
[119]	Edge	ND	Single	ML, RL, DRL, (PPO)	○	○	○	●	Python	○	Low
[120]	Hybrid	ND	Single	ML, RL, DDRL	○	○	●	●	Python	●	Low
[121]	IoT	ND	Single	Tr, DO, Approx	○	○	○	○	ND	○	ND
[122]	Edge	ND	Single	Tr, Other, (Min-cut)	○	○	○	○	Java	●	Low (MP 2)
[1]	Edge	FIFO	Single	Tr, MetaH, (GA)	○	○	●	●	Python	●	Low
[92]	Edge	ND	Single	Tr, DO, Approx	○	○	○	○	ND	ND	ND
[123]	Edge	ND	Batch	Tr, MetaH, (NSGA3)	○	○	○	○	Java	○	ND
[124]	IoT	ND	Single	ML, Sup, DDeepL	○	○	●	○	Python	●	Low
[125]	Edge	ND	Single	Tr, DO, Exact	○	○	○	○	Java	○	high

[126]	Edge	ND	Single	Tr, DO, Approx	○	○	○	○	Python	○	ND
[127]	Edge	ND	Single	Tr, H	○	○	○	○	C++	○	Low
[36]	Edge	ND	Single	Tr, H	○	○	○	○	Java	○	ND
[128]	Edge	ND	Single	Tr, H	○	○	○	○	Java	○	Low
[129]	Cloud	ND	Single	Tr, DO, Approx	○	○	○	○	ND	○	ND
[73]	IoT	ND	Single	ML, Sup, (DDeepL)	○	○	●	○	Python	○	Low
[69]	Edge	ND	Single	ML, RL, DRL, (DQN)	○	○	○	●	ND	○	Low
[71]	Edge	FIFO	single	ML, RL, DRL, (DoubleDQN)	○	○	○	●	ND	○	Low
[130]	IoT	ND	Batch	ML, RL, DRL, (DQN)	○	○	○	●	ND	○	Low
[131]	Edge	ND	Single	ML, RL, DDRL, (D3PG)	○	○	●	●	ND	○	Low

[132]	Edge	FIFO	Single	ML, RL, DRL, (DQN)	○	○	○	●	Python	○	Low
[133]	Edge	ND	Single	ML, RL, DRL, (DoubleDQN)	○	○	○	●	Python	○	Low
[78]	ND	ND	Single	ML, RL, DRL, (DQN)	○	○	○	●	Java	○	Low
[34]	Edge	FIFO	Single	Tr, H	●	●	○	○	Java	●	Low (MP 2)
[134]	Edge	ND	Single	Tr, H	●	○	○	○	ND	○	ND
[135]	Edge	ND	Single	Tr, DO, Approx	●	○	○	○	Java	○	ND
[136]	Cloud	ND	Single	ML, Sup, (Gradient Tree Boosting)	●	○	○	○	Java	●	Low
[137]	Edge	ND	Single	Tr, MetaH, (GA)	●	○	○	○	ND	○	Low (MP 2)
[138]	ND	ND	Batch	Tr, MetaH, (GA)	●	○	○	○	ND	○	ND

[139]	IoT	FIFO	Batch	Tr, DO, Approx	●	○	●	○	ND	○	Low
[140]	IoT	ND	Batch	ML, RL, DDRL, (A3C)	●	○	●	●	ND	○	ND
[141]	Edge	ND	Batch	Tr, DO, Approx	●	○	○	○	ND	○	ND
[83]	Edge	ND	Single	ML, RL, DRL, (DQN)	●	○	○	●	Python	○	Low
[142]	Edge	ND	Single	ML, Sup, (Regression Tree)	○	○	○	○	Java	○	ND
[143]	ND	ND	Single	ML, Unsup, (AHP)	○	○	○	○	ND	○	ND
[144]	Edge	ND	Single	ML, Unsup, (Baum-Welch Algorithm)	●	○	○	○	ND	○	ND
[145]	Edge	ND	Batch	ML, Unsup, (K-means)	○	○	○	○	ND	○	ND
[146]	ND	ND	Single	ML, Unsup, (K-means)	○	○	○	○	ND	○	ND

[147]	Edge	ND	Single	Tr, MetaH, (Tabu)	●	○	○	○	ND	○	ND
[148]	Edge	Priority	Single	Tr, H, (Spring Algorithm)	○	○	○	○	ND	○	ND
[149]	Edge	ND	Batch	Tr, MetaH, (ACO)	○	○	○	○	ND	○	ND
[150]	Edge	ND	Batch	Tr, MetaH, (MA)	●	○	○	○	ND	○	Low (MP 2)
[151]	Edge	Priority	Batch	Tr, MetaH, (GA)	○	○	○	●	ND	○	ND
[94]	Edge	Priority	Sibgle	Tr, H	○	●	○	○	Python	○	Low (MP 2)
[84]	IoT	ND	Single	ML, RL, DDRL, (DDPG)	○	○	●	●	Python	○	Low
[152]	Edge	ND	Batch	Tr, MetaH, (AEO)	○	○	○	○	ND	○	ND
[153]	Edge	Priority	Batch	Tr, MetaH, (GA)	○	○	○	○	ND	○	ND
[154]	Edge	ND	Single	Tr, H	○	○	○	○	ND	○	Low (MP 2)

[155]	Edge	Priority	Batch	Tr, MetaH, (PSO)	○	○	○	○	ND	○	Low
[85]	Edge	ND	Single	Tr, MetaH, (ACO)	○	○	○	○	ND	○	Low
[156]	Edge	ND	Batch	Tr, MetaH	○	○	○	○	Python	○	Low (MP 2)
[86]	Edge	ND	Batch	Tr, MetaH, (GA)	○	○	○	○	Python	○	Low
[157]	Edge	ND	Batch	Tr, MetaH, (GA)	○	○	○	○	Python	○	Low
[32]	Edge	FIFO	Single	Tr, H	○	○	○	○	Java	○	Medium (MP of 3)
[158]	Edge	ND	Batch	Tr, MetaH, (SSA)	○	○	○	○	Python	●	ND
[79]	Edge	Priority	Single	Tr, H	○	○	○	○	Java	○	ND
[75]	IoT	ND	Single	Tr, MetaH, (GA)	○	○	●	○	ND	○	Medium (MP 3)
[159]	Edge	Priority	Single	Tr, H, Greedy	○	○	●	○	Go	●	Low

[160]	Edge	Priority	Single	ML, RL, DRL, (DQN)	○	○	○	●	Python	○	Low
[93]	IoT	ND	Batch	Tr, DO, Approx	○	○	●	○	ND	○	Low
[161]	IoT	ND	Single	ML, RL, DRL	○	○	●	●	Python	●	Low
[74]	Edge	ND	Single	ML, RL, DRL, DQN	○	○	○	●	Python	○	ND
[162]	Edge	ND	Single	Tr, MetaH, (SPEA)	○	○	○	○	ND	○	Low (MP 2)

ND: Not Defined, ML: Machine Learning, Tr: Traditional, RL, Reinforcement Learning, DRL: Deep Reinforcement Learning, DDRL: Distributed Deep Reinforcement Learning, MP: Max Power, H: Heuristics, MetaH: Metaheuristics, DO: Direct Optimization, BB: Branch and Bound, MA: Memetic Algorithm, FIFO: First-In-First-Out, Approx: Approximation, MAB: Multi-Arm Bandit, A3C: Asynchronous Actor-Critic Agents, DeepL: Deep Learning, DDeepL: Distributed Deep Learning, Imitation: Imitation Learning, SA: Simulated Annealing, GA: Genetic Algorithm, SAA: Sample Average Approximation, PPO: Proximal Policy Optimization, D3PG: Double-Dueling-Deterministic Policy Gradients, AHP: Analytic Hierarchy Process, Sup: Supervised, Unsup: Unsupervised, ACO: Ant Colony Optimization, AEO: Artificial Ecosystem-based Optimization, Tabu: Tabu Search, PSO: Particle Swarm Optimization, SSA: Sparrow Search Algorithm, SPEA: Strength Pareto Evolutionary Algorithms

Simulation

Simulators keep the advantages of analytical tools while improving the credibility of evaluations by simulating the dynamics of resources, applications, and environments. In the literature, iFogSim [26, 27] is among the most popular simulators for Fog computing [3, 78, 97, 111]. Besides, several researchers have used Cloudsim [165] such as [80, 123] or SimPy[2] such as [126, 133] to simulate their scenarios in Fog computing.

Practical

The most credible approach for the evaluation of proposals is practical implementation. However, due to high monetary cost, implementation time, and reproducibility time, it is not the most efficient approach for different scenarios, especially evaluations requiring high scalability. In the literature, few works such as [105, 114, 136, 146] evaluated their proposals using small-scale practical implementations.

Hybrid

In this approach, researchers evaluate their proposals using practical implementations in small-scale and simulators or analytical tools in large-scale. Although implementation and reproducibility time of this approach is high, it provides high scalability and credibility. In the literature, few works such as [76, 117, 120] follow the hybrid approach.

2.7.2 Metrics

The metrics used in performance evaluation in Fog computing are directly or indirectly related to the optimization parameters and system properties. Based on the nature and popularity of these metrics in the literature, we categorize them into 1) **Time** (e.g., deadline, response time, execution time, makespan) [1, 69, 77, 80], 2) **Energy** (e.g., battery percentage, saved energy) [68, 71, 73, 73], 3) **Monetary cost** (e.g., service cost, switching cost) [72, 126, 135, 147], and 4) **Other** metrics (e.g., number of interrupted tasks, resource

[2]https://simpy.readthedocs.io/en/latest/

utilization, throughput, deadline miss ratio) [34, 92, 94, 148]. Also, we consider 5) **Decision overhead** as an important evaluation metric to study the overhead of proposals (often in terms of time and energy), used in some works such as [76, 111, 122, 162].

2.7.3 Discussion

In this section, we describe the lessons that we have learned regarding identified elements in the performance evaluation of the current literature. Besides, we identify several research gaps accordingly. Table 2.6 provides a summary of characteristics related to performance evaluation in Fog computing.

Lessons learned

Our findings regarding the performance evaluation in the surveyed works are briefly described in what follows:

1. More than half of the works used the simulation as their performance evaluation approach while 30% of the proposals used an analytical approach. The practical and hybrid approaches equally share the rest of 20% of the works. For the analytical approach, the most of works used Matlab or Python programming languages, while Java and Python are mostly used for the simulation approach. In practical and hybrid approaches, Java and Python are equally employed in proposals.

2. As the performance evaluation metric, time and its variations (e.g., response time, makespan) are used in more than 80% of the works. The second-highest-used metric is energy at 35%. However, the decision overhead and cost are only studied in 15% of the works. Besides, less than 5% of the proposals studied the performance of their scheduling technique using all the identified metrics.

Research Gaps

We have identified several open issues for further investigation that are discussed below:

1. Although the monetary costs of sensors and edge devices (e.g., Rpi, Jetson Platform) have reduced and they are highly available in different configurations, compared to a few years ago, the majority of proposals still consider analytical tools and simulators as their only approach for performance evaluation. While some works have considered practical and hybrid approaches for the performance evaluation of their work, further efforts are required to study the dynamics of the system, resource contention, and collaborative execution of the application in real environments, especially considering new machine learning techniques such as DRL and DDRL [76, 120].

2. The decision overhead of proposals has direct effects on users and resources in terms of the startup time of requested services and resource utilization. To illustrate, not only do healthcare applications require low response time, but they also need low startup time, especially for critical applications such as emergency-related applications (e.g., heart-attack prediction and detection). Also, the overhead of proposals can severely affect the resource usage and energy consumption of servers, especially battery constrained ones. Among the techniques considered decision overhead as a metric, they mostly focus on time while other metrics (e.g., energy, cost) need further investigation.

Table 2.6: Summary of existing works considering performance evaluation taxonomy

Ref	Performance Evaluation					
	Approach	Decision Overhead	Time	Energy	Cost	Other
[72]	Simulation (OPNET)	○	○	○	●	○
[68]	ND	○	○	●	○	○
[96]	Analytical	○	●	●	●	○
[80]	Simulation (Cloudsim)	○	●	○	●	○
[76]	Hybrid (Simulation+Practical)	●	●	●	○	Weighted Cost
[3]	Simulation (iFogSim)	●	●	●	○	Weighted Cost

[70]	Analytical	○	●	○	○	Resource Utilization
[77]	Simulation (Cloudsim)	○	●	○	○	○
[97]	Simulation (iFogSim)	●	●	●	●	○
[98]	Simulation	○	●	○	○	○
[82]	Simulation	○	●	○	○	○
[99]	Simulation	○	●	○	○	Computation ratio
[100]	Analytical (Matlab)	○	○	○	○	Weighted System Cost
[81]	Analytical (Matlab)	○	●	○	○	Migration cost
[101]	Analytical	○	●	●	○	○
[102]	Hybrid (Simulation+Practical)	●	●	●	○	○
[103]	Simulation	○	●	○	○	Dropped Tasks
[104]	Simulation	●	●	○	○	Swithing Cost
[105]	Practical	○	●	●	○	Computation Throughput
[106]	Simulation	○	●	○	○	○
[107]	Analytical	○	○	○	○	Weighted Cost
[108]	Analytical (Matlab)	○	○	○	○	Satisfied Requests
[90]	Hybrid (Simulation+Practical)	●	●	○	○	○
[87]	Analytical (Matlab)	○	●	○	○	○
[91]	Analytical	○	○	●	○	○
[88]	Analytical	○	●	○	○	○
[109]	Simulation	○	●	○	○	○
[110]	Simulation	○	●	○	○	Weighted Cost
[111]	Hybrid (Simulation (iFogSim)+Practical)	●	●	●	●	○
[112]	Analytical	○	●	○	○	Average Offloaded Tasks
[113]	Analytical (Matlab)	○	●	●	○	○
[89]	Analytical	○	○	○	○	System Utility
[114]	Practical	○	●	○	○	○
[115]	Analytical	○	●	○	●	○

[116]	Practical	○	○	○	●	○
[117]	Hybrid (Simulation +Practical)	○	○	○	●	○
[118]	Practical	○	●	○	○	○
[95]	Hybrid (Simulation +Practical)	○	●	○	○	○
[119]	Simulation	○	●	○	○	○
[120]	Hybrid (Simulation +Practical)	○	○	●	○	○
[121]	Hybrid (Simulation +Practical)	○	●	○	○	○
[122]	Analytic	●	●	●	○	○
[1]	Practical	○	●	○	○	Startup Time, Ram Usage
[92]	Analytical	○	○	○	○	Performance Gain
[123]	Simulation (Cloudsim)	○	●	●	○	○
[124]	Analytical	○	○	○	○	Weighted Reward
[125]	Simulation (iFogSim)	●	●	○	○	Resource gain, QoS
[126]	Simulation (SimPy)	○	●	○	●	Application Loss
[127]	Analytical	○	○	○	●	Deadline Miss Ratio
[36]	Simulation (iFogSim)	○	●	○	○	Deadline Miss Ratio
[128]	Hybrid (Simulation +Practical(iFogSim))	○	●	○	○	Resource Overhead
[129]	Analytical	○	○	○	○	Weighted Cost
[73]	Simulation	●	●	●	○	○
[69]	Simulation	○	●	●	○	Task Drop Rate
[71]	Simulation	○	●	●	○	Task Drop Rate
[130]	Simulation	○	○	●	○	○
[131]	Simulation	○	●	●	○	Task Success Rate
[132]	Simulation	○	●	○	○	Resource Utilization
[133]	Simulation (SimPy)	○	●	●	●	○
[78]	Simulation (iFogSim)	○	●	●	●	Network Usage, weighted score

[34]	Simulation (iFogSim)	●	●	●	○	Weighted Cost, Total Interrupted Tasks, Number of Migrations
[134]	simulation	○	●	○	○	○
[135]	Simulation (One Simulator)	●	●	○	●	○
[136]	Practical	○	●	○	○	○
[137]	Simulation	○	○	○	○	Weighted Cost
[138]	Analytic	○	○	○	○	Weighted Cost, Total Offloaded and Migrated tasks
[139]	Analytic	○	●	○	○	○
[140]	Simulation	○	●	○	○	○
[141]	Simulation	○	○	○	○	○
[83]	Simulation	○	○	○	○	QoS
[142]	Simulation (Cloudsim)	○	●	●	○	○
[143]	Analytical (Matlab)	○	●	●	○	Offloaded Tasks, Failed Tasks, Server Load
[144]	Simulation	○	●	○	○	Amount of Finished Tasks
[145]	Analytical (Matlab)	○	●	○	○	○
[146]	Practical	○	●	○	○	Bandwidth
[147]	Simulation	●	●	○	●	Resource Usage
[148]	Analytical (Matlab)	○	●	●	○	Failed Transmission
[149]	Analytical (Matlab)	○	●	○	○	○
[150]	Simulation (Mininet and Sumo)	○	●	○	○	○
[151]	Simulation	○	●	○	○	○
[94]	Simulation (EdgeSimDAG)	○	●	●	○	Success rate, Utilization
[84]	Simulation	○	●	●	○	Weighted Cost
[152]	Analytical (Matlab)	○	●	○	○	Throughput

[153]	Analytical (Matlab)	○	●	●	○	Weighted Cost
[154]	Analytical (Matlab)	○	●	●	○	○
[155]	ND	○	●	○	○	Missed Deadline
[85]	Analytical (Matlab)	○	●	○	○	○
[156]	Analytical	○	●	○	○	Resource Utilization
[86]	simulation	○	●	○	●	Availability
[157]	Practical	○	●	●	●	Utilization
[32]	Simulation (iFogSim)	●	●	○	○	Network Usage
[158]	Simulation	○	○	○	○	Utility Function
[79]	Simulation (iFogSim)	○	●	○	○	Network Usage
[75]	ND	○	●	●	○	System Gain
[159]	Practical	○	●	○	○	Deployed Instances
[160]	simulation	○	●	○	○	Weighted Cost
[93]	Analytical (Matlab)	○	●	○	○	Throughput
[161]	Simulation	○	●	●	○	Weighted Cost
[74]	Simulation	○	●	●	○	Weighted Cost
[162]	Simulation (FogWorkflowSim)	●	●	●	●	○

2.8 Scheduling Technique: Important Design Options

In this section, we discuss the real-world characteristics of application structure and environmental architecture and accordingly present several guidelines for designing a scheduling technique.

1. The number of IoT applications is constantly increasing in different application domains. The majority of these applications are defined as a set of dependent modules/services [95]. Besides, sharing and reusing modules/services for faster development and better management of applications is of paramount importance. Moreover, dependent IoT applications are usually modeled as a graph of tasks and their respective invoca-

tions. In this case, IoT applications with monolithic and independent design can also be defined as an application graph with only one module and an application graph with several modules where the size of invocations is zero, respectively. Hence, we consider IoT applications with dependent modules/services (i.e., modular and loosely-coupled categories) as the main architectural design choices in the application structure. Accordingly, the decision engine requires a component for identifying and satisfying the constraint among modules/services.

2. Besides, in a real-world scenario, application modules have different characteristics (e.g., computation size, input size, ram usage). Thus, the best assumption for application modules is applications with heterogeneous granularity specifications. As the number of contributing parameters and the dynamicity of the application elements increases, capturing the application parameters with temporal patterns for efficient scheduling decisions becomes more complex [76, 97]. Although traditional-based placement techniques (e.g., heuristic, meta-heuristic) often work well in general scenarios, they fail to adapt to continuous changes and dynamic contexts. ML-based decision engines, such as RL, can more efficiently work in a dynamic context and provide higher adaptability.

3. In large-scale Fog computing environments, numerous IoT applications with different workload models and hybrid CCR may exist. Hence, the decision engine requires an admission control component with an appropriate queuing mechanism (based on application requirements) to manage diverse incoming requests and prioritize them for making the decision.

4. Regarding the environmental architecture, the most generalized scenario is when the environment consists of several heterogeneous IoT devices, several heterogeneous FSs, and multiple heterogeneous CSs. Also, the required mechanisms for intra-tier and inter-tier cooperation among servers should be embedded to support diverse IoT application scenarios, such as mobility. Besides, multiple distributed servers can collaboratively provide better performance for the execution of IoT applications. Moreover, different fault domains can be prepared to improve the availability of services. However, as the number of IoT applications and available servers in the environment increase,

the complexity of making decisions increases. Hence, the optimal scheduling decision cannot be obtained in a timely manner. Consequently, other placement techniques such as heuristics and ML-based techniques should be employed to obtain the scheduling decision in a reasonable time.

5. The decision engine can be implemented as a set of distributed services/microservices. A decision engine developed as a monolithic application may not be able to be deployed on a single server, especially on resource-limited FSs. Hence, distributed deployment of decision engine components on several distributed servers can provide several benefits: 1) more efficient deployment of resource-limited devices, 2) provides better fault tolerance 3) offers better scalability 4) support different deployment models (e.g., deployment of decision engine on FSs, CSs, or hybrid on both FSs and CSs). Hybrid deployment of decision engine components on both FSs and CSs can lead to a better user experience for end-users. To illustrate, applications requiring low latency and startup time can be managed at the low-level FSs (i.e., at the Edge), and then be scheduled based on the decision engine deployed at the Edge. However, application requests that are insensitive to latency or startup time can be forwarded to CSs for scheduling.

6. Regardless of application and environmental characteristics, failure recovery mechanisms and policies should be integrated into any decision engine. Independent failures and the non-deterministic nature of any components (either hardware or software) in distributed systems cause the most impactful issues in distributed systems. If the decision engine, which manages the scheduling and execution of incoming IoT application requests, does not have an appropriate failure recovery mechanism, the smooth execution of the whole system stalls.

2.9 Summary

In this chapter, we mainly focused on scheduling IoT applications in Fog computing environments. We identified several main perspectives playing an important role in scheduling IoT applications, namely application structure, environmental architecture,

optimization characteristics, decision engines properties, and performance evaluation. Next, we separately identified and discussed the main elements of each perspective and provided a taxonomy and research gaps in the recent literature.

Chapter 3

Fog-Driven Network Resource Allocation

Allocation of network resources in highly-dense Edge and Fog computing environment becomes more crucial when ever-increasing IoT applications with different network resource requirements forward their requests to Edge and Fog servers for processing and storage. In this chapter, a hierarchical technique, consisting of a dynamic distributed clustering and a Fog-driven resource allocation, to optimize the total throughput of the network while mitigating the interference is proposed. The fully distributed clustering method is designed so that Edge and Fog servers adaptively form clusters with dynamic size based on the current status of the network and end-users. Moreover, a policy-aware resource allocation method is proposed to address the intra and inter-cluster interference, which are two potential types of interference in clustering-based resource allocation techniques. The extensive simulation results demonstrate that our proposed hierarchical technique significantly improves total throughput, user satisfaction, and fairness compared to other proposed techniques by up to 21%, 97%, and 10%, respectively.

3.1 Introduction

A rapid growth in deployment and the use of IoT devices such as smartphones, tablets, and sensors has resulted in rapid increase of data-streaming applications such as video streaming, online games, health-care, and Voice over Internet Protocol (VoIP). This leads to a significant amount of data to be transferred over cellular networks [2, 28, 29]. Con-

This chapter is derived from:

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Fog-driven Dynamic Resource Allocation Technique in Ultra Dense Femtocell Networks", *Journal of Network and Computer Applications (JNCA)*, Volume 145, ISSN: 1084-8045, Elsevier Press, Amsterdam, Netherlands, November 2019.

sidering that the number of cellular network resources is restricted, the requested QoS can be satisfied for only a limited number of users. Besides, recent studies have revealed that approximately 70 percent of the data is originated from indoor places where severe wall penetration loss and longer transmission distance incur poor received signal quality [166]. To address these issues, Femtocell Base Stations (FBSs), which are low-power, short-range, low-cost, and low-level Edge devices are deployed over macrocell network to effectively improve the indoor received signal quality and overall network throughput. This latter is obtained by reusing same frequency by several FBSs while the former one is satisfied by decreasing the distance between transmitter and end-users [167].

However, in densely deployed femtocell networks, neighboring FBSs experience severe co-tier interference (i.e., interference between adjacent femtocells [168]) due to finite domain of shared spectrum unless an efficient interference management technique is used. The co-tier interference can be significantly reduced in downlink Orthogonal Frequency Division Multiple Access (OFDMA)-based femtocell networks using an efficient allocation of Resource Blocks (RBs) between interfering FBSs [169]. To achieve this, researchers have proposed several Resource Allocation (RA) techniques, including centralized and clustering. However, due to the non-convex non-deterministic polynomial-time (NP-hard) nature of this problem, centralized techniques are not efficiently practical and result in high complexity, signaling overhead, and single point of failure, specifically in dense and ultra-dense networks [167, 170]. To overcome this problem, the clustering-based RA techniques, which are partially decentralized, are introduced by which the complexity of the RA problem is significantly reduced. In the majority of these techniques, each cluster has access to the entire set of RBs, while FBSs in one cluster cannot use the same RBs simultaneously. This latter enables the RA technique to be performed in each cluster independently of other clusters [28].

To effectively utilize the benefits of clustering in RA, several issues should be carefully addressed. Clusters can be formed either by the gateway (GW) centrally or by FBSs in a distributed manner [171]. Moreover, the maximum size and number of clusters can be statically determined or can be obtained dynamically by the GW or cluster heads (CHs) at the runtime. In addition, the RA in each cluster can be performed by a CH individually or all FBSs collaboratively. Besides, in the dense and ultra-dense fem-

tocell networks, interference between clusters should be mitigated so that FBSs located at the edge of clusters (Edge FBSs) do not suffer from decreased throughput, which reduces total throughput and end-users' quality of experience. Last but not least, it is worth mentioning that centralized and clustering-based RA techniques, in which GWs and CHs respectively perform the majority of responsibilities, suffer from the scalability issues in dense and ultra-dense networks, because the above-mentioned burdens are not proportionally distributed.

Taking cognizance of these issues, we propose a **D**istributed **D**ynamic **C**lustering (D^2C)-**FO**g-driven **R**esource **A**llocation **T**echnique (D^2C-FORAT) to optimize the total throughput of the downlink OFDMA-based Edge networks, specifically femtocells. The proposed solution is divided into two methods, including distributed dynamic clustering and RA, so that we proportionally distribute responsibilities over the network entities, containing FBSs, CHs, GW, and local Fog servers. In the D^2C-FORAT, FBSs make clusters in a distributed dynamic manner so that FBSs with the highest co-tier interference on each other can join the same cluster and select a CH. Afterward, the CH monitors the available resources and users' demands in its cluster and dynamically controls the size of its cluster in the runtime. In addition, the Fog servers collect the Edge FBSs' information of each cluster which will be used to form the Edge FBSs' interference graph. Besides, the Fog servers employ a graph-coloring-based technique to assign a set of policies for Edge FBSs in each cluster to reduce inter-cluster interference. These policies are then forwarded to respective CHs, by which the RA can be performed more efficiently, resulting in increased throughput and user satisfaction.

The major **contributions** of this chapter are:

- Proposes a hierarchical RA technique, aiming at maximizing the total throughput while mitigating the interference, to satisfy the ever-increasing users' demands in dense and ultra-dense Edge and Fog computing environments.

- Puts forward a distributed dynamic clustering algorithm by which CHs adaptively control their cluster size based on the requested demands of their end-users. This results in better scalability so that our technique can be effectively adapted to dense and ultra-dense Edge and Fog computing environments.

- Because sufficient resources are available in each cluster due to the proposed clustering method, no intra-cluster interference occurs. To address the inter-cluster interference problem, it develops a Fog-driven RA method by which the Fog servers assign a set of policies to CHs to be considered in their RA. This latter leads to decreasing inter-cluster interference, which significantly improves the total throughput and user satisfaction.

The rest of this chapter is organized as follows: after discussing related work in Section 3.2, the system model and problem formulations are presented in Section 3.3. Proposed distributed clustering and RA methods are presented in Section 3.4 and Section 3.5, respectively. Section 3.6 evaluates the performance of the proposed policy in respect to existing policies. Finally, Section 3.7 concludes the chapter.

3.2 Related Work

A significant number of studies has been focused on RA techniques in OFDMA-based femtocell networks to address the co-tier interference, among which we study the current literature in clustering-based RA. The proposed techniques are categorized into two groups of centralized and distributed based on their clustering approach. Besides, the main elements of each technique are identified, by which we can qualitatively compare these techniques.

In the centralized clustering techniques, clustering is performed by the GW. The authors in [172] proposed a centralized clustering technique for the RA in femtocell networks, in which, after the formation of the interference graph, the GW obtains the minimum-interfered clusters by Max k-Cut algorithm. Then, a heuristic algorithm is used to assign available RBs to different clusters. Authors in [173] proposed a hierarchical RA technique, in which the GW collects the channel gain between each pair of FBSs and builds the interference graph. The GW forms clusters based on the correlation clustering concept. Afterward, due to the NP-hard nature of correlation clustering, the problem is formulated as a semi-definite program (SDP) and solved by randomized rounding. Authors in [174] proposed a cluster-based solution for the RA, in which the FBSs are clustered together according to their geographical positions by the GW. Then,

in each cluster, the FBS with the largest interference degree is selected as the CH, whose main task is the RA in its cluster. This latter is performed by solving an optimization problem via a sub-gradient iteration-based RA algorithm. In [167], the GW collects the interference degree of each FBS and performs a predetermined clustering accordingly. Then, for each cluster derived in the predetermined clustering, the GW specifies a set of best candidate sub-clusters. This process is repeated until the GW recognizes the best sub-clustering for each cluster. The authors in [175] proposed a RA technique in which GW centrally clusters FBSs by a modified k-mean clustering algorithm. Afterward, a greedy algorithm is used to distribute available resources to the FBSs.

Although the main goal of centralized clustering techniques is to find the best clustering, this is a time-consuming process, specifically in the dense and ultra-dense femtocell networks [171]. Moreover, considering the dynamic nature of femtocell networks, the GW requires to re-cluster all the FBSs whenever any change occurs in the network to find the best clustering solution.

In the distributed clustering techniques, FBSs collaboratively make clusters without the participation of the GW. If any changes occur in the status of the network and end-users, the change can be handled locally, and there is no need to repeat the clustering algorithm for all FBSs. Authors of [176] proposed a graph-based RA technique in which clusters of non-interfering FBSs are formed in a distributed manner by FBSs, and then, the GW performs the RA for each cluster according to its average users' demands. Since the RA is performed centrally by the GW, this technique cannot be efficiently employed in dense and ultra-dense networks. Moreover, RBs are not fairly allocated to end-users because this technique only considers the average demand of each cluster. However, each user's demand can be more than the obtained cluster's average demand, resulting in less fairness and user satisfaction. In [177], the authors proposed a Quality-of-service-based Femtocell Cluster-based RA (QFCRA). Initially, each FBS builds a neighboring list containing its one-hop FBSs and sends it to all of its proximate neighbors. This latter assists each FBS to obtain the interference degree of its one-hop neighbors. Afterward, the FBS with the largest interference degree among its neighbors announces itself as the CH, and other FBSs connect themselves to it. Each CH has the responsibility of RA, especially in the dense and ultra-dense femtocell networks. In [171], the

authors proposed a learning-based scheme (LFCRA) to solve the inter-cluster interference of the QFCRA. Although the proposed technique is more efficient than QFCRA in handling inter-cluster interference, similar to QFCRA, its performance mainly depends on its cluster size, which is not dynamically set.

3.2.1 A Qualitative Comparison

Table 3.1 identifies and compares key elements of current works in terms of architectural parameters, clustering parameters, interference management, and evaluation parameters. The clustering parameters include an approach describing whether that proposal is centralized or distributed, dynamicity showing whether that cluster updates itself whenever a new FBS is added or removed, or even when the users' demands are changed. The cluster size can be set by GW or CHs in a predefined manner statically or according to the current status of FBSs dynamically. The clustering criteria show the main factors used for the creation of clusters, which can be interference degree, interference intensity, and users' demands. The architectural parameters contain hierarchical components which describe what entities participate in clustering and RA and introduce specific roles of each entity. Furthermore, although all proposals consider intra-cluster interference management, only some of them address the inter-cluster interference which has a significant effect on network throughput, specifically in the densely deployed edge networks, specifically femtocell. Finally, the evaluation parameters depict the main parameters by which the performance of each proposal is evaluated.

To address the above-mentioned issues, we propose a hierarchical technique, called D^2C-FORAT, containing clustering and RA methods. The clustering is performed in a distributed manner, where FBSs collaboratively form clusters based on interference intensity, and cluster members (CMs) identify a CH for each cluster. Each CH dynamically controls the size of its cluster in the runtime based on available resources and interference intensity. Besides, CHs, based on a proposed routine, dynamically update the cluster parameters whenever users' demands change or an FBS sends a join/disjoin message to the CHs. In addition, the RA method concurrently addresses the intra and inter-cluster interference problem. Since each CH guarantees that there are always suffi-

Table 3.1: A qualitative comparison of related works with ours

Techniques	Clustering Parameters				Architectural Parameters				Interference Management		Evaluation Parameters			
	Approach	Dynamicity	Size	Criteria	Hierarchical Components	GW Role	CH Role	CM Role	Intra Cluster	Inter Cluster	Throughput	Interference	Fairness	Satisfaction
[172]	Centralized	D	D	Intf I	GW, CM	Clustering, RA	-	-	✓	×	✓	×	×	×
[173]		S	D	Intf I	GW, CH, CM	Initial Clustering, CH Sel	RA, Improve Clustering	-	✓	×	✓	✓	×	×
[174]		S	S	Intf D	GW, CH, CM	Clustering	RA, PA	CH Sel	✓	×	✓	×	×	×
[167]		S	S	Intf D	GW, CH, CM	Initial Clustering, CH Sel	RA, PA	-	✓	✓	✓	×	✓	×
[175]		S	D	Intf D	GW, CH, CM	Clustering, CH Sel	RA	-	✓	×	✓	×	×	✓
[176]	Distributed	S	S	Intf D	GW, CM	RA	-	-	✓	×	×	×	✓	×
[177]		S	S	Intf D	CH, CM	-	RA	CH Sel	✓	✓	×	×	✓	✓
[171]		S	S	Intf D	CH, CM	-	RA, Power Adj	CH Sel	✓	✓	✓	×	✓	×
D^2C-FORAT		D	D	Intf I, Intf D, users' demands	GW, Fog Servers, CH, CM	Set Policies for RA	RA, Notify Fog Server, Cluster size Control	CH Sel	✓	✓	✓	✓	✓	✓

The abbreviated terms are as follows:Clustering Parameters (D:Dynamic, S:Static, Intf I=Interference Intensity, Intf D=Interference Degree), Architectural Parameters (GW:Gateway, CH:Cluster Head, CM:Cluster Member, RA:Resource Allocation, CH Sel:CH Selection, PA:Power Allocation, Power Adj:Power Adjustment)

cient RBs for FBSs within a cluster, the intra-cluster interference never occurs. Moreover, we employ local Fog servers that are aware of the location and demands of Edge FBSs to run a graph-coloring-based algorithm to address the inter-cluster interference.

3.3 System Model and Problem Formulation

In this section, we describe the system model and formulate the RA as an optimization problem to maximize the network throughput.

3.3.1 System Model

We consider an OFDMA-based femtocell network in which FBSs are densely deployed. In such networks, FBSs may face two types of interferences, including cross-tier interference (i.e., interference between femtocell and macrocells [168]) and co-tier interference. This latter interference, which is significantly aggravated in dense and ultra-dense networks, can be reduced by forming clusters of FBSs and coordinating among them through their X2 interfaces [178]. Besides, the interference between macrocell and FBSs can be regarded as Additive White Gaussian Noise (AWGN) [179].

We use the 3GPP dual-strip residential apartment model [180] to represent how FBSs are deployed in our network. In this model, we have two strips of one-floor buildings, so that strips are separated by a 10m-wide street, and each one contains 20 buildings (10m $\times$ 10m). Each building comprises an FBS so that its location is set based on a uniform distribution model. In addition, since the owners can turn on/off their FBSs randomly, which change the network topology, we set the activation status of each FBS, $S_a \in \{0,1\}$, and FBS density, $\lambda \in [0,1]$. This latter represents the ratio of active FBSs to all FBSs in the network [181]. Each active FBS can support up to a maximum of four end-users that are uniformly distributed in its proximity. Moreover, the user demand of each FBS is defined as the number of requested RBs by that user. Fig. 3.1 represents an example of FBSs' deployment in the 3GPP dual-strip apartment model, where FBSs are grouped into seven different clusters. Moreover, it demonstrates the concept of intra and inter-cluster interference.

In our model, we use LTE specification for downlink, in which the available 5 MHz bandwidth is divided into several RBs so that each RB contains 12 consecutive subcarriers with 15 KHz of spacing between adjacent subcarriers and 7 OFDMA symbols with the time duration of 0.5 ms [182].

Hierarchical Architecture

We propose a hierarchical architecture in which responsibilities are proportionally distributed over the network entities, including GW, Fog servers, CHs, and CMs, to meet the requirements of dense and ultra-dense networks. Fig. 3.2 depicts an overview of

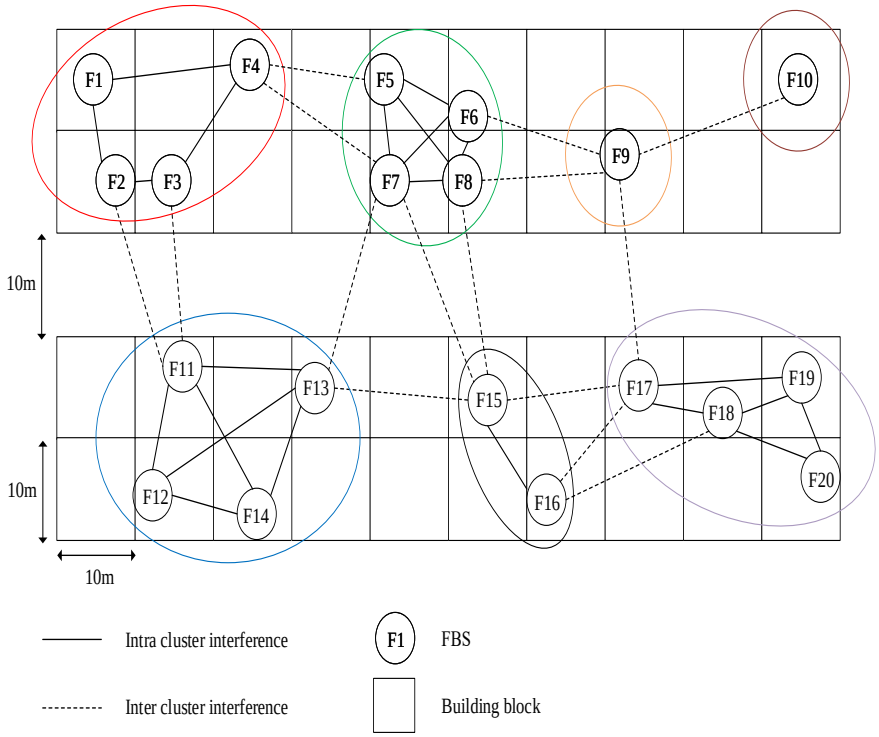


Figure 3.1: The 3GPP dual-strip residential apartment model

our hierarchical architecture. In what follows, we briefly illustrate the responsibilities of each entity and define how they collaborate.

The GW. The main responsibility of this entity is managing Operation and Maintenance (OAM) information, including FBSs' location, identification, authentication, aggregating, and validating signaling traffic [172].

The Fog Server. This entity is located between the GW and CHs, and its main responsibility is forwarding policies to its in-range CHs so that they can efficiently allocate RBs to their FBSs. It receives the clusters' configuration of FBSs and forms an Edge FBSs' interference graph. Afterward, through its computing capacity, the Fog server attempts to solve the inter-cluster interference and forward specific policies to CHs.

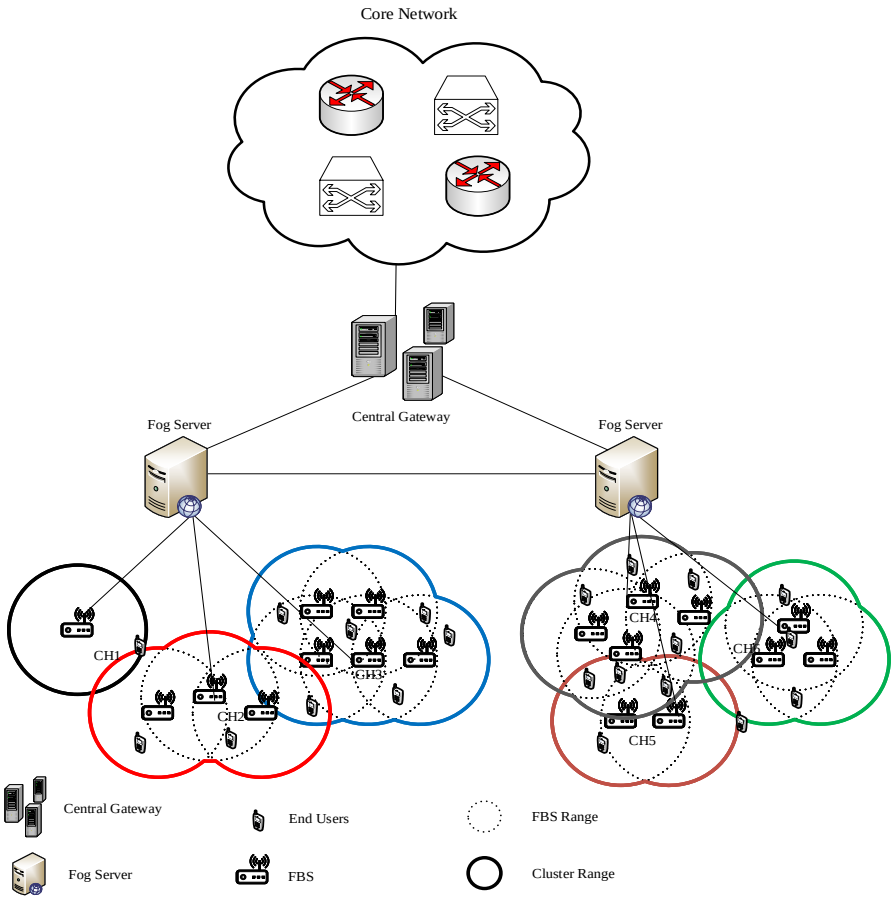


Figure 3.2: An overview of proposed hierarchical architecture

The CH. The CH is placed between CMs and the Fog server, and its main duties include notifying the Fog server of its Edge FBSs' configuration, and RA runtime. This feature helps the clusters to dynamically change and adapt themselves to the current state of the network. Furthermore, it periodically notifies the Fog server about its Edge FBSs' configuration. This latter configuration is mainly because each CH has a local view of its CMs' configuration and is not aware of other clusters' configurations, which results in less-precise RA. Finally, it allocates available resources to its CMs while considering the received policies from the Fog servers. This leads to more efficient RA, which improves the total throughput and satisfaction of end-users.

The FBS. The main responsibilities of FBSs in this architecture, alongside satisfying their users, are cluster formation and CH selection which are obtained in a distributed dynamic manner.

3.3.2 Problem Formulation

We define a set of FBSs as $\mathcal{F} = \{f_1, f_2, f_3, \ldots, f_M\}$, so that each FBS is a member of disjoint cluster set $\mathcal{C} = \{c_1, c_2, c_3, \ldots, c_L\}$. Hence, each FBS $f_i \in c_l$ can be represented by $f_{i,l}$. Moreover, we define the in-range neighbors of FBS f_i as a set of FBSs shown by $\mathcal{N}_{f_i}$. It is important to note that $\mathcal{N}_{f_i}$ contains members that are not necessarily in the same cluster as f_i. Moreover, the set of end-users of f_i are defined as $\mathcal{U}_{f_i}$.

We denote the set of RBs as Δ, and hence, the received amount of signal to interference plus noise (SINR) of each $u \in \mathcal{U}_{f_i}$ on the RB $k \in \Delta$ is defined as follows [183, 184]:

$$\gamma_{u,k}^{f_{i,l}} = \frac{P_k^{f_{i,l}} \times H_{u,k}^{f_{i,l}}}{\sigma^2 + \sum\limits_{f_{j,l'} \in \mathcal{F}, j \neq i, l \neq l'} P_k^{f_{j,l'}} \times H_{u,k}^{f_{j,l'}}} \tag{3.1}$$

where $P_k^{f_{i,l}}$ and $H_{u,k}^{f_i,l}$ are the transmission power of $f_{i,l}$ and the channel gain between u and $f_{i,l}$ on RB k, respectively. Moreover, $\sum\limits_{f_{j,l'} \in \mathcal{F}, j \neq i, l \neq l'} P_k^{f_{j,l'}} \times H_{u,k}^{f_{j,l'}}$ is the interference generated by other adjacent FBSs belonging to other clusters, called inter-cluster interference, and σ^2 is noise power density.

According to the amount of $\gamma_{u,k}^{f_{i,l}}$ on RB k, the user u can select an appropriate Modulation-and-Coding Scheme (MCS) to achieve the highest possible data rate while guaranteeing reliability. We represent the achievable data rate on the RB of this user by $R_{u,k}^{f_{i,l}}$ so that this should be always between the maximum and minimum achievable data rate, which is calculated based on MCS and SINR threshold used in [185], as follows:

$$R_{max} = (7symbol \times 4.8bit/symbol) \times 0.5ms \times 12 = 806.4kbs \tag{3.2}$$

$$R_{min} = (7symbol \times 0.66bit/symbol) \times 0.5ms \times 12 = 110.88kbs \tag{3.3}$$

To calculate R_{max} and R_{min}, the QAM-64 with code rate 4/5 and the QPSK with code rate 1/3 are used, respectively.

The principal goal of this work is to maximize the total throughput of the network while mitigating the severe interference through allocating appropriate RBs to each FBS. Thus, according to the above-mentioned goal, the clustering-based RA problem can be formulated as follows:

$$maximize \sum_{c_l \in \mathcal{C}} \sum_{f_i \in c_l} \sum_{u \in \mathcal{U}_{f_i}} \sum_{k \in \Delta} a_{u,k}^{f_{i,l}} \times R_{u,k}^{f_{i,l}} \tag{3.4}$$

s.t.

$$C1 : a_{u,k}^{f_{i,l}} \in \{0,1\}, \quad \forall u \in \mathcal{U}_{f_i}, \ \forall f_i \in c_l,$$

$$\forall c_l \in \mathcal{C}, \ \forall k \in \Delta$$

$$C2 : \sum_{f_i \in c_l} \sum_{u \in \mathcal{U}_{f_i}} \sum_{k \in \Delta} a_{u,k}^{f_{i,l}} \leq |\Delta|, \quad \forall c_l \in \mathcal{C}$$

$$C3 : \sum_{f_i \in c_l} \sum_{u \in \mathcal{U}_{f_i}} a_{u,k}^{f_{i,l}} \leq 1, \quad \forall c_l \in \mathcal{C}, \forall k \in \Delta$$

$$C4 : \sum_{k \in \Delta} a_{u,k}^{f_{i,l}} \times R_{u,k}^{f_{i,l}} \geq \hat{R}_u^{f_{i,l}}, \ \forall u \in \mathcal{U}_{f_i},$$

$$\forall f_i \in c_l, \ \forall c_l \in \mathcal{C}, \ \forall k \in \Delta$$

$$C5 : a_{u,k}^{f_{i,l}} \times R_{u,k}^{f_{i,l}} \geq a_{u,k}^{f_{i,l}} \times R_{min}, \quad \forall u \in \mathcal{U}_{f_i},$$

$$\forall f_i \in c_l, \ \forall c_l \in \mathcal{C}, \ \forall k \in \Delta$$

$$C6 : \bigcup_{c_l \in \mathcal{C}} c_l = \mathcal{F}$$

$$C7 : c_l \bigcap c_{l'} = \emptyset, \quad \forall c_l, c_{l'} \in \mathcal{C}, l \neq l'$$

In the optimization problem Eq. 3.4, the $a_{u,k}^{f_{i,l}}$ is an exclusion factor to represent whether the RB k is assigned to the user u of FBS $f_{i,l}$ or not, as depicted in the constraint $C1$. The second constraint, $C2$, expresses that each cluster can use all available network's RBs,

while the $C3$ denotes that each RB k in each cluster can be assigned only to one user. Thus, there is no intra-cluster interference in our problem. The $C4$ indicates that each user u of FBS $f_{i,l}$ should at least receive its minimum requested data rate, depicted as $\hat{R}_u^{f_{i,l}}$. The $C5$ expresses that the RB k should not be assigned by FBS f_i to its end-users whenever a severe interference exists on that RB. The $C6$ and $C7$ denote that each FBS f_i is a member of one cluster, and the set of clusters are disjoint. Table 3.2 summarizes the parameters used in this chapter and their respective definitions.

Table 3.2: Parameters and respective definitions

Parameter	Definition	Parameter	Definistion
S_a	Activation status of each FBS	λ	FBSs' density
$\mathcal{F}$, $\mathcal{M}$	Set of all FBSs, Number of all FBSs	f_i	The ith FBS
$f_{i,l}$	The FBS f_i in cluster c_l	$\mathcal{C}$, L	Set of clusters, Number of clusters
$\mathcal{N}_{f_i}$	Set of one hop neighbors of FBS f_i	$\mathcal{U}_{f_i}$	Set of end-users of FBS f_i
$\gamma_{u,k}^{f_{i,l}}$	The SINR that end user u receives from FBS $f_{i,l}$ on RB k	$R_{u,k}^{f_{i,l}}$	Data rate of end user u belongs to $f_{i,l}$ on RB k
Δ	Set of all RBs	σ^2	Noise power density
$P_k^{f_{i,l}}$	Transmission power of FBS $f_{i,l}$ on RB k	$\hat{R}_u^{f_{i,l}}$	Minimum data rate required by end user u
$H_{u,k}^{f_{i,l}}$	Channel gain between the FBS $f_{i,l}$ and the end user u on the RB k	$I(f_i, c_l)$	Relative sum interference of FBS f_i on cluster c_l
R_{max}	Maximum throughput on each RB	R_{min}	Minimum data rate on each RB
FC_{c_l}	Free capacity of cluster c_l	$f_{c_l}^{\star}$	The worst CM in the cluster c_l
$\mathcal{N}(f_i, c_l)$	Set of one-hop neighbors of FBS f_i belonging to cluster c_l	$I(f_i, f_j)$	Interference between FBS f_i and FBS f_j
$deg(f_i, c_l)$	Relative interference degree of FBS f_i on cluster c_l	$\mathcal{P}_{c_l}$	Set of all possible partitions for cluster c_l
$policy_{cl}$	The set of policies enacted for the cluster c_l	$range(f_{i,l})$	The set of authorized RBs for $(f_{i,l})$
G	The interference graph of Edge FBSs	K_{max}	The maximum number of colors
$a_{u,k}^{f_{i,l}}$	Exclusion factor indicating whether RB k is assigned to end user u of FBS $f_{i,l}$, or not	$FC_{c_l}^{\star}$	Free capacity of cluster c_l without considering $f_{c_l}^{\star}$
$I^{\star}(f_i, c_l)$	Relative sum interference of FBS f_i on cluster c_l without considering the $f_{c_l}^{\star}$	$demand_u$	The demand of end user u in terms of number of RBs

3.4 Distributed Dynamic Clustering Method

In this section, we propose a Distributed Dynamic Clustering (D^2C) method, in which FBSs with the highest relative interference form different clusters. Moreover, FBSs in each cluster select one FBS as their CH. The CH dynamically controls the cluster size based on the requested demands of its end-users and makes the decision whether a new FBS can join the cluster or not, accordingly. The D^2C has three principal functions including new FBS arrival (NFA), update clustering parameters (UCP), and cluster migration possibility (CMP).

3.4.1 New FBS Arrival (NFA)

Whenever a new FBS f_i joins the network, it creates its neighbor list $\mathcal{N}_{f_i}$, in which each $f_j \in \mathcal{N}_{f_i}$ is either a CM or CH. If $\mathcal{N}_{f_i}$ does not have any CH member, the f_i creates the cluster c_l, set itself as the CH, and calculates the free capacity of the cluster as follows:

$$FC_{c_l} = |\Delta| - \sum_{f_i \in c_l} \sum_{u \in \mathcal{U}_{f_i}} demand_u \quad \forall c_l \in \mathcal{C} \tag{3.5}$$

where $demand_u$ depicts the number of RBs requested by the user $u \in \mathcal{U}_{f_i}$.

If the $\mathcal{N}_{f_i}$ contains CH members, the f_i sends a message to those CHs and requests their free capacities, FCs. Afterward, clusters whose FCs are greater than or equal to $\sum_{u \in \mathcal{U}_{f_i}} demand_u$ are considered as candidate clusters by f_i. To select the best candidate cluster to join, the f_i calculates the relative sum interference of itself on all members of each candidate cluster as follows:

$$I(f_i, c_l) = \sum_{f_j \in c_l, i \neq j} I(f_i, f_j) \tag{3.6}$$

where $I(f_i, f_j)$ represents the interference between f_i and f_j as calculated in [186].

$$I(f_i, f_j) = P^{f_j} \times H_{f_i}^{f_j} \tag{3.7}$$

To simplify the problem, we assume the mutual interference between two FBSs is sym-

metric, as shown in the following:

$$I(f_i, f_j) = I(f_j, f_i) \tag{3.8}$$

Among all candidate clusters, f_i selects the candidate cluster by which it has the highest relative sum interference, $I(f_i, c_l)$, and sends the soft-join request to that cluster's CH. The soft-join indicates that the candidate cluster has enough FC to accept the join-request of f_i while the CH guarantees to allocate sufficient requested RBs to its current end-users.

If there is no candidate clusters with sufficient FCs to support the required demand of f_i, the hard-join is considered as a potential solution. In this latter, the candidate cluster c_l should substitute its worst FBS, called $f^{\star}_{c_l}$, for the f_i. To identify the $f^{\star}_{c_l}$, each $f_{j,l}$ creates the $\mathcal{N}(f_j, c_l) \subseteq \mathcal{N}_{f_j}$where $\mathcal{N}(f_j, c_l)$ denotes one-hop neighbors of f_j that are in the cluster c_l. Moreover, the FBS f_j calculates its relative interference degree on cluster c_l as $deg(f_j, c_l) = |\mathcal{N}(f_j, c_l)|$. Hence, the CH selects the member with the lowest relative interference degree as $f^{\star}_{c_l}$ on its cluster. In a case that there are several members with the lowest $deg(f_j, c_l)$, the member whose interference with other members is the lowest, is selected as the $f^{\star}_{c_l}$. Considering the fact that any cluster c_l has its $f^{\star}_{c_l}$, the f_i sends message to each neighboring CH to obtain its free capacity while worst FBS $f^{\star}_{c_l}$ is excluded, as shown in the following:

$$FC^{\star}_{c_l} = FC_{c_l} + \sum_{u \in \mathcal{U}_{f^{\star}_{c_l}}} demand_u, \quad \forall c_l \in \mathcal{C} \tag{3.9}$$

Afterward, each cluster whose $FC^{\star}_{c_l}$ is greater than or equal to $\sum_{u \in \mathcal{U}_{f_i}} demand_u$ is considered as candidate cluster for hard-join. Among these candidate clusters, the f_i sends the hard-join request to the CH of the cluster by which it has the highest relative sum interference $I^{\star}(f_i, c_l)$.

$$I^{\star}(f_i, c_l) = I(f_i, c_l) - I(f_i, f^{\star}_{c_l}) \tag{3.10}$$

In a case that there is no possibility to perform soft-join and hard-join, the f_i acts exactly the same as the condition that there are no CHs in its $\mathcal{N}_{f_i}$, as discussed earlier, and forms a new cluster. Fig. 3.3 represents the process of NFA function.

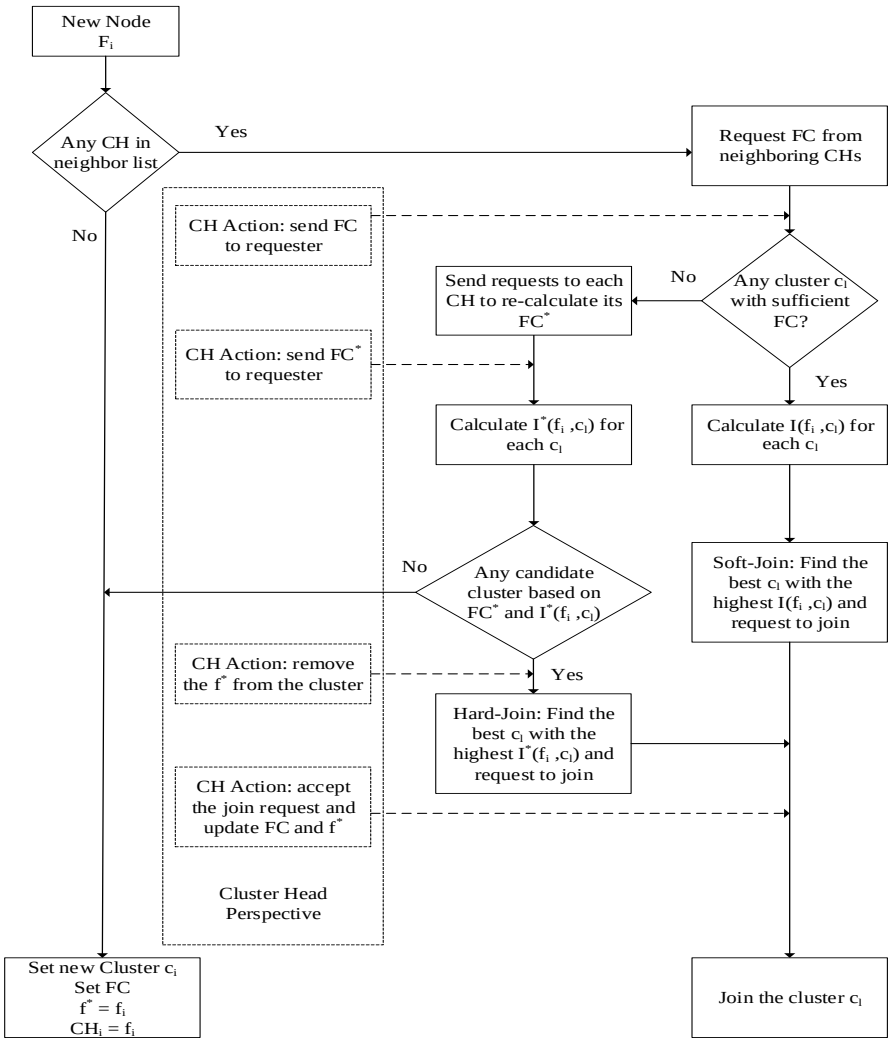


Figure 3.3: New FBS Arrival (NFA) flowchart

3.4.2 Update Clustering Parameters (UCP)

When the configuration of a cluster c_l changes (e.g. joining or removing a member), its CH requests all its members $f_{x,l}$ to calculate their $deg(f_x, c_l)$ and $I(f_x, c_l)$, and send them back. Then, it makes a priority list of its members according to these parameters so that members with the largest relative interference degree receive higher priority. If there are several members with the same relative interference degree, those members are sorted in terms of the relative sum interference. Finally, the member with the highest priority is selected as the CH, and the member with the lowest priority is selected as the $f^{\star}_{c_l}$.

3.4.3 Cluster Migration Possibility (CMP)

The main goal of this function is to check whether any CM can migrate to other clusters so that the quality of clustering improves or not. This function is called by the CMs that at least have one another CH in their neighbor lists belonging to another cluster. To illustrate, we consider $f_{i,l}$ and $f_{j,l'}$, $f_j \in \mathcal{N}_{f_i}$ and the fact that $f_{j,l'}$ is the CH of $c_{l'}$. Then, the $f_{i,l}$ should request the $FC_{c_{l'}}$ and periodically calculate the $I(f_i, c_{l'})$. If $I(f_i, c_l) < I(f_i, c_{l'})$, then f_i sends the soft-join request to the f_j to join the cluster $c_{l'}$. This guarantees that each FBS always attempts to join a cluster to which it has the highest relative interference. This latter helps to improve clusters as the network configuration changes, and consistently attempts to maintain FBSs with the highest relative sum interference as a cluster, which finally leads to less inter-cluster interference.

3.5 A New Resource Allocation Method

In this section, we propose a RA method in which Fog servers and CHs collaborate to mitigate the interference and improve the network throughput.

Each CH is responsible for allocating the RBs so that no intra-cluster interference occurs in its cluster. Since each CH is unaware of adjacent clusters' RAs, there is a high probability of inter-cluster interference on Edge FBSs. This problem is aggravated in dense and ultra-dense networks to the point that the network throughput is severely dropped [171]. Hence, we concurrently consider both intra and inter-cluster interference for the RA so that our method can be applied to femtocell networks ranging from sparse to ultra-dense. In this method, the Fog servers are responsible to provide a set of policies to CHs to minimize inter-cluster interference. CHs consider these policies and their users' demands, and aim at maximizing clusters' throughput alongside decreasing the interference by proper allocation of resources.

3.5.1 Policy Identification

We divide the FBSs of each cluster into two categories containing central and Edge FBSs. The $f_{i,l}$ is considered as a central member whenever $\mathcal{N}_{f_i}$ only contains neighbors from c_l,

while it is considered as Edge member if the $\mathcal{N}_{f_i}$ contains any member from other clusters, as noted in Eq. 3.11. Apparently, the central nodes never experience inter-cluster interference.

$$f_{i,l} \ is \begin{cases} \text{Central}, & \text{if } \mathcal{N}_{f_i} - \mathcal{N}(f_i, c_l) = \text{ø} \\ \text{Edge}, & \text{if } \mathcal{N}_{f_i} - \mathcal{N}(f_i, c_l) \neq \text{ø} \end{cases} \tag{3.11}$$

The Fog servers provide a set of policies for each cluster c_l, in which each element contains the specific FBS on which that policy should be applied, and a specific subset of RBs, called *range* representing the RBs which can be assigned to that FBS, as shown in the following.

$$policy_{c_l} = \{(f_{i,l}, range(f_{i,l})) \mid f_{i,l} \ is \ Edge, \ range(f_{i,l}) \subseteq \Delta\} \tag{3.12}$$

The policies to mitigate the inter-cluster interference are provided in three phases including graph formation, graph coloring, and graph relaxation, as discussed in the following. The Algorithm 1 represents a general view of policy identification through these phases.

Graph Formation

The main goal of this phase is to form an interference graph of Edge FBSs. To achieve this, CHs send their neighbor lists of Edge FBSs and their respective demands to the Fog servers. Afterward, the Fog servers create the weighted graph of Edge FBSs, $G(V, E, W_v, W_e)$, based on the information received from their corresponding CHs. V represents the set of vertices so that each vertex denotes an Edge FBS. E explains the set of edges so that each edge represents the interference between two Edge FBSs, as follows:

$$e_{v_i,v_j} \ is \begin{cases} 1, & \text{if } f_i \in \mathcal{N}_{f_j} \\ 0, & \text{if } f_i \notin \mathcal{N}_{f_j} \end{cases} \tag{3.13}$$

Algorithm 1: General view of policy identification algorithm

```
   %Graph Formation Phase:
 1 create G(V, E, W_v, W_e)
   /* The graph G is comprised of several connected graph g_i
      as G = {g_1, g_2, ..., g_Z}                                   */
   %Graph Coloring Phase:
 2 for z = 1 to Z do
 3 |   calculate K_max from Eq. 3.17
   |   /* The K_max is the maximum number of colors                */
 4 |   for k=2 to K_max do
 5 |   |   g_z^* = graph-simplification(g_z, k)
 6 |   |   if graph-coloring(g_z^*, k)==false then
 7 |   |   |   if k + 1 ≤ K_max then
 8 |   |   |   |   continue
 9 |   |   |   else
   |   |   |   |   %Graph Relaxation Phase:
10 |   |   |   |   g_z = graph-relaxation(g_z)
11 |   |   |   |   go to line 3
12 |   |   |   end
13 |   |   else
14 |   |   |   send policies to CHs
15 |   |   |   break
16 |   |   end
17 |   end
18 end
```

It is important to note that based on the Eq. 3.13, Edge FBSs that are in the same cluster and in range of each other are connected by an edge in the G. Moreover, the W_v and W_e depict the set of weights for vertices and edges respectively. The weight of each vertex v_i is calculated as the total users' demands of each Edge FBS as follows:

$$W_{v_i} = \sum_{u \in \mathcal{U}_{f_i}} demand_u \tag{3.14}$$

In addition, the weight of each edge e_{v_i,v_j} is the amount of interference between those vertices, as depicted in the following:

$$if \;\; e_{v_i,v_j} \in E \;\; then \;\; w_{v_i,v_j} = I(f_i, f_j) \tag{3.15}$$

Finally, the weighted graph G is not necessarily a connected graph, and it can be comprised of several connected graphs $G = \{g_1, g_2, ..., g_Z\}$, with the following condition:

$$g_z \bigcap g_{z'} = \text{ø}, \quad \forall g_z, g_{z'} \in G, \quad z \neq z' \tag{3.16}$$

Graph Coloring

In this phase, we address the inter-cluster interference by a graph coloring method for every connected graph g_z in the G, created in the graph formation phase. The main goal of the graph coloring method is to find a set of different colors (so that each color represents a set of RBs) for Edge FBSs and assign a set of respective policies to CHs for the RA so that the inter-cluster interference reduces.

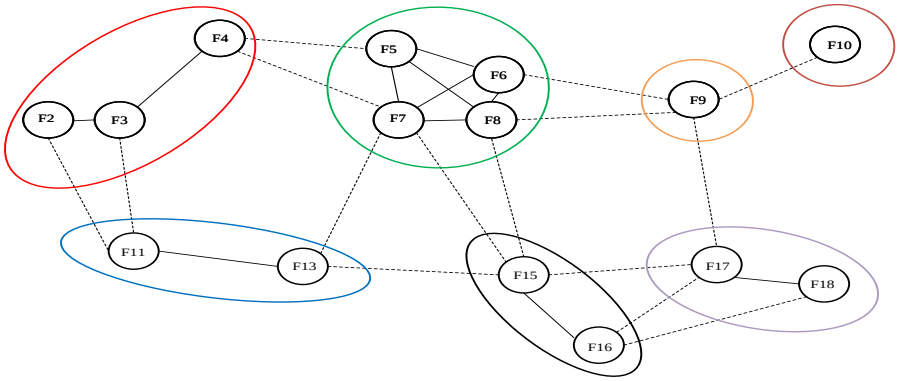
Because each assigned color to an Edge FBS should contain sufficient RBs to support demands of all users belonging to that FBS, the maximum number of colors K_{max} is restricted and is calculated as follows:

$$K_{max} = \lfloor \frac{\Delta}{W_x} \rfloor, \; x = arg\,max\,(W_{v_i}), \;\; \forall v_i \in V_{g_z} \tag{3.17}$$

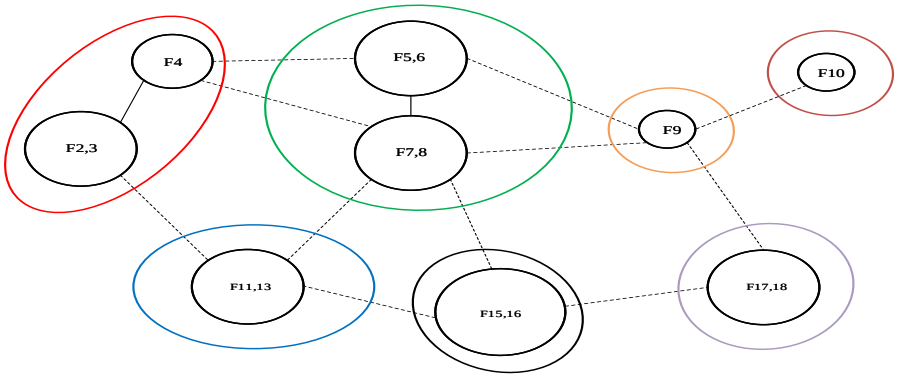
where the W_x represents the weight of the heaviest vertex $x \in V_{g_z}$.

Thus, the graph coloring problem is changed to the k-coloring problem, so that we should iterate from $k = 2$ to K_{max} to find the least possible number of colors, which is a time-consuming problem. In this phase, we also propose a greedy simplification algorithm to simplify the connected graph g_z, so that we can color this graph in a timely manner. The Algorithm 2 demonstrates the graph simplification.

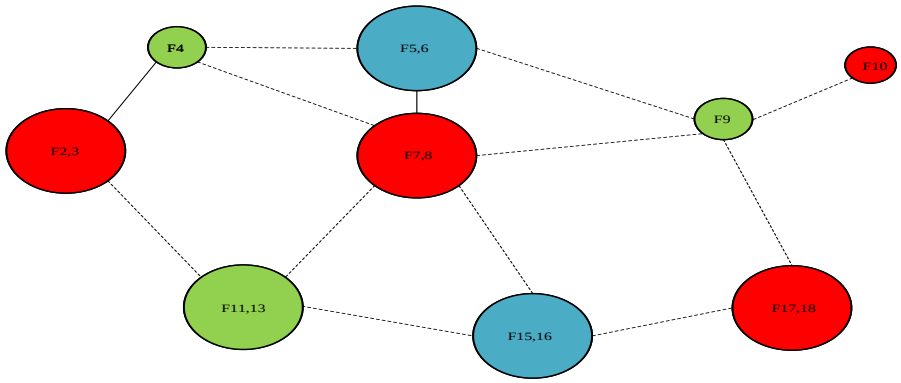
The simplification algorithm starts from the least possible number of colors $k = 2$, and tries to combine the Edge FBSs of each cluster, if possible, and creates the $g_z^\star$. We can combine any two Edge FBSs $v_i, v_j \in g_z$ if they belong to same cluster, and an edge $e_{v_i,v_j} \in E_{g_z}$ exists between them, and their aggregate vertices' weights is not higher than the capacity of colors ($\frac{|\Delta|}{k}$). According to this latter, we can make a new vertex v_{ij} in the $g_z^\star$, whose weight is the aggregate weights of v_i, v_j, and v_{ij} connects to any vertices to



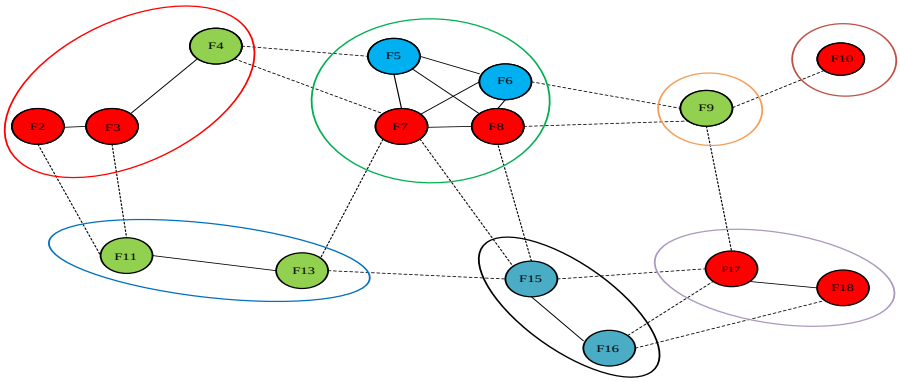
(a) The graph g_z of Edge FBSs



(b) The simplified graph $g_z^\star$



(c) A successful coloring of simplified graph $g_z^\star$ by 3 colors



(d) The colored version of g_z

Figure 3.4: An example demonstrating graph coloring phase based on FBS configuration depicted in Fig. 3.1

Algorithm 2: Graph simplification

input : g_z: An instance of subgraph,
k: The number of colors

/* $\mathcal{P}_{c_l}$: Set of all possible partitions of c_l, Π: The partition with the lowest number of classes */

1 initialize $g_z^\star = g_z$
2 **for** $l = 1$ to find-size-cluster(g_z) **do**
  /* The method find-size-cluster(g_z) determines the number of clusters belongs to g_z */
3 $\mathcal{P}_{c_l}$ = make-partition (c_l)
  /* The make-partition (c_l) method creates ascending sorted list of all partitions, and its size is obtained from Dobinsky Formula */
4 Π = find-partition($\mathcal{P}_{c_l}$)
  /* The find-partition($\mathcal{P}_{c_l}$) method returns the partition with the least number of classes, $\Pi = \{A_1, A_2, ..., A_t\}$, t = number of classes in Π */
5 **for** $j = 1$ to t **do**
6 **if** $\sum_{v_i \in A_j} W_{v_i} > \frac{|\Delta|}{k}$ **then**
7 $\mathcal{P}_{c_l} = \mathcal{P}_{c_l} - \Pi$
8 go to line 4
9 **end**
10 **end**
11 **for** $j = 1$ to t **do**
12 combine(A_j)
  /* The combine(A_j) method merges all vertices in A_j */
13 **end**
14 **end**

which v_i or v_j was connected previously.

$$\begin{aligned} &if \;\; v_i, v_j \in c_l \;\; \& \;\; e_{v_i,v_j} \in E_{g_z} \;\; \& \;\; w_{v_i} + w_{v_j} \leq \tfrac{|\Delta|}{k} \\ &then \;\; combine(v_i, v_j) \;\; as \;\; v_{ij}, \;\; \forall v_i, v_j \in V_{g_z}, \forall c_l \in \mathcal{C} \end{aligned} \tag{3.18}$$

It is crystal clear that if the number of Edge FBSs within the c_l is equal to n, $n > 2$, several configurations for the combination of any number of Edge FBSs in c_l can be considered,

which is obtained by the Dobinsky's formula, as follows [187].

$$|\mathcal{P}(c_l)| = \sum_{m=1}^{n} \frac{m^n}{m!} \sum_{j=0}^{n-m} \frac{(-1)^j}{j!} \tag{3.19}$$

where we denote the set of all possible partitions of cluster c_l as $\mathcal{P}(c_l)$, in which each partition is a set of classes $\{A_1, A_2, ..., A_t\}$. Each class A_j contains one or more Edge FBSs of cluster c_l that should be evaluated whether they can be considered as one vertex or not. The *make-partition()* method creates an ascending sorted list of all partitions based on the number of classes, t, of the partitions. In the next step, the *find-partition()* method returns the partition with the least number of classes, defined as Π, from the partition set $\mathcal{P}(c_l)$. For each class A_j in Π, it is examined whether the aggregate weights of that class is less than $\frac{|\Delta|}{k}$ or not. If there is even one class in the partition Π that does not satisfy this condition, the partition is removed from the $\mathcal{P}(c_l)$ and the algorithm searches for the next candidate partition. However, if all classes satisfy the condition, FBSs of the same class can be combined, and a new simplified graph $g_z^\star$ will be created.

If $g_z^\star$ is colored by k colors, the range of RBs for each color can be specified and respective policies will be sent to the CHs. But, in a case that the $g_z^\star$ cannot be colored by k colors, the k increases and the *graph-simplification()* method is invoked to simplify the graph. Fig. 3.4 depicts an example of graph coloring phase. Fig. 3.4a represents the graph of Edge FBSs derived from FBSs' configuration of the Fig. 3.1. Fig. 3.4b demonstrates a candidate simplified graph $g_z^\star$, in which several nodes in each cluster are combined. Afterward, the $g_z^\star$ is colored by three colors as it can be seen in Fig. 3.4c, and the Fig. 3.4d denotes how these colors are represented in the g_z.

In an ultra-dense network that even simplification cannot help to color the graph by K_{max} colors, we provide a backup plan as graph relaxation phase.

Graph Relaxation

This phase is the backup phase for the graph coloring and is invoked if graph coloring cannot find any solution to color the g_z up to K_{max} colors. The graph relaxation phase attempts to decrease the maximum vertex degree of sub-graph g_z by ignoring the weak

interferences leading to the creation of new g_z with fewer constraints. The *relaxation()* method receives the subgraph g_z and finds the vertex or vertices with the largest interference degree because they incur the strongest constraints on the problem. Afterward, the lightest Edge of those vertices is omitted to reduce their interference degree, as shown in the following:

$$\begin{gathered} if \;\; deg(v_x) = deg(g_z) \;\; then \;\; E_{g_z} = E_{g_z} - e_{v_x,v_y}, \\ y = arg\,min(W_{v_x,v_k}), \;\; \forall v_k \in V_{g_z}, e_{v_x,v_k} \in E_{g_z}, \forall v_x \in V_{g_z} \end{gathered} \tag{3.20}$$

The new subgraph g_z is then created and sent to the graph coloring phase.

3.5.2 Policy Aware Resource Allocation

Because CHs receive policies for their Edge FBSs from Fog servers, they should apply those policies in their RA. The Algorithm 3 indicates an overview of policy-aware RA. To achieve the aforementioned goals, the FBSs that are not assigned any RBs, are divided into two sets, including S_1, S_2 by each CH. The first set S_1 belongs to FBSs for which the Fog server sends some policies to their respective CHs, while the second set S_2 contains the FBSs for which no policies are assigned. Because more restrictions are applied on the S_1, it has higher priority compared to S_2 whenever CH assigns resources. These lists are sorted based on total users' demands of FBSs so that FBSs with the highest required RBs that are not satisfied yet are placed in front of the lists. In each iteration, each CH selects an RB from the unallocated RBs, shown as Δ', and assigns that RB to a FBS f_i, existing in S_1, if that RB is in $range(f_i)$. In the case that there exists no FBS in the S_1 or the *range* of that FBS does not comply with the selected FBS, the CH searches the S_2 to find a proper FBS to which it can assign that RB. The algorithm is finished whenever there are no FBSs in the sets S_1, S_2, demonstrating that all FBSs are satisfied.

Algorithm 3: Policy aware resource allocation of each cluster c_l

input : Δ: Set of RBs,
Δ': Set of unallocated RBs,
S_1: Unsatisfied sorted list of FBSs with assigned policies,
S_2: Unsatisfied sorted list of FBSs without assigned policies

initialize $\Delta' = \Delta$
while $S_1 \neq \varnothing, S_2 \neq \varnothing$ **do**
  r = select RB from Δ'
  if $\exists f_i \in S_1$ so that $r \in (\Delta' \bigcap range(f_{i,l}))$ **then**
    assign RB r to f_i
    update set S_1
    sort set S_1
  end
  if $\exists f_j \in S_2$ **then**
    assign RB r to f_j
    update set S_2
    sort set S_2
  end
  $\Delta' = \Delta' - r$
end

3.6 Performance Evaluation

In this section, we evaluate the performance of our proposed solution through extensive simulations under different scenarios and compare it with the state-of-the-art RA techniques to understand its efficiency. We discuss system parameters and study the obtained results in the performance study subsection.

3.6.1 System Setup and Parameters

With regard to the simulation study, all algorithms are implemented in the MATLAB version R2018b on a machine with a 2.2 GHz Intel Core i7 CPU and 16 GB of RAM.

We assume an environment in which FBSs are located according to the dual-strip model, discussed in Section 3.3, so that each FBS has two users in its proximity. The channel model of [180] is used for the propagation environment so that the channel gain includes path-loss and shadowing. The transmission power and range of each FBS is supposed to be 13dBm and 30m, respectively. Besides, the FBS density is assumed to

Table 3.3: Evaluation parameters

Evaluation Parameters	**Value**
Carrier frequency	2 GHz
Bandwidth	5 MHz
No. of available RBs	25
No. of sub-carrier per RB	12
Bandwidth per sub-carrier	15 KHz
Bandwidth per RB	180 KHz
Path loss model	3GPP TR 36.814
FBS transmitted power	13dBm
Apartment dimension	$10m \times 10m$
FBS radius	30m
Minimum separation between end-users and FBS	2m
User demand	1-4 RBs
FBS density	λ = 0.5 (20 active FBS), λ = 1 (40 active FBS)
Number of end-users per FBS	2
MCS	QPSK (1/3,1/2,2/3,3/4), 16-QAM (1/2,2/3,3/4,4/5), 64-QAM (2/3,3/4,4/5)
Variance of AWGN	σ^2 = -174 dBm/Hz

be $\lambda = 0.5$ and $\lambda = 1$, of which the first one represents a dense FBS network while the second one illustrates an ultra-dense FBS network. Moreover, the total number of available RBs in the network equals to 25, and the users' demands for different experiment scenarios vary between 1 to 4 RBs. We used the MCS table of [185], which has 12 different steps for three modulations including QPSK, 16-QAM, and 64-QAM. Table 3.3 summarizes evaluation parameters and their respective values.

3.6.2 Performance Study

We employed four quantitative parameters, including throughput, interference, fairness, and throughput satisfaction, to comprehensively study the behavior of our proposed solution, called D^2C-FORAT, and to compare its efficiency with other solutions in the literature. We implemented QFCRA [177] and LFCRA [171] which are distributed

clustering proposals discussed in related work, and distributed random access (DRA) proposal [188]. The DRA works based on a random selection of resources by each FBS and re-selection of interfered RBs by the randomized hashing function. Each experiment is conducted for two different values of λ, and the outcomes are the average of 200 runs.

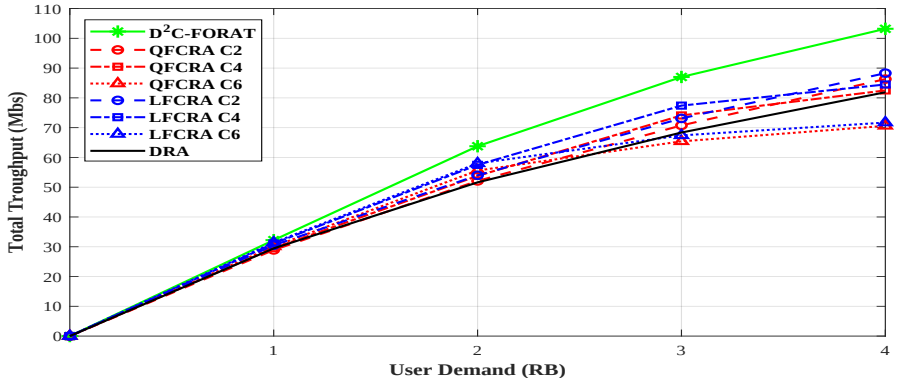
Throughput Analysis

The total throughput of each technique is calculated based on Eq. 3.4 which represents the total throughput of all users in the network. Fig. 3.5 illustrates the total throughput of D^2C-FORAT and its counterparts for different values of λ. As it can be seen from Fig. 3.5a and Fig. 3.5b, the throughput of all techniques increases as the users' demands grow, while the growth rate decreases in higher demands due to increased interference. Besides, the D^2C-FORAT outperforms its counterparts by the maximum of 17% ($\lambda = 0.5$) and 21% ($\lambda = 1$) compared to the second-best technique. This improvement is the result of policies enacted for Edge FBSs and our dynamic cluster size, resulting in better RA. The throughput of the QFCRA and the LFCRA heavily depends on their cluster size, to the extent that their throughput falls below the DRA when their cluster size is 6.

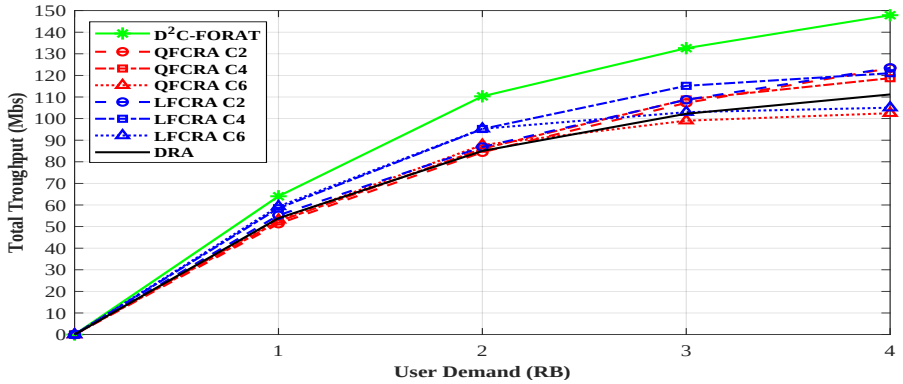
Interference Analysis

The interference between FBSs occurs whenever the overlapping FBSs use the same RBs simultaneously. Considering the SINR on the interfered RBs, the interference can be so weak, by which the throughput on those RBs does not decrease, or it can be so high, which results in unusable RBs. As the FBS density and users' demands increase, this problem occurs more often which has a significant negative impact on the total network throughput. In this chapter, we consider RBs on which the throughput is less than R_{max} as interfered RBs. Based on the interference intensity, the interfered RBs can be divided into three categories including strong, moderate, and weak, for which the QPSK, 16-QAM, and 64-QAM are respectively selected as the MCS.

Fig. 3.6 represents obtained results for the number of interfered RBs with their corresponding categories. The QFCRA follows a conservative approach, and if any interference occurs between two FBSs, one of them will be prevented from using that RB, which



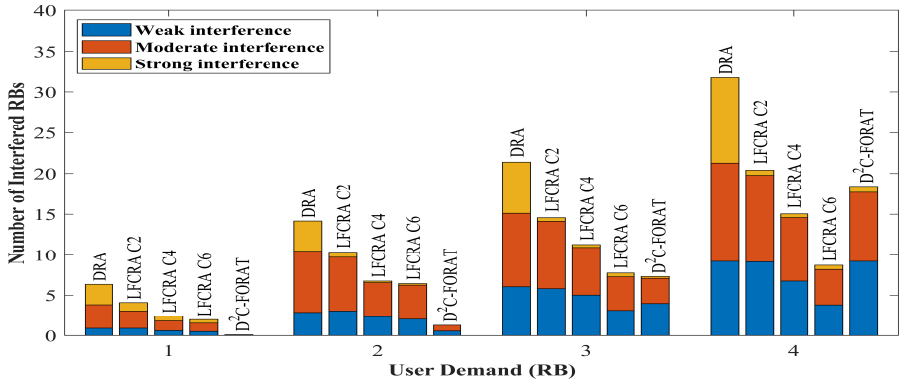
(a) FBS density $\lambda = 0.5$



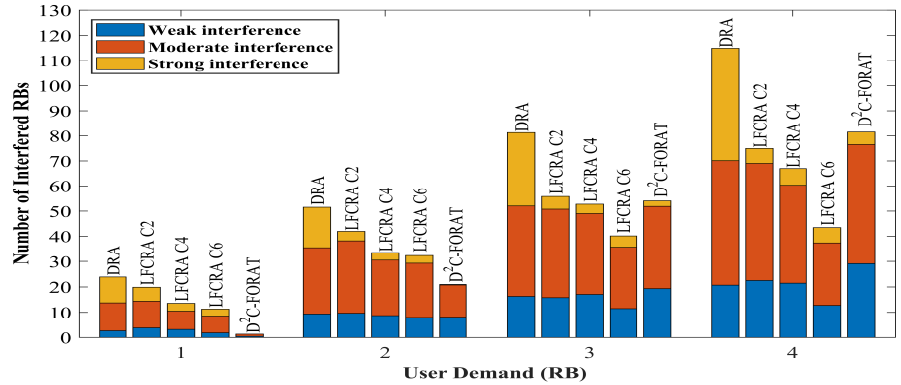
(b) FBS density $\lambda = 1$

Figure 3.5: Total throughput analysis using different FBS density λ and users' demands

results in no interference whenever the system is converged. However, this brings about several issues, such as less throughput due to smaller RB reuse, as shown in the throughput analysis. As it can be seen from Fig. 3.6a and Fig. 3.6b, as the users' demands and FBS density increase, the number of interfered RBs increases. These results show that the DRA, due to its intrinsic random behavior and lack of coordination between FBSs, suffers from a high number of interfered RBs, so that number of RBs that experiences strong interference is also more than its counterparts. The performance of LFCRA heavily depends on its cluster size, so that smaller cluster size incurs more interference, and larger cluster size results in a reduced number of interfered RBs. In contrary to LFCRA, as long as unallocated RBs are available, the D^2C-FORAT achieves the minimum number of interfered RBs compared to its counterparts (Fig. 3.6a: users' demands 1 to 3 RB, and Fig. 3.6b: users' demands: 1 to 2 RBs), while it accepts some interference if



(a) FBS density $\lambda = 0.5$



(b) FBS density $\lambda = 1$

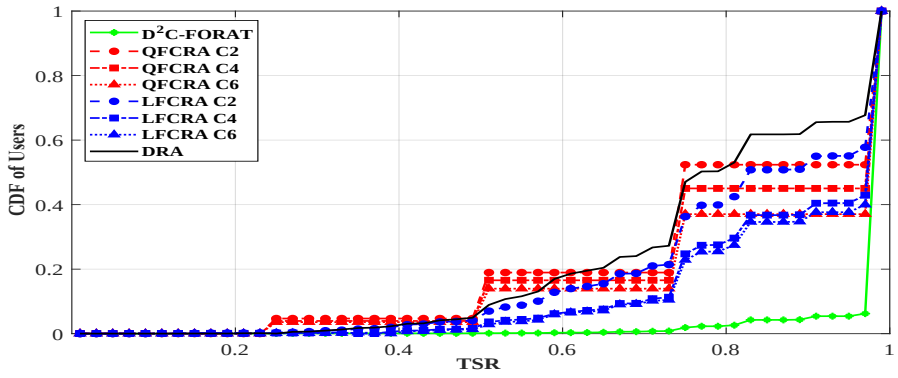
Figure 3.6: Interference analysis using different FBS density λ and users' demands with three different interference levels including Weak, Moderate, and Strong

throughput is satisfying. Besides, it is worth mentioning that the D^2C-FORAT obtains the minimum number of strong interference, which has the most negative effect on the total throughput.

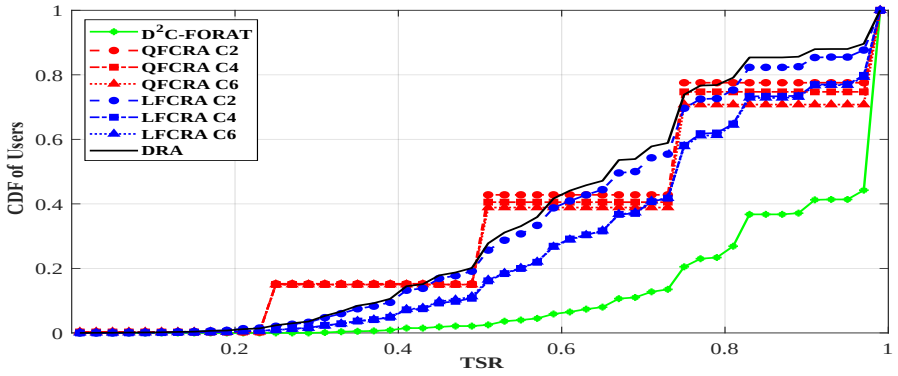
Throughput Satisfaction Analysis

The Throughput Satisfaction Rate (TSR) is a quantitative parameter demonstrating the satisfaction degree of each user. The TSR is defined as the ratio of the actual data rate of one user to its requested data rate, as depicted in the following.

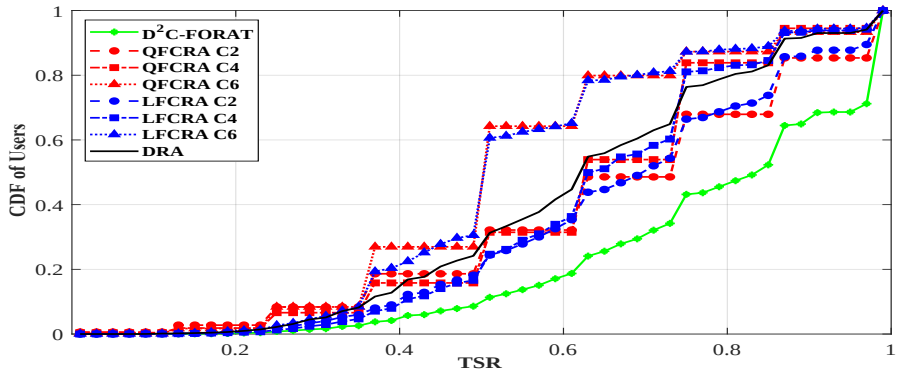
$$TSR(u) = \frac{\sum_{k \in \Delta} a_{u,k}^{f_i} \times R_{u,k}^{f_i}}{demand_u \times R_{max}}, \quad \forall u \in \mathcal{U}_{f_i}, \forall f_i \in \mathcal{F} \tag{3.21}$$



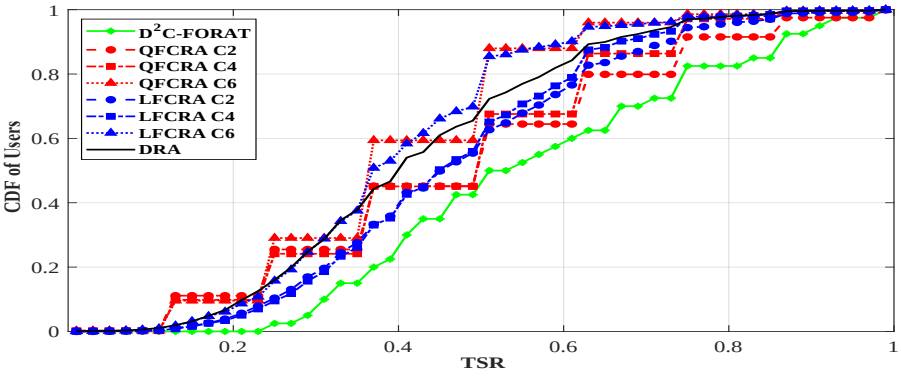
(a) $\lambda = 0.5$ and user demand = 2 RB



(b) $\lambda = 1$ and user demand = 2 RB



(c) $\lambda = 0.5$ and user demand = 4 RB



(d) $\lambda = 1$ and user demand = 4 RB

Figure 3.7: Throughput Satisfaction Rate (TSR) using different FBS density λ and users' demands

Fig. 3.7 represents the Cumulative Distributed Function (CDF) of the TSR for different values of λ and users' demands. As it can be observed, increasing users' demands in all techniques leads to less satisfaction, however, the D^2C-FORAT still outperforms its counterparts. This latter is because our technique dynamically controls cluster size, so that requested RBs can be completely assigned to end-users. Also, it mitigates the interference on those RBs by the policy-aware RA technique. In scenarios in which the number of RBs is greater than users' demands, the clustering techniques with larger cluster size incur less interference and better performance, however, as the users' demands increase the CHs are obliged to distribute the RBs between more users, and hence, the TSR significantly decreases. This latter can be observed in Fig. 3.7 when users' demands increase from 2 RB (Fig. 3.7a and Fig. 3.7b), in which techniques with larger cluster size are more efficient in terms of the TSR, to 4 RB (Fig. 3.7c and Fig. 3.7d) in which techniques with smaller cluster size can better satisfy the end-users. Fig. 3.7a denotes that more than 95% of the end-users have their TSR greater than 0.95 for D^2C-FORAT, which achieves 51% improvement compared to the second-best technique. Fig. 3.7b represents that 59% of end-users have the TSR greater than 0.95, which improves the second-best technique by 96%. Fig. 3.7c and Fig. 3.7d depict the TSR results whenever users' demands are 4 RB, in which the D^2C-FORAT achieves 0.8 user satisfaction for more than 55% and 18% of end-users, respectively. These results demonstrate that our technique improves second-best techniques in Fig. 3.7c and Fig. 3.7d by 67% and 97%, respectively.

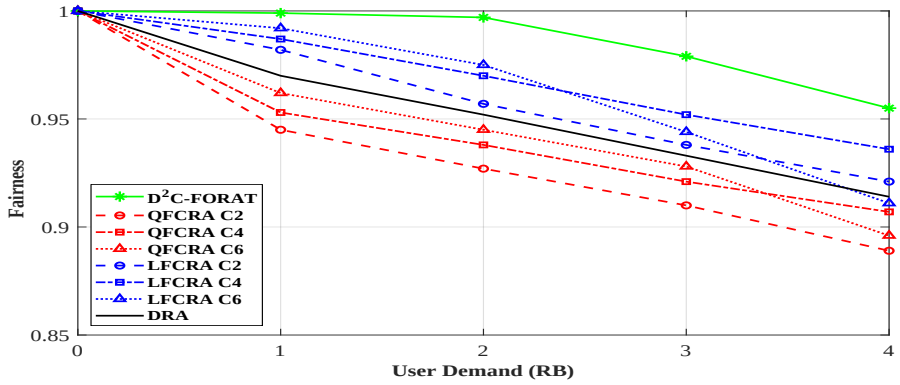
Fairness Analysis

The Jain fairness index [189] is used to evaluate how fairly RBs are allocated between different end-users, as expressed in the following:

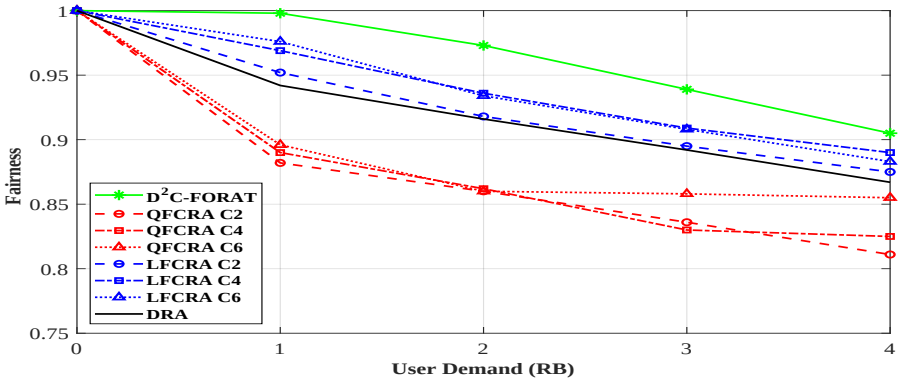
$$Fairness = \frac{(\sum_{u=1}^{N} TSR(u))^2}{N \times \sum_{u=1}^{N} TSR(u)^2} \tag{3.22}$$

where N represents the total number of end-users in the system, and the maximum value for fairness is equal to 1 when all RBs are fairly allocated between end-users.

As it can be seen from Fig. 3.8, as the users' demands and the FBS density increase,



(a) FBS density $\lambda = 0.5$



(b) FBS density $\lambda = 1$

Figure 3.8: Fairness analysis using different FBS density λ and users' demands

the fairness decreases due to the increased number of interfered RBs. However, the D^2C-FORAT outperforms other techniques due to the policies enacted for Edge FBSs and dynamic clustering that considers users' demands for controlling cluster size. Besides, the LFCRA obtains better results compared to the QFCRA because it improves the management of inter-cluster interference, which results in less interference for the Edge FBSs. Furthermore, the DRA outperforms the QFCRA, because all end-users in the DRA receive RBs either interfered or non-interfered ones, while the QFCRA attempts to assign only non-interfered RBs to end-users which results in unsatisfied end-users, specifically end-users of the Edge FBSs.

3.7 Summary

In this chapter, we proposed a distributed dynamic clustering-Fog driven RA technique, called D^2C-FORAT, to address the interference problem of Edge devices, and to increase the total network throughput. Moreover, we used a hierarchical architecture, including the GW, Fog servers, CHs, and CMs, among which the clustering and RA responsibilities are distributed. This latter results in better scalability, helping our technique to be efficiently run in sparse, dense, and ultra-dense networks. We proposed a distributed dynamic clustering method, in which FBSs select CHs, which are responsible to manage their corresponding cluster size based on the total demands of their CMs and available RBs. Moreover, each CH is responsible for allocating the cluster RBs and notifying its corresponding Fog server of the cluster's parameters. Because inter-cluster interference is a big issue in clustering techniques, which decreases the total throughput of the network, we proposed a policy-aware Fog-driven RA method to reduce such interferences. This method has three phases including graph formation, simplification, and relaxation which are performed on the Fog servers located at the proximity of clusters. The outcome of these phases is a set of policies for Edge FBSs of each cluster, by which the CH can assign the RBs more efficiently and prevent severe inter-cluster interference. The effectiveness of the D^2C-FORAT is analyzed through extensive experiments and comparison by state-of-the-art techniques in the literature. The obtained results demonstrate that our proposed solution outperforms other existing techniques in terms of total network throughput, user satisfaction, and fairness by up to 21%, 97%, and 10%, respectively.

This chapter presented a Fog-driven technique to improve the total throughput and interference of the computing environment. In the next chapter, we study scheduling of concurrent IoT applications to improve their execution cost.

Chapter 4

Batch Application Placement Technique for Concurrent IoT Applications

The placement of IoT applications in the Fog computing environment, in which several distributed and heterogeneous Fog servers and centralized Cloud servers are available, is a challenging issue. In this chapter, we propose a weighted cost model to minimize the execution time of IoT applications and energy consumption of IoT devices in a computing environment with multiple IoT devices, multiple Edge and Fog servers, and hybrid Cloud servers. Besides, a new application placement technique based on the Memetic Algorithm is proposed to make batch application placement decisions for concurrent IoT applications. Due to the heterogeneity of IoT applications, we also propose a lightweight pre-scheduling algorithm to maximize the number of parallel tasks for concurrent execution. Results demonstrate that our technique significantly improves the weighted cost of executing IoT applications compared to its counterparts by up to 65%.

4.1 Introduction

The number of IoT applications such as smart transportation, smart health-care, augmented reality, and smart buildings requiring large amounts of computing and network resources has dramatically increased [52]. Moreover, execution of such resource-hungry applications requires a considerable amount of energy to be consumed, which signifi-

This chapter is derived from:

- **Mohammad Goudarzi**, Huaming Wu, Marimuthu Palaniswami, and Rajkumar Buyya, "An Application Placement Technique for Concurrent IoT Applications in Edge and Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC)*, Volume 20, Number 4, Pages: 1298-1311, ISSN: 1536-1233, IEEE Press, New York, USA, January 2020.

cantly affects the performance of IoT devices such as mobile devices and sensors, due to their limited battery lifetime.

Since the Edge and Fog computing environments are replete with heterogeneous computing resources, optimized placement of IoT applications with diverse resource requirements on a set of suitable computing resources is an important and yet challenging problem. There are several works that address the placement of a single IoT application at a time in Edge and Fog computing Environments. However, placement of one IoT application can affect the placement of other IoT applications as the amount of resources dynamically change in Edge and Fog computing environments.

Therefore, in this chapter, we propose an efficient batch application placement technique to jointly optimize the execution time of IoT applications and energy consumption of IoT devices in an environment with multiple heterogeneous computing resources (i.e., Edge, Fog, and Cloud servers).

The main **contributions** of this chapter are:

- Proposes a weighted cost model for application placement of multiple IoT devices to minimize the execution time of IoT applications and energy consumption of IoT devices.

- Puts forward a dynamic and lightweight pre-scheduling technique to maximize the number of parallel tasks for execution. Considering the NP-Complete nature of application placement in Fog computing environments, we propose an optimized version of the Memetic Algorithm (MA) to achieve a well-suited solution in a reasonable decision time.

- Proposes a fast failure recovery method to assign failed tasks to appropriate servers in a timely manner.

The rest of the chapter is organized as follows. Relevant works of application placement techniques in Edge and Fog computing environments are discussed in Section 4.2. Section 4.3 presents the system model and problem formulation. Section 4.4 presents our proposed applications placement technique. Section 4.5 reflects the simulation environment and the performance evaluation. Finally, Section 4.6 concludes the chapter.

4.2 Related Work

In this section, related works for application placement techniques in Edge and Fog computing environments are discussed, where Cloud and Fog servers work collaboratively to satisfy the requirements of IoT applications. They are divided into independent and dependent categories based on the dependency model of their IoT applications' constituent parts (e.g., tasks, modules). Each IoT application can be modeled as a set of independent or dependent tasks. The dependent one refers to applications consisted of several dependent tasks so that each new task runs only when its predecessor tasks are completely performed. However, in the independent one, the applications' tasks do not have such constraints for execution.

4.2.1 Independent Tasks

Huang et al. [124] proposed a task placement algorithm where multiple mobile devices offload their independent tasks to multiple Edge servers and one Cloud server. In this technique, each mobile device decides whether each task should be offloaded or not, and in case of offloading, which Edge or Cloud server is suited for the execution of each task. An energy-aware cloudlet selection technique was proposed in [190] to meet the latency requirement of incoming tasks from one IoT device. Haber et al. [129] proposed an offloading algorithm deployed in the Cloud layer, aiming at minimizing the energy consumption of several mobile devices while satisfying the latency requirements of mobile applications. It is obtained by optimizing mobile devices' transmission power and the assigned server computation. An offloading algorithm based on the Lyapunov optimization was proposed in [126] to reduce the execution time of IoT applications by offloading the task to either the single Fog server or one Cloud server. Mahmud et. al. [125] proposed a Quality of Experience (QoE)-aware application placement technique in which independent tasks of IoT devices are placed in the Fog or Cloud servers. Chen et al. [191] considered a multi-user environment with a single computing access point and a remote Cloud server, in which the independent tasks of mobile users can be processed locally, at the computing access point, or the Cloud server. Hong et al. [92] proposed a game-theoretic approach for computation offloading, and a multi-

hop cooperative-messaging mechanism for IoT devices. It considers that each IoT device decides either to forward its single task to the Fog or Cloud server if it has access to wireless networks or to collaborate with other IoT devices that have access to wireless networks for forwarding its task.

4.2.2 Dependent Tasks

In the dependent category, related works modeled their applications by Directed Acyclic Graph (DAG), in which each vertex represents one task of the IoT application, and each edge shows data flow (i.e., dependency) between two tasks.

Neto et al. [102] and Wu et al. [122] proposed a partitioning algorithm for a single mobile device to offload their computation-intensive tasks to a single Edge or Cloud server. The placement engine of these proposals is placed at the mobile device aiming at finding a group of tasks for offloading, by which the execution time of mobile application and energy consumption of mobile device become reduced. The main goal of [192, 193] is to minimize the execution time of IoT applications in an environment in which multiple Fog servers and a Cloud server are accessible for the application placement. Lin et al. [192] considered only one mobile device in its system model for offloading while Stavrinides et al. [193] attempted to place tasks of multiple users requiring low communication overhead at the Cloud server and those tasks that have more communication overhead at the Edge layer. Mahmud et al. [36] proposed a latency-aware application placement policy in an environment with multiple Fog servers and a single Cloud server. Although the above-mentioned techniques consider task placement as their principal objective, Bi et al. [194] proposed a solution for joint optimization of service caching placement and computation offloading.

The proposed placement engines in the above works made application placement decisions for different users at different time-slots, or only consider a fraction of a whole of each user's tasks at each time slot. However, Xu et al. [123] proposed a batch task placement based on a Genetic Algorithm (GA), in which mobile applications of multiple users are forwarded to the single central Edge server for application placement decision.

Table 4.1: A qualitative comparison of related works with ours

Techniques	IoT Application Properties			Architectural Properties								Placement Engine Properties				
	Model	Task Number	Hetero	IoT Device		Edge Layer			Cloud Layer			Position	Batch Placement	Decision Parameters		
				Number	Request Number	Fog Number	Coop	Hetero	Cloud Number	Coop	Hetero			Time	Energy	Weighted
[124]	Independent	Multiple	✓	Multiple	Different	Multiple	×	×	Single	×	×	IoT device	No	✓	✓	✓
[190]		Single	✓	Single	Same	Multiple	×	✓	Single	×	×	Edge Layer		✓	✓	×
[129]		Single	✓	Multiple	Same	Multiple	×	×	Single	×	×	Cloud Layer		✓	✓	✓
[126]		Single	✓	Multiple	-	Single	×	×	Single	×	×	Edge Layer		✓	×	×
[125]		Single	✓	Multiple	Same	Multiple	×	✓	Single	×	×	Edge Layer		✓	×	×
[191]		Multiple	✓	Multiple	Same	Single	×	×	Single	×	×	Edge Layer		✓	✓	✓
[92]		Single	✓	Multiple	Same	Multiple	✓	✓	Single	×	×	Edge Layer		✓	✓	✓
[102]	Dependent	Multiple	✓	Single	Same	Single	×	-	Single	×	×	IoT Device		✓	✓	✓
[122]		Multiple	✓	Single	Same	Single	×	-	Single	×	×	IoT Device		✓	✓	✓
[193]		Multiple	✓	Single	Same	Multiple	✓	✓	Single	×	×	Edge Layer		✓	×	×
[194]		Multiple	✓	Single	Same	Single	×	×	-	×	×	-		✓	✓	✓
[192]		Multiple	✓	Multiple	-	Multiple	×	×	Single	×	×	Edge Layer		✓	×	×
[36]		Multiple	✓	Multiple	Different	Multiple	✓	✓	Single	×	×	Edge Layer		✓	×	×
[123]		Multiple	×	Multiple	Different	Single	×	×	Single	×	×	Edge Layer	Yes	✓	✓	✓
Our Technique		Multiple	✓	Multiple	Different	Multiple	✓	✓	Multiple	✓	✓	Edge Layer		✓	✓	✓

The abbreviated terms are as follows: Hetero: Heterogeneity, Coop: Cooperation

4.2.3 A Qualitative Comparison

Table 4.1 identifies and compares key elements of related works with ours in terms of their IoT application, architectural, and placement engine properties. In the IoT application section, the dependency model of each proposal is studied, which can be either independent or dependent. Moreover, we study how each proposal modeled IoT application in terms of the number of tasks and heterogeneity. This latter study demonstrates whether IoT applications consist of homogeneous or heterogeneous tasks in terms of their computation and data flow. In the architectural section, the attributes of IoT devices, Fog/Edge servers, and Cloud servers are studied. For IoT devices, the overall number of devices and their type of requests are identified. The different request number shows that each device has a different number of requests compared to other IoT devices. In the Fog and Cloud layers, the number of Fog and Cloud servers, the coop-

eration between different Fog/Cloud servers, and the heterogeneity in terms of servers' specifications are identified, respectively. The position of the placement engine, the capability of batch placement, and decision parameters are also studied in the placement engine section.

Considering application placement techniques proposed for Fog computing environments, this chapter proposes a batch application placement technique for an environment consisting of multiple devices in the IoT layer, multiple Fog/Edge servers in the Edge layer, and multiple Cloud servers in the Cloud layer. To the best of our knowledge, this is the only work that considers the aforementioned Fog computing environment and proposes a weighted cost model to jointly minimize the execution time of IoT applications and energy consumption of IoT devices. Our weighted cost model not only can be applied for our general Fog computing environment, but it also can be used for simpler Fog computing environments with a single IoT device, single Fog server, single Cloud server, or any combination thereof. In addition, it is important to note that the IoT applications are considered as heterogeneous DAGs (i.e., workflows) with a different number of tasks and data flows. Hence, we propose a lightweight pre-scheduling algorithm to organize incoming tasks of different DAGs, so that the number of tasks for parallel execution becomes maximized. Then, an optimized version of the MA is proposed to perform application placement in a timely manner.

4.3 System Model and Problem Formulation

We consider a framework consisting of multiple IoT devices, multiple Fog (i.e., Edge) servers, multiple Cloud servers, and brokers, in which IoT devices can locally execute their workflows (i.e., DAGs) or completely/partially place them on Cloud servers and/or Fog servers for execution. Fig. 4.1 represents an overview of our system model.

In this system framework, each broker supports up to N IoT devices, which are distributed in its proximity. The broker (which can be a Fog server) receives workflows from different IoT devices, and periodically makes task placement decisions based on the requirements of IoT applications and the current status of the network. According to the result of application placement decisions, each IoT device understands to which

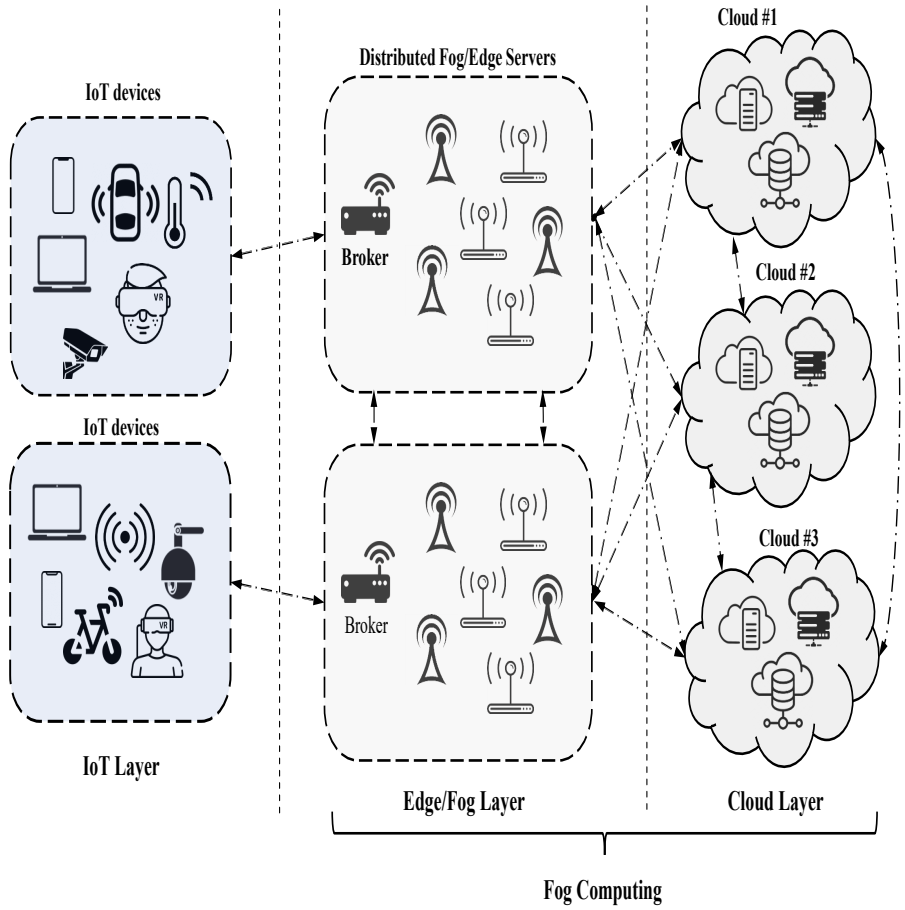


Figure 4.1: An overview of our system model

server each constituent part of its workflow should be sent, or it should be executed locally on the IoT device.

4.3.1 Application Workflow

Each IoT application can be partitioned based on different levels of granularity such as class and task, just to mention a few [195]. Without loss of generality, we represent the application running on the nth IoT device as a DAG (i.e., workflow) of its tasks $G_n = (\mathcal{V}_n, \mathcal{E}_n), \forall n \in \{1, 2, \cdots, N\}$, where $\mathcal{V}_n = \{v_{n,i} | 1 \leq i \leq |\mathcal{V}_n|\}$ denotes the set of tasks running on the nth IoT device, and $\mathcal{E}_n = \{e_{n,i,j} | v_{n,i}, v_{n,j} \in \mathcal{V}_n, i \neq j\}$ illustrates the set of data flows between tasks. As an illustration, $e_{n,i,j}$ represents the dependency

between $v_{n,i}$ and $v_{n,j}$ of the application running on the nth IoT device.

Considering the number of instructions for each task $v_{n,i}$, its corresponding weight is represented as $v^{w}_{n,i}$. Besides, the associated weight of each edge $e^{w}_{n,i,j}$ shows the amount of data that the task $v_{n,j}$ receives as an input from $v_{n,i}$. Since IoT applications are modeled as DAGs, each task $v_{n,i}$ cannot be executed unless all its predecessor tasks, denoted as $\mathcal{P}(v_{n,i})$ finish their execution.

4.3.2 Problem Formulation

We formulate task placement problem as an optimization problem aiming at minimizing the overall execution time of IoT applications and energy consumption of IoT devices.

Since different servers are available to run each task $v_{n,i}$, the set of all available servers is represented as $\mathcal{S}$ with $|\mathcal{S}| = M$. The $\mathcal{S}^{y,z}$ represents one server, in which y represents the type of server (the IoT device ($y = 0$), Fog servers ($y = 1$), Cloud servers ($y = 2$)) and z denotes that server's index. The offloading configuration of the workflow belonging to the nth IoT device is represented as X_n, and $x_{n,i}$ denotes the offloading configuration for each task $v_{n,i}$, which is obtained from the following criteria:

$$x_{n,i} = \begin{cases} 0, & s^{y,z} = s^{0,n}, \\ 1, & s^{y,z} \in \{s^{1,1}, s^{1,2}, \cdots, s^{1,f}\} \quad |z| = f \\ 2, & s^{y,z} \in \{s^{2,1}, s^{2,2}, \cdots, s^{2,c}\}, \quad |z| = c \end{cases} \tag{4.1}$$

where $x_{n,i} = 0$ depicts that the ith task is assigned to the nth IoT device ($s^{0,n}$) for local execution, and $x_{n,i} = 1$ and $x_{n,i} = 2$ denote that the ith task is assigned to one of the Fog servers and Cloud servers, respectively, for the remote execution. Moreover, f and c show the number of available Fog servers and Cloud servers respectively.

Weighted cost model

The goal of the task placement technique is to find the best possible configuration of available servers for each IoT application so that the weighted cost of execution for each

IoT device becomes minimized, as depicted in the following:

$$\min_{\psi_\gamma, \psi_\theta \in [0,1]} \Psi(X_n), \quad \forall n \in \{1, 2, \cdots, N\} \tag{4.2}$$

where

$$\Psi(X_n) = \psi_\gamma \times \frac{\Gamma(X_n)}{\Gamma_{Loc_n}} + \psi_\theta \times \frac{\Theta(X_n)}{\Theta_{Loc_n}} \tag{4.3}$$

s.t.

$$\begin{aligned} &C1: \ VM_{fog,i} \leq C_{fog,i}, \ \forall i \in \{\mathcal{S}^{1,1}, \cdots, \mathcal{S}^{1,f}\} \\ &C2: \ |x_{n,i}| = 1, \ \forall n \in \{1, 2, \cdots, N\}, 1 \leq i \leq |\mathcal{V}_n| \\ &C3: \ \Psi(\mathcal{P}(v_{n,i})) \leq \Psi(\mathcal{P}(v_{n,i}) + v_{n,i}) \end{aligned}$$

where $\Gamma(X_n)$, $\Theta(X_n)$, Γ_{Loc_n}, and Θ_{Loc_n} demonstrate the execution time, energy consumption, local execution time and local energy consumption of the nth IoT device's workflow, respectively. Besides, ψ_γ and ψ_θ are control parameters for execution time and energy consumption, by which the weighted cost model can be tuned according to the users' requirements. Moreover, we assume that each task can be exactly assigned to one Virtual Machine (VM) of one Fog or Cloud server. $C1$ denotes that the number of instantiated VMs of the ith Fog server $VM_{fog,i}$ is less or equal to the maximum capacity of that Fog server $C_{fog,i}$. $C2$ represents that each task i belonging to the workflow of nth IoT device can only be assigned to one server in each time slot. In addition, $C3$ indicates that the predecessor tasks of $v_{n,i}$ should be executed before the execution of the task $v_{n,i}$.

Execution time model

Considering the Eq. 4.3, the weighted cost optimization is equal to the execution time model when $\psi_\gamma = 1$ and $\psi_\theta = 0$.

The goal of execution time optimization model is to find the optimal configuration of the application running on the nth IoT device so that the execution time of that application decreases. The overall execution time of each candidate configuration can be defined as the sum of latency in task offloading ($\Gamma^{lat}_{X_n}$), the computing time of workflow's

tasks based on their assigned servers ($\Gamma^{exe}_{X_n}$) and the data transmission time between each pair of dependent tasks in each workflow ($\Gamma^{tra}_{X_n}$), as depicted in the following:

$$\Gamma(X_n) = \Gamma^{exe}_{X_n} + \Gamma^{lat}_{X_n} + \Gamma^{tra}_{X_n} \tag{4.4}$$

The computing execution time that corresponds to the application running on the nth IoT device is calculated by:

$$\Gamma^{exe}_{X_n} = \sum_{x_{n,i} \in X_n} \gamma^{exe}_{x_{n,i}} \tag{4.5}$$

where $\gamma^{exe}_{x_{n,i}}$ shows the computing time of task $v_{n,i}$, and is calculated based on its corresponding assigned server from the following equation:

$$\gamma^{exe}_{x_{n,i}} = \begin{cases} \frac{v^w_{n,i}}{loc^{cpu}}, & x_{n,i} = 0 \\ \frac{v^w_{n,i}}{SF^f \times loc^{cpu}}, & x_{n,i} = 1 \\ \frac{v^w_{n,i}}{SF^c \times loc^{cpu}}, & x_{n,i} = 2 \end{cases} \tag{4.6}$$

where loc^{cpu} demonstrates the computing power of the IoT device, and SF^f and SF^c denote the speedup factor of Fog servers and Cloud servers, respectively. The offloading latency $\Gamma^{lat}_{X_n}$ of tasks corresponding to the nth IoT device is calculated based on tasks' assigned servers:

$$\Gamma^{lat}_{X_n} = \sum_{x_{n,i} \in X_n} \gamma^{lat}_{x_{n,i}} \tag{4.7}$$

where $\gamma^{lat}_{x_{n,i}}$ illustrates the offloading latency of task $v_{n,i}$, and is calculated according to its corresponding assigned server from the following equation:

$$\gamma^{lat}_{x_{n,i}} = \begin{cases} 0, & x_{n,i} = 0 \\ L_{LAN}, & x_{n,i} = 1 \\ L_{WAN}, & x_{n,i} = 2 \end{cases} \tag{4.8}$$

where L_{LAN} and L_{WAN} correspond to the latency of LAN and WAN respectively. The

tasks' transmission time of the workflow corresponding to the nth IoT device is calculated by:

$$\Gamma_{X_n}^{tra} = \sum_{e_{n,i,j} \in \mathcal{E}_n} \gamma_{e_{n,i,j}}^{tra} \tag{4.9}$$

where the transmission time of each pair of dependent tasks $v_{n,i}$ and $v_{n,j}$ is calculated as follows:

$$\gamma_{e_{n,i,j}}^{tra} = \begin{cases} \frac{e_{n,i,j}^{w}}{B_{LAN}}, & CT_i = CT_1, CT_3 \\ \frac{e_{n,i,j}^{w}}{B_{WAN}}, & CT_i = CT_2, CT_4 \\ 0, & CT_i = CT_5 \end{cases} \tag{4.10}$$

where B_{LAN} and B_{WAN} stand for the bandwidth (i.e., data rate) of LAN and WAN respectively. The CT_i represents possible transmission configuration for each edge $e_{n,i,j}$ according to the assigned servers of its tasks $v_{n,i}$ and $v_{n,j}$ to calculate transmission time.

The possible transmission configurations are defined as:

$$CT_i(e^w_{n,i,j}) = \begin{cases} x_{n,i} \oplus x_{n,j} = 0 \\ \& \quad x_{n,i} = 1 \quad i = 1 \\ \& \, SI(v_{n,i}) \oplus SI(v_{n,j}) \neq 0 \\ \\ x_{n,i} \oplus x_{n,j} = 0 \\ \& \quad x_{n,i} = 2 \quad i = 2 \\ \& \, SI(v_{n,i}) \oplus SI(v_{n,j}) \neq 0 \\ \\ x_{n,i} \oplus x_{n,j} = 1, \quad i = 3 \\ \\ x_{n,i} \oplus x_{n,j} > 1, \quad i = 4 \\ \\ x_{n,i} \oplus x_{n,j} = 0 \\ \& \, SI(v_{n,i}) \oplus SI(v_{n,j}) = 0, \quad i = 5 \end{cases} \tag{4.11}$$

where $\oplus$ is XOR binary operation and $SI(v_{n,i})$ is a function that returns the assigned server's index (i.e., z) of ith task belonging to the nth workflow. CT_1 denotes that the invocation is between two tasks $v_{n,i}$ and $v_{n,j}$ that are assigned to two different Fog servers, and CT_2 represents the configuration in which the two tasks run on two different Cloud servers. The invocation between two tasks assigned to the IoT device and one of Fog server is depicted in CT_3. CT_4 is used to show two different configurations. The first one is whenever the two tasks are assigned to the IoT device and one of the Cloud servers, while the second one illustrates that one task is assigned to one of the Cloud servers and

the second task is assigned to one of the Fog servers. Finally, CT_5 refers to the condition that two tasks are assigned exactly to the same server, for which the transmission time is equal to zero.

Energy consumption model

According to Eq. 4.2, the weighted cost optimization is equal to the energy consumption model when $\psi_\gamma = 0$ and $\psi_\theta = 1$. The energy consumption model aims at finding the best-possible configuration of the application's tasks to minimize the energy consumption of the nth IoT device. The overall energy consumption of each candidate configuration can be defined as the sum of energy consumed in task offloading ($\Theta^{lat}_{X_n}$), the energy consumption for the computing of tasks ($\Theta^{exe}_{X_n}$), and the energy consumed for the data transmission between each pair of dependent tasks ($\Theta^{tra}_{X_n}$) of that application, as depicted in the following:

$$\Theta(X_n) = \Theta^{exe}_{X_n} + \Theta^{lat}_{X_n} + \Theta^{tra}_{X_n} \tag{4.12}$$

The amount of energy consumed to compute the application belonging to the nth IoT device is defined as follows:

$$\Theta^{exe}_{X_n} = \sum_{x_{n,i} \in X_n} \theta^{exe}_{x_{n,i}} \tag{4.13}$$

where $\theta^{exe}_{x_{n,i}}$ represents the energy consumption required to compute the task $v_{n,i}$, as calculated in the following:

$$\theta^{exe}_{x_{n,i}} = \begin{cases} \gamma^{exe}_{x_{n,i}} \times P_{cpu}, & x_{n,i} = 0 \\ \gamma^{idle}_{x_{n,i}} \times P_{idle}, & x_{n,i} = 1, 2 \end{cases} \tag{4.14}$$

where P_{cpu} is the CPU power of the IoT device on which the task $v_{n,i}$ runs. Since we only consider the energy consumption from IoT device perspective, whenever each task is offloaded to the Fog servers ($x_{n,i} = 1$) or Cloud servers ($x_{n,i} = 2$), the respective energy consumption is equal to the idle time of the IoT device $\gamma^{idle}_{x_{n,i}}$ multiplied to the power consumption of that device in its idle mode P_{idle}.

The energy consumed to offload applications' tasks belonging to the nth IoT device

$\Theta^{lat}_{X_n}$ is calculated by:

$$\Theta^{lat}_{X_n} = \sum_{x_{n,i} \in X_n} \theta^{lat}_{x_{n,i}} \tag{4.15}$$

where $\theta^{lat}_{x_{n,i}}$ stands for the offloading energy consumption of the task $v_{n,i}$ and is obtained from:

$$\theta^{lat}_{x_{n,i}} = \begin{cases} 0, & x_{n,i} = 0 \\ \gamma^{lat}_{x_{n,i}} \times P_{idle}, & x_{n,i} = 1, 2 \end{cases} \tag{4.16}$$

The transmission energy consumption $\Theta^{tra}_{X_n}$ corresponding to the nth IoT device is obtained from:

$$\Theta^{tra}_{X_n} = \sum_{x_{n,i} \in X_n} \theta^{tra}_{x_{n,i}} \tag{4.17}$$

where the transmission energy between each pair of dependent tasks $v_{n,i}$ and $v_{n,j}$ is calculated as follows:

$$\theta^{tra}_{e_{n,i,j}} = \begin{cases} \frac{e^w_{n,i,j}}{B_{LAN}} \times P_{transfer}, & CE_i = CE_1 \\ \frac{e^w_{n,i,j}}{B_{WAN}} \times P_{transfer}, & CE_i = CE_2 \\ 0, & CE_i = CE_3 \end{cases} \tag{4.18}$$

where the transmission power of the IoT device is denoted as $P_{transfer}$, and the CE_i shows transmission configuration for each edge $e_{n,i,j}$ based on the assigned servers of its tasks to calculate the transmission energy, which is calculated from:

$$CE_i(e^w_{n,i,j}) = \begin{cases} x_{n,i} \oplus x_{n,j} = 1, & i = 1 \\ x_{n,i} \oplus x_{n,j} = 2, & i = 2 \\ \text{otherwise}, & i = 3 \end{cases} \tag{4.19}$$

where CE_1 denotes that the data flow is between two tasks $v_{n,i}$ and $v_{n,j}$ that are assigned to the IoT device and Fog servers. Moreover, CE_2 is used to represent the invocation

between two tasks that are assigned to IoT device and Cloud servers. Because the energy consumption is considered from the IoT device perspective, the transmission energy consumption is equal to zero whenever one of the participating tasks in each edge $e^{w}_{n,i,j}$ is not assigned to the IoT device, as represented in CE_3.

4.4 Proposed Application Placement Technique

Our application placement technique is divided into three phases: pre-scheduling, batch application placement, and failure recovery. In the pre-scheduling phase, an algorithm is proposed by which brokers can organize the concurrent IoT devices' workflows. Next, we propose an optimized version of Memetic Algorithm (MA) for batch application placement to minimize the weighted cost of each IoT device. Beside, to overcome any potential failures in the runtime, we embed a lightweight failure recovery method in our technique.

4.4.1 Pre-scheduling Phase

The broker receives concurrent workflows from IoT devices in its decision time-slot and organizes them based on their respective dependencies. Also, it calculates the local execution time and energy consumption of IoT devices based on their respective workflows.

Workflows of IoT devices are heterogeneous in terms of the number and weight of tasks, dependencies, and the amount of dataflow between each pair of dependent tasks. Moreover, the order of execution of tasks in each workflow should be sorted so that a new task $v_{n,i}$ cannot be executed unless all tasks in its $\mathcal{P}(v_{n,i})$ finish their execution.

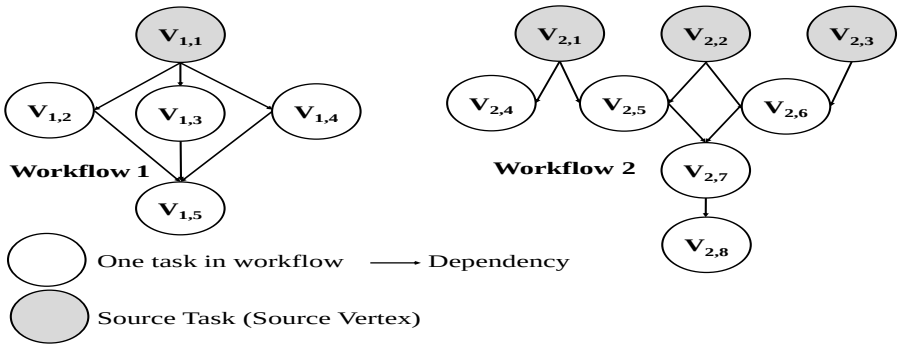
Algorithmic process

Algorithm 4 depicts how the pre-scheduling phase organizes tasks of each workflow and accordingly creates a list of schedules of concurrent workflows. In Algorithm 4, for each workflow, the local execution time and energy consumption are calculated and stored in *LocTime* and *LocEnergy*, respectively (lines 3 and 4). Since DAGs can have several root vertices (i.e., source nodes), the *RootFinder* method finds all the root vertices of

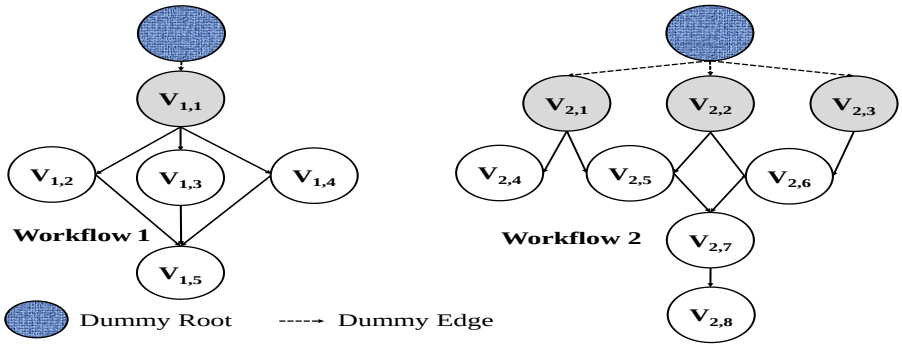
each DAG and stores them in $Source_n$ (line 5). This method checks whether the $\mathcal{P}(v_{n,i})$ is equal to *null* or not for each task i in the nth workflow, and if it equals to *null* returns that task as one source root. The $SingleRootTransformer$ method receives the WF_n and $Source_n$ and creates a new DAG, called $DAG_n^\star$, in which the workflow has only a single source root (line 6). To obtain this, we create a dummy vertex (called $DummyRoot_n$) and connect this vertex to all source vertices of $Source_n$ obtained from the original DAG. This enables us to run Breadth-First-Search (BFS) algorithm over $DAG_n^\star$ starting from the $DummyRoot$, by which we can specify scheduling number for each vertex (i.e., BFS level of each vertex) (line 7). The main outcome of the first loop (lines 2-8) of the algorithm is providing a schedule number for tasks of each workflow by which the concurrent tasks of each workflow are specified. Because our proposed batch application placement algorithm concurrently decides for several workflows at each time slot, it is required to combine these workflows based on their respective schedule number. To achieve this, the algorithm iterates over all workflows so that tasks with same schedule number (either from same or different workflows) are stored in the respective row of a 2D Arraylist called $FinalOrderedList$. The $get(x)$ and $add(v_{n,i})$ methods are used to access a row in the 2D Arraylist (i.e., one schedule), and to add a new entry to a list, respectively (line 12).

Example

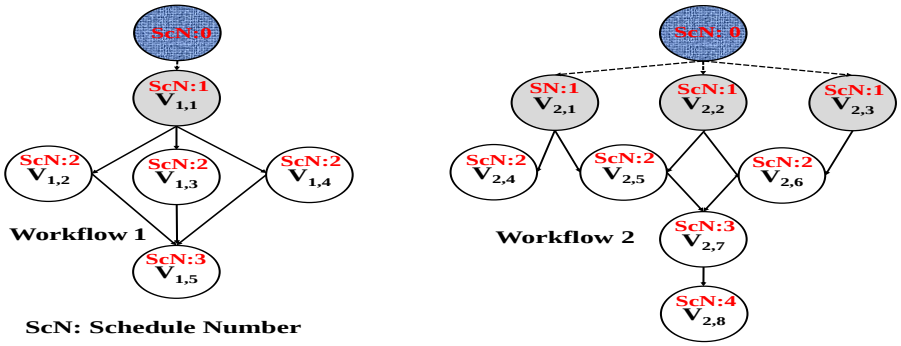
Fig. 4.2 demonstrates how this pre-scheduling phase works. Fig. 4.2a represents two workflows with five and eight vertices. The first workflow has one source vertex while the second workflow has three source vertices (represented by gray color). After identifying the source vertices, the $SingleRootTransformer$ method creates a $DAG_n^\star$ with single source vertex, as depicted in Fig. 4.2b. Next, the BFS algorithm is applied on the $DAG_n^\star$ to specify the schedule number for each task as depicted in Fig. 4.2c. This latter technique helps to identify how many tasks can be executed in parallel in each schedule. When the schedule number of all tasks in all workflows are identified, the tasks with the same schedule numbers are placed together in a 2D Arraylist (called $FinalOrderedList$) as depicted in Fig. 4.2d.



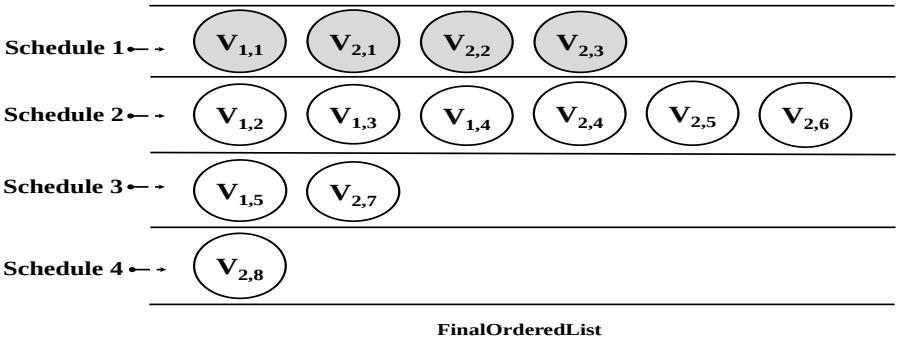
(a) Different workflows with identified source vertices



(b) Transforming workflows to single root $DAG_n^\star$



(c) Assigning schedule numbers to tasks based on BFS



(d) Assigning schedule numbers to tasks based on BFS

Figure 4.2: An example demonstrating the pre-scheduling phase

Algorithm 4: Pre-scheduling phase

Input : WF: List of all workflows
Output : $FinalOrderedList$, $LocTime$, $LocEnergy$
/* N: Number of workflows, WF_n: The nth workflow in the WF, $LocTime$ & $LocEnergy$: Lists storing local execution time and energy consumption of workflows, $FinalOrderedList$: A 2D Arraylist in which tasks in each row can be executed in parallel */

1 $N = |WF|$
2 **for** $n = 1$ to N **do**
3 $LocTime$.add(CalLocalExeTime(WF_n))
4 $LocEnergy$.add(CalLocalExeEnergy(WF_n))
5 $Source_n$ = RootFinder(WF_n)
6 $DAG_n^\star$ = SingleRootTransformer(WF_n, $Source_n$)
7 BFS($DAG_n^\star$, $DummyRoot_n$)
8 **end**
9 **for** $n = 1$ to N **do**
10 **for** $i = 1$ to $|WF_n|$ **do**
11 integer x = CheckOrderNumber($v_{n,i}$)
12 $FinalOrderedList$.get(x).add($v_{n,i}$)
13 **end**
14 **end**

4.4.2 Batch Application Placement Phase

We propose a batch application placement algorithm in which the MA is employed to make placement decisions for tasks of each schedule. Because tasks in each schedule are either independent tasks in one workflow or tasks from different workflows (which do not have any dependency), they can be executed in parallel. An overview of the batch application placement phase is presented in Algorithm 5. This phase receives the list of all workflows WF and schedules $FinalOrderedList$ as an input, and outputs the workflows' configuration $finalConfigs$ and the execution cost of all workflows $finalCost$. Considering the number of schedules, the Application Placement Memetic Algorithm (APMA) is invoked to decide for tasks of the current schedule while considering the server assignments of previous schedules (line 3). Since tasks in each schedule are from one or several workflows, the $ResultProcessor(MAResultList)$ method receives tasks assignments of all schedules $MAResultList$, organize tasks assignments of each

Algorithm 5: Batch task placement phase

Input : WF: The list of all workflows ,$FinalOrderedList$: The 2D Arraylist containing all schedules
Output : $finalConfigs$, $finalCost$

```
/* N: Number of workflows, WF_n: The nth workflow, Q:
   Number of all schedules, MAResultList: A global 2D list
   container in which each row stores the offloading
   configuration of one schedule, finalConfigs: A 2D
   Arraylist container storing obtained severs'
   configuration of each workflow, finalCost: An array to
   store the execution cost of each workflow            */
1 MAResultList = null
2 for i = 1 to Q do
3 |   MAResult.get(i) = APMA(FinalOrderedList.get(i))
4 |   finalConfigs = ResultProcessor(MAResultList.get(i))
5 end
6 for n = 1 to N do
7 |   finalCost[n] = CostCalculator(finalConfigs)
8 end
```

workflow, and stores them in a 2D Arraylist called $finalConfigs$ so that each row represents one workflow (line 4). When task assignment of all schedules is finished, the $CostCalculator(finalConfigs)$ method calculates the execution cost of each workflow based on the respective obtained configuration. As the main function of this phase is the APMA, we illustrate how this algorithm works in detail in what follows.

Application Placement Memetic Algorithm (APMA)

The MA is algorithmic pairing of evolutionary-based search methods such as GA with one or more refinement methods (i.e, local search, individual learning), used for different types of optimization problems such as routing and scheduling [196]. In the MA, each candidate solution is represented by an individual and the solution is extracted from a set of candidate individuals called population.

We propose the APMA based on the GA functions, in which local search is applied to the selected individuals of each iteration. This approach helps the APMA converge faster to the best-possible solution. In the APMA, each candidate configuration of

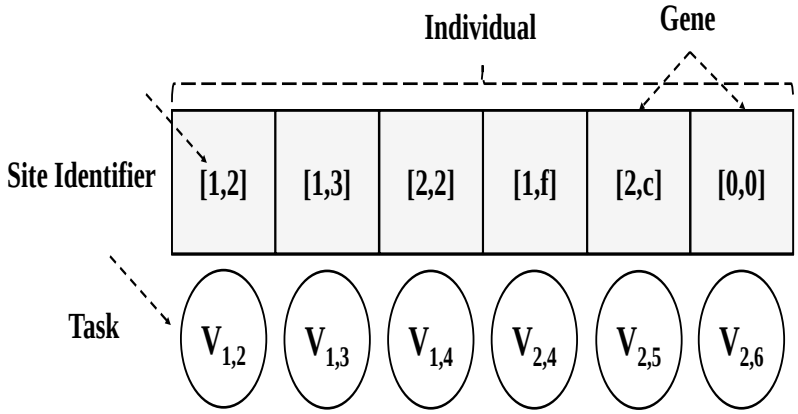


Figure 4.3: An individual representing a sample server configuration for second schedule of Fig. 4.2d

servers assigned to tasks of one schedule is encoded as an individual. The atomic part of each individual is a gene which represents a task in a schedule and carries a tuple (x, y) illustrating the type of assigned server x and the index of that server y. The values for each tuple is derived from the Eq. 4.1 in which values for type and index of servers are defined. Moreover, the length of individuals in each schedule depends on the number of genes (i.e., tasks) on that schedule. A sample individual in our technique is depicted in Fig. 4.3 representing a sample configuration for tasks in the second schedule of Fig. 4.2d.

The APMA is made up of five main steps called initialization, selection, crossover, mutation, and local search. The first four steps are among population-based operations used in GA while the local search step is used as the refinement method. Besides, the utility of each candidate individual is evaluated by a fitness function enabling the APMA to select the best individuals in each iteration. An overview of the APMA is presented in Algorithm 6.

Initialization step

In this step, required parameters for the APMA including the maximum number of iterations I, population size $PopSize$, and individuals in the population are initialized. Moreover, alongside with Original Population (OP), a new population is defined to enhance the diversity of solutions, called Diversity Population (DP). Since the main goal of the APMA is to find the best-possible configuration of servers by which the local ex-

Algorithm 6: An overview of APMA

Input : *scheduleTasks*: A set of tasks for one schedule
Output : $selectedList^{op}.get(0)$
/* I: Maximum iteration number, *selectedList*: The best individuals of respective population found in the in each iteration */
1 $selectedList^{op}$=null; $selectedList^{dp}$=null
2 Initialization(*scheduleTasks*)
3 $selectedList^{op}$=Selection(OP)
4 $selectedList^{dp}$=Selection(DP)
5 **for** i=1 to I **do**
6 Crossover($selectedList^{op}$,$selectedList^{dp}$)
7 Mutation($selectedList^{op}$,$selectedList^{dp}$)
8 LocalSearch($selectedList^{op}$,$selectedList^{dp}$)
9 $selectedList^{op}$ = selection(OP)
10 $selectedList^{dp}$ = selection(DP)
11 **end**

ecution cost decreases, a pre-defined individual is produced for the OP, in which tuple values of all genes are set to their respective local servers (i.e., IoT devices). This reduces the number of low utility individuals because those whose fitness values are worse than the pre-defined individual are not selected in the subsequent iterations. The rest of the individuals in the OP and individuals in the DP are generated randomly in the initialization step.

Fitness function

The APMA uses two global and local fitness functions for OP, which are used to evaluate the utility of each individual $F_g^{op}(indv)$ (representing the utility of a servers' configuration for tasks of one schedule $indv$), and each task of one workflow on that schedule $F_l^{op}(v_{n,i})$ (representing the cumulative utility of the given task plus the utility of other tasks in that workflow), respectively. The $F_l^{op}(v_{n,i})$ receives a task $v_{n,i}$ and calculates the local fitness value based on Eq. 4.2 with the assumption that the execution cost of unassigned tasks in one workflow is equal to zero. Moreover, Algorithm 7 demonstrates how the global fitness of each individual $F_g^{op}(indv)$ is calculated. The $F_g^{op}(indv)$ is the

sum of local fitness $F_l^{op}(v_{n,i})$ of tasks on that schedule. However, due to the parallel execution of multiple tasks of one workflow in each schedule, the maximum of local fitness $F_l^{op}(v_{n,i})$ values of tasks belonging to the same workflow *MaxLoc* are first calculated (line 1-11). The responsibility of finding tasks of the same workflow in one schedule is handled by the *ParallelTaskCheck* method that stores parallel tasks of one workflow in the *parallelSet* (line 3). Then, the local fitness of each task in the *parallelSet* is calculated and the maximum local fitness of tasks belonging to that workflow is stored in *MaxLoc* (line 4-10). Finally, the global fitness value *gBest* can be obtained by summation on all values of *MaxLoc*, which stores the maximum local fitness of each workflow up to that schedule (line 12-14). The principal goal of the diversity population (*DP*) is to diversify

Algorithm 7: Global fitness function of *OP*: F_g^{op}

Input : *indv*: An individual showing tasks of one schedule
Output : *gBest*
/* *WF*: Set of all workflows , *parallelSet* = A container to store parallel tasks of one workflow, *MaxLoc*: A container to store the maximum local fitness of each workflow in the schedule, *gBest*: The global best fitness value, $N = |WF|$ */

```
1  for n=1 to N do
2      parallelSet = null
3      parallelSet = ParallelTaskCheck(indv, WF_n)
4      MaxLoc[n] = F_l^op(parallelSet.get(1))
5      for i=1 to |parallelSet| do
6          tempMax = F_l^op(parallelSet_i)
7          if tempMax > MaxLoc[n] then
8              MaxLoc[n] = tempMax
9          end
10     end
11 end
12 for i=1 to MaxLoc do
13     gBest = gBest + MaxLoc.get(i)
14 end
```

the individuals in the APMA so that the probability of getting stuck in local optimum decreases. Hence, the fitness function of *DP*, $F_g^{dp}(indv)$, is different from the *OP* and is

calculated in what follows:

$$F_g^{dp}(indv_r^{dp}) = \sum_{i=1}^{PopSize} H(indv_i^{op}, indv_r^{dp}) \tag{4.20}$$

where $PopSize$ represents the population size of OP and DP in the APMA. Individual of OP and DP are displayed by $indv_i^{op}$ and $indv_r^{dp}$, respectively. Besides, $H(indv_i^{op}, indv_r^{dp})$ is the Hamming distance function that calculates the difference between individuals received as its arguments in terms of assigned servers to their tasks, and is defined as:

$$H(indv_i^{op}, indv_r^{dp}) = \sum_{k=1}^{f} df \tag{4.21}$$

where f displays the size of that individual (i.e., schedule). In Eqs. 4.20 and 4.21, to calculate the fitness of one individual of DP, we calculate its difference by all individuals in the OP, and the individual with a higher difference receives better fitness value. This helps to maintain individuals with a higher difference in the DP that better diversify the individuals in the APMA. Since different type of servers (i.e., IoT, Fog, and Cloud) with different number of servers in each type (i.e., server index) are considered in the system model, a diversity factor df is defined which describes the fitness of each task according to the type and index of its assigned server. df is obtained from what follows:

$$df = \begin{cases} 2, & sgn(|ST(indv_{i,k}^{op}) - ST(indv_{r,k}^{dp})|) = 1 \\ & \\ 1, & \begin{array}{c} sgn(|ST(indv_{i,k}^{op}) - ST(indv_{r,k}^{dp})|) = 0 \\ \& \\ sgn(|SI(indv_{i,k}^{op}) - SI(indv_{r,k}^{dp})|) = 1 \end{array} \\ & \\ 0, & \begin{array}{c} sgn(|ST(indv_{i,k}^{op}) - ST(indv_{r,k}^{dp})|) = 0 \\ \& \\ sgn(|SI(indv_{i,k}^{op}) - SI(indv_{r,k}^{dp})|) = 0 \end{array} \end{cases} \tag{4.22}$$

where the kth task (i.e., gene) on those individuals are depicted as $indv_{i,k}^{op}$ and $indv_{r,k}^{dp}$, respectively. sgn is the symbolic function, which is defined as:

$$sgn(|x-y|) = \begin{cases} 0, & x = y \\ 1, & x \neq y \end{cases} \tag{4.23}$$

According to Eq. 4.22, if the server type of each task in the DP (i.e., $ST(indv_{r,k}^{dp})$) is different from the server type of corresponding task in an individual of OP (i.e., $ST(indv_{i,k}^{op})$), it receives higher fitness value. However, in condition that the server types of these tasks are equal, the df is set to 1. Moreover, if the two tasks are assigned to exactly one server (i.e., same server type and server index), the fitness value for that task in the DP is equal to zero.

Selection step

The goal of selection is to choose the high utility individuals from both OP and DP based on their respective fitness functions for next iterations. To achieve this, the individuals of OP and DP are sorted based on their respective fitness functions and the top three of individuals plus one random individual from each population are selected and stored in the $selectedList^{op}$ and $selectedList^{dp}$, respectively.

Crossover and Mutation steps

The goal of crossover step is to generate new individuals (called offspring) by a combination of individuals selected in the selection step (called parents). The APMA applies a two-point crossover operation to each pair of selected parents and creates two offspring from them. In each iteration, the total number of new offspring for each population is calculated based on the following equation:

$$offspringNumber = \frac{n!}{(n-k)!}, \quad k = 2 \tag{4.24}$$

In the two-point crossover, two crossover points are randomly selected from the parents. Then, genes in between the two crossover points are exchanged between the parent individuals while the rest remain unchanged. Since the APMA uses two populations OP and DP, the crossover between individuals of each population is called inbreeding, while the crossover between individuals of different populations is called crossbreeding. The crossbreeding provides diversity in individuals which helps to avoid local optimal values with higher probability. Besides, the outcomes of crossbreeding are stored in selected list of both populations $selectedList^{op}$, $selectedList^{dp}$, while the results of inbreedings are only stored in the selected list of respective populations.

In the APMA, the mutation function, based on the pre-defined probability, modifies several genes of offspring in hope of generating individuals with higher utility.

Local search step

Considering the fact that crossover points and genes for the mutation are selected randomly, a new function called local search is defined which works based on the local fitness function of the OP ($F_l^{op}(v_{n,i})$). It is worth mentioning that the randomness provided by the crossover function and mutation is essential since it provides the opportunity to jump out from local optimal points with a higher probability. The local search function, alongside with those random functions, leads to faster convergence to the global optimal solution. Algorithm 8 demonstrates the process of local search step. Although the local search function increases the probability to converge faster to the global optimal solutions, two problems may occur. First, if the local search functions are used solely, the probability of getting stuck in the local optimal points increases. Second, for problems with a large solution space, the local search function requires a significant amount of time to visit the search space. Hence, these two factors should be considered while designing a local search function in the APMA. To address the first issue, the crossover and mutation functions which provide randomness are kept in the APMA. Moreover, the diversity population DP is created which ensures diversity in each iteration. To benefit from the local search function while decreasing its searching time, we reduce the search space for local search by only considering the individuals in the selected list of

Algorithm 8: Local search step

Input : $selectedList^{op}$: Selected list of the OP, $selectedList^{dp}$: Selected list of the DP

/* *tempList*: A temporary list container storing the best-found tuple values for each gene in the individual */

1 $size$=$|selectedList^{op}|$
2 $tempList$=setList(MAXINT)
3 **for** i=1 to $|indv|$ **do**
4 **for** j=1 to $size$ **do**
  % j iterates over $|selectedList^{op}|$
5 **if** $F_l^{op}(indv_{j,i}^{op}) < tempList.get(i)$ **then**
6 $tempList[i]$=$F_l^{op}(indv_{j,i}^{op})$
7 **end**
8 **end**
9 **end**
10 $selectedList^{op}$.add(CreateNewIndv($tempList.get(i)$))
11 UpdatePop(OP,$selectedList^{op}$)
12 UpdatePop(DP,$selectedList^{dp}$)

OP (i.e., $selectedList^{op}$) (line 1). The $setList(MAXINT)$ initializes the $tempList$ with infinite value for all its indexes. Considering individuals in the $selectedList^{op}$, genes with the same index number are evaluated in terms of their local fitness values $F_l^{op}(indv_{j,i}^{op})$ and best genes are selected and stored in the respective index number of $tempList$ (line 3-9). Since the fitness function is defined according to the execution cost, the less fitness value means better assignment (line 5). Afterward, a new individual is created and stored in the $selectedList^{op}$ (line 10). Finally, the updated $selectedList^{op}$ in the local search step and the $selectedList^{dp}$ are then combined with the OP and DP respectively, and top individuals of each population (up to the $PopSize$) are selected for the populations of the next iteration (line 11-12).

Whenever the APMA reaches its stopping criteria, the best individual of the OP stored in $selectedList^{op}.get(0)$ is returned as the result of the APMA.

4.4.3 Failure Recovery Phase

Failures can happen in any systems, and hence, providing an efficient failure recovery method is of paramount importance. In our system, brokers always keep records of free servers and check whether they are planned to perform a task in the near future or not. Besides, considering the assigned server to each task, they estimate the completion cost of each task based on its local fitness value $F_l^{op}(v_{n,i})$. So, if the execution of any tasks fails, the failure recovery method is called to select a surrogate server for that task. The failure recovery method receives the list of current free servers (including IoT devices) and failed task as inputs. Then, it calculates the local fitness value $F_l^{op}(v_{n,i})$ of those tasks for free servers. Finally, tasks will be forwarded to the server with the least $F_l^{op}(v_{n,i})$ for the execution.

4.4.4 Complexity Analysis

The Time Complexity (TC) of our technique depends on its three phases. We consider the number of incoming workflows to the broker as N and the maximum number of tasks for all workflows as L. The most time-consuming part in the pre-scheduling phase (Algorithm 4) is the BFS which requires $O(L+|E|)$ time to visit all tasks of one workflow in which $|E|$ represents the number of data flows. In the dense DAG, the $|E| = O(L^2)$. Hence, the TC of the pre-scheduling phase at the worst case is of $O(N \times L^2)$. In addition, in the best-case scenario, if we assume $N = 1$, and $|E| = O(L)$ for sparse DAGs, the TC is of $O(L)$.

The batch task placement phase (Algorithm 5) calls the APMA (Algorithm 6) Q times where Q represents the number of schedules. To calculate the TC of the second phase, we ignore the iteration size I and the population size $popSize$ of the APMA since they are constant values. In the APMA, the local fitness function $F_l^{op}(v_{n,i})$ and *ParallelTaskCheck* which are invoked from the global fitness function (Algorithm 7) are the most repeated functions, defining the TC of the batch application placement phase. The TC of *ParallelTaskCheck* depends on the size of *indv* which at most can be $N \times (L-1)$ in the case that each workflow has $L-1$ parallel tasks in one schedule. Hence, the TC of *parallelTaskCheck* at the worst case is of $O(Q \times N^2 \times L)$. The maxi-

mum length of *parallelSet* (line 5 of Algorithm 7) is $L-1$, and hence, the local fitness function $F_l^{op}(v_{n,i})$ is called $Q \times N \times (L-1)$ times. Moreover, the instructions in the $F_l^{op}(v_{n,i})$ at most can be executed L times since the local fitness function only considers tasks of one workflow which are at most L. Finally, the TC of the batch task placement phase (Algorithm 5) at the worst case is of $O(Q \times (N \times L^2 + N^2 \times L))$. In addition, in the best-case scenario, if we assume $N = 1$, the TC is of $O(Q \times L^2)$.

The TC of the failure recovery phase depends on the TC of local fitness function $F_l^{op}(v_{n,i})$ which is of $O(L)$, and the number of free servers which at most is equal to all available servers in the system M. Hence the TC of this phase at the worst case is of $O(M \times L)$. In addition, in the best-case scenario, no failure happens in the system.

Considering that in all cases $2 \leq Q$, the TC of our technique at the worst case is polynomial and is represented as $O(Q(NL^2 + N^2L) + ML)$. Besides, in the best-case scenario, where $N = 1$, $Q = 2$, and no failures occur in the system, the TC is of $O(L^2)$.

4.5 Performance Evaluation

In this section, the system setup and parameters, and detailed performance analysis of our technique in comparison to its counterparts (especially [123]) are provided.

4.5.1 System Setup and Parameters

In our experiments, all techniques are implemented and evaluated using iFogSim simulator [26]. We used two types of workflows, namely, real workflows of applications and synthetic workflows. For the real workflows, we used the DAGs extracted from the face recognition application [122] ($Workflow_1$) and the QR code recognition application [197] ($Workflow_2$). Moreover, to consider other possible forms of workflows, several synthetic workflows are generated ($Workflow_3$ to $Workflow_6$). We consider an environment in which six IoT devices are available and each IoT device has one specific workflow from $Workflow_1$ to $Workflow_6$. Each group of six IoT devices is connected to one Fog broker, and Fog brokers have access to six Fog servers and three Cloud servers. In this setup, each Fog server has three VMs while each Cloud server is assumed to

have 16 VMs. The computing power of IoT devices is considered as 500 MIPS [123] and their power consumption in processing and idle states are 0.9W and 0.3W respectively. Besides, the transmission power consumption of IoT devices is 1.3W [198]. We also assume that the computing power of each VM of Fog servers is 6 or 8 times more than IoT devices [123, 199] while the computing power of each VM of Cloud servers are 10 or 12 times more than IoT devices [123]. The summary of our evaluation parameters and their respective values is presented in Table 4.2.

Table 4.2: Evaluation parameters

Evaluation Parameters	**Value**
Number of IoT devices	6
Number of Fog/Edge servers	6
Number of Cloud servers	3
Bandwidth of LAN	(2000,4000) KB/s
Bandwidth of WAN	(500,1000) KB/s
Delay of LAN	0.5 ms
Delay of WAN	30 ms
Computing power of IoT devices	500 MIPS
Speedup Factor of Fog/Edge Servers' VMs	(6, 8)
Speedup Factor of Cloud Servers' VMs	(10, 12)
Idle Power Consumption of IoT device	0.3 W
CPU power of IoT devices	0.9 W
Transmission Power of IoT devices	1.3 W

4.5.2 Performance Study

We employed three quantitative parameters including execution time, energy consumption, and weighted cost to comprehensively study the behavior of our technique in different experiments. Five experiments are conducted to analyze the efficiency of techniques in terms of various bandwidths, different iteration sizes, techniques' decision times, failure recovery, and system size analysis. Both ψ_γ and ψ_θ are set to 0.5 meaning that the importance of execution time and energy consumption is equal in the results. However, these parameters can be adjusted based on the users' requirements and network conditions. To analyze the efficiency of our technique, the following methods are implemented for comparisons:

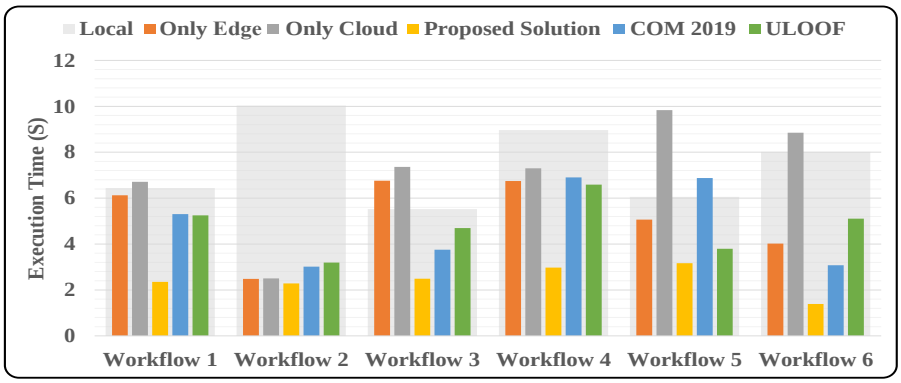
- Local: In this method, all tasks of workflows are executed locally on their respective IoT devices, and hence, no parallel execution of tasks can be performed for workflows. The results of this method can be used as a reference point to analyze the gain of application placement techniques.
- Only Edge: In this method, all tasks of workflows are offloaded to the Fog/Edge servers in the Edge layer for the execution. If the VMs of all servers are full and there is no free VMs, the remaining tasks have to wait until free computing resources become available.
- Only Cloud: In this method, all tasks of workflows are executed on the Cloud servers.
- COM2019: To the best of our knowledge, there is no work considering batch application placement in a scenario with multiple IoT devices, multiple Fog servers, and multiple Cloud servers. Therefore, we updated the fitness function and chromosome structure of the [123], which only consider single Fog server and single Cloud server, to become compatible with our system model. Afterward, the efficiency of its heuristics and searching methods are compared with the other techniques.
- ULOOF: This is the extended version of user level online offloading technique [102], so that it can consider scenarios with multiples Cloud and Fog/Edge server for task placement.

The obtained results of each workflow are the average of 10,000 runs with a 95% confidence interval.

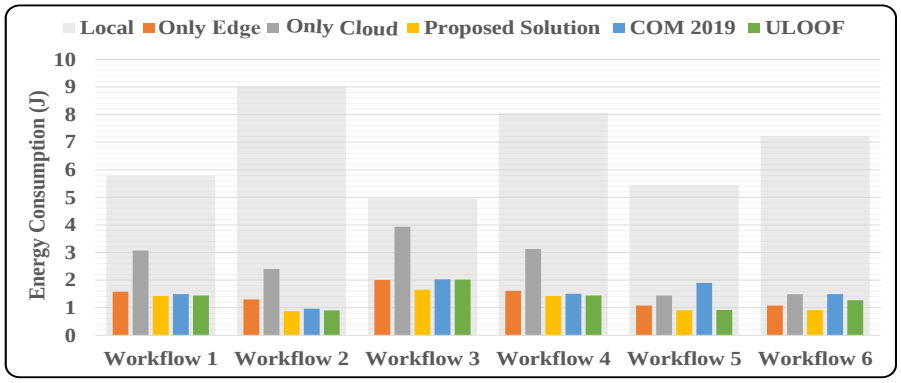
Bandwidth Analysis

In this experiment, we study the behavior of techniques in various bandwidth values as depicted in Fig. 4.4 and Fig. 4.5. The maximum iteration size I and population size $PopSize$ are set to 100 and 20, respectively.

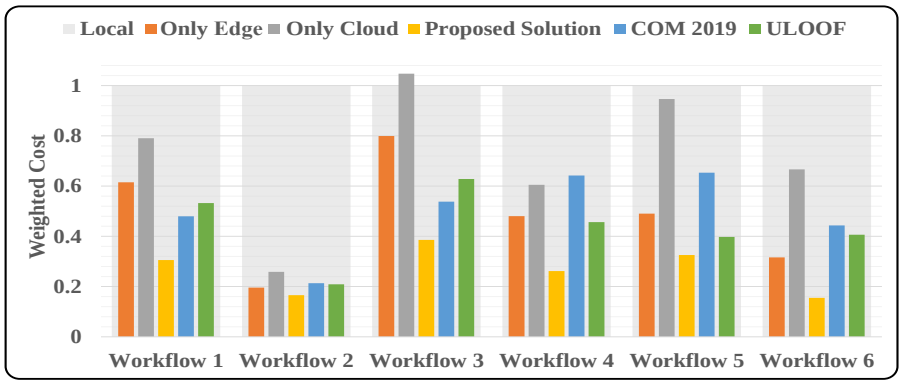
Fig. 4.4 and Fig. 4.5 show that as the bandwidth increases, the execution time, energy consumption, and weighted cost of workflows decrease, meaning better application placement gain in comparison to local execution of workflows. Moreover, in most



(a) Execution time



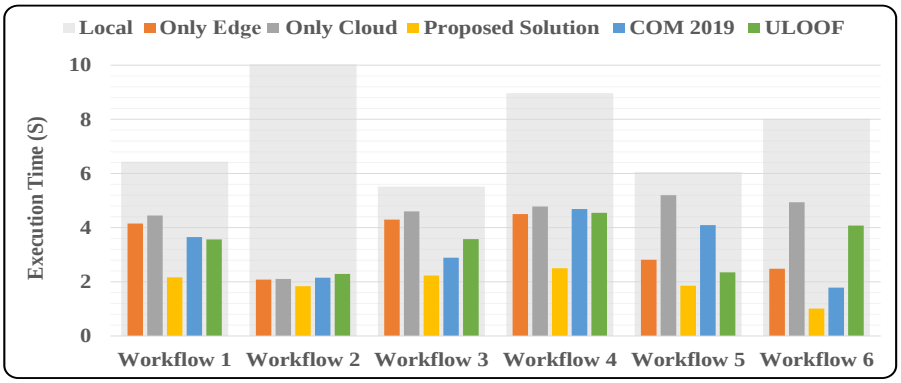
(b) Energy consumption



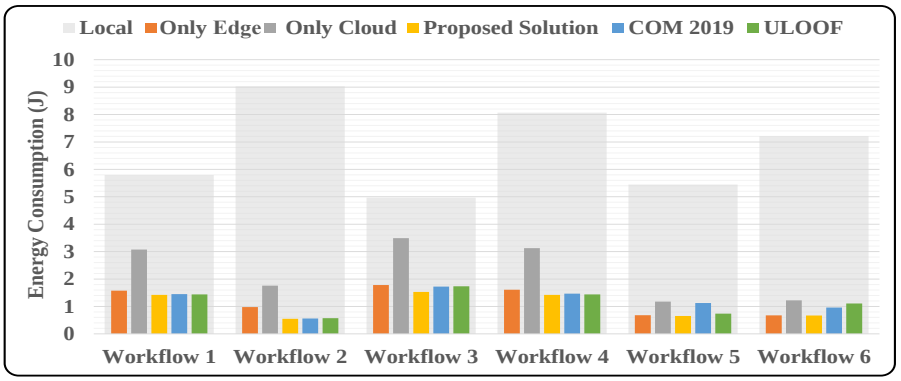
(c) Weighted cost

Figure 4.4: Execution cost of workflows when bandwidth values are (LAN:2000 KB/s, WAN:500 KB/s)

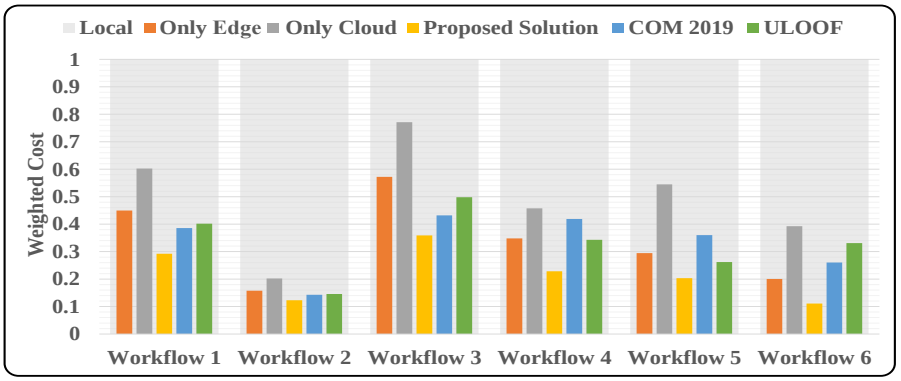
of cases, the only Edge method outperforms the only Cloud because the Fog servers are distributed at the proximity of IoT devices and can be accessed by higher Bandwidth



(a) Execution time



(b) Energy consumption



(c) Weighted cost

Figure 4.5: Execution cost of workflows when bandwidth values are (LAN: 4000 KB/s, WAN: 1000 KB/s)

and less latency. However, since the resources of Fog servers are limited compared to Cloud servers, it cannot obtain the best-possible outcome. This is why the COM2019

and the ULOOF obtain better results in most scenarios than only Cloud and only Edge methods. They use the resources of Cloud and Fog servers simultaneously, resulting in the parallelization of more tasks. As it can be seen, our proposed technique is superior to all other methods due to two important reasons. First, similar to the COM2019 and the ULOOF, it utilizes the resources of Fog and Cloud servers simultaneously. Second, due to its local fitness function, local search, and the diversity provided by the DP, it stays away from local optimal values with higher probability, converges faster to the optimal solution, and hence, outperforms the COM2019 and the ULOOF.

It is worth mentioning that in some cases such as $Workflow_5$ in Fig. 4.5b, the weighted cost of the only Cloud method is less than the local execution, however, its execution time in Fig. 4.4a is far more than the local execution. This is because the ψ_γ and ψ_θ are set to 0.5, which give equal importance to execution time and energy consumption. Therefore, due to lower value for the energy consumption in this workflow compared to its obtained execution time, the weighted cost shows low gain for the task placement.

Maximum iteration number analysis

One of the important parameters for comparing evolutionary application placement techniques is the maximum iteration number, through which their convergence speed to the optimal solution can be evaluated. In this experiment, the performance of COM2019 and our technique are studied. Since the solution of the local execution, only Edge, only Cloud, and ULOOF methods do not change in different iterations, the obtained results of these methods are just depicted to better understand the efficiency of other techniques. For this experiment, the $PopSize$, the LAN, and WAN bandwidths are set to 20, 2000 KB/s and 500 KB/s, respectively.

It can be seen from Fig. 4.6 that the increase in maximum number of iterations I leads to better solutions for both our technique and the COM2019 for all workflows in comparison to the ULOOF, local, only Edge, and only Cloud methods. However, our technique converges to the better solution in a smaller number of iteration compared to the COM2019. The Fig. 4.6a shows that the obtained results of our technique in $I = 50$ for all workflows outperform the obtained results of the COM2019 even at $I = 200$. This

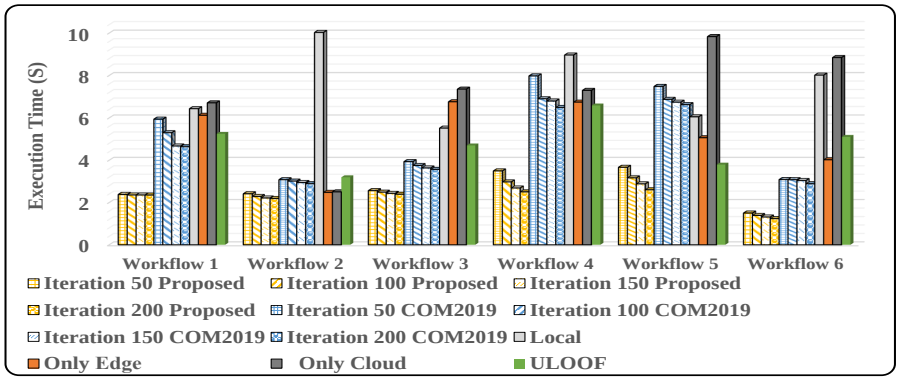
Table 4.3: Decision time analysis

Decision Time	Technique	Workflow Execution Time Result					
		WF1	WF2	WF3	WF4	WF5	WF6
100 ms	Proposed	2.412	2.467	2.758	3.638	3.837	1.649
	COM2019	4.333	2.917	3.422	6.276	6.526	3.09
200 ms	Proposed	2.345	2.397	2.610	3.430	3.384	1.446
	COM2019	4.073	2.707	2.984	5.344	5.109	2.529
300 ms	Proposed	2.288	2.302	2.455	2.869	3.362	1.344
	COM2019	3.656	2.494	2.868	4.388	4.709	2.746
400 ms	Proposed	2.229	2.204	2.403	2.587	2.870	1.304
	COM2019	3.623	2.445	2.753	3.663	4.295	2.523

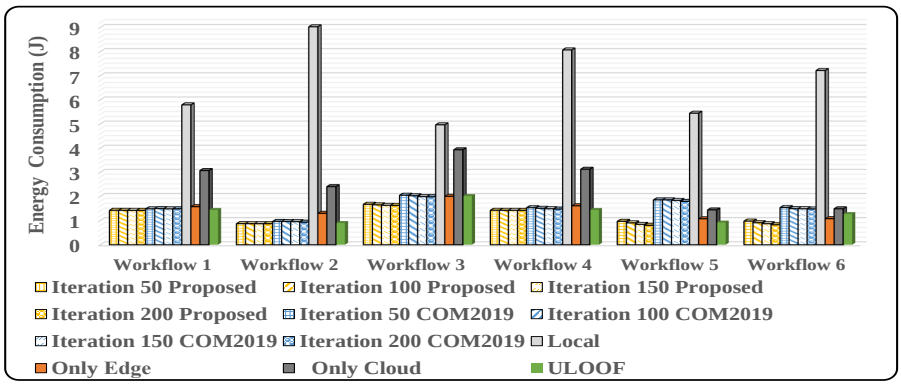
trend can also be seen in Fig. 4.6c for weighted cost of execution, while in Fig. 4.6b the obtained results of the COM2019 and our technique are closer to each other. It is important to note that although better solutions can be found by increasing the maximum number of iterations (if the techniques do not get stuck in the local optimal points), the decision time of algorithms also increases that can be critical for some of workflows, especially for latency-sensitive ones.

Decision time analysis

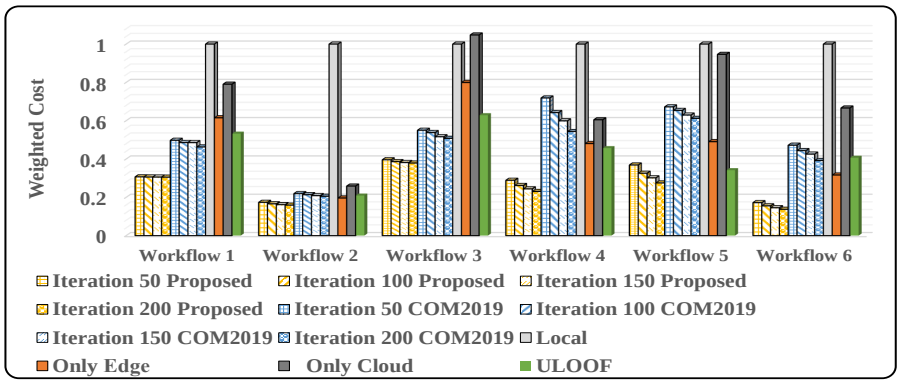
This experiment analyzes the efficiency of each technique based on the decision time required to obtain a well-suited solution. Although application placement algorithms offer server configurations by which the execution time and energy consumption of IoT applications can be reduced, the time that they spend to reach that solution is also important. This is mainly because obtaining good server configurations for IoT applications in a long period of time can negatively affect the execution time requirements of IoT applications. Another important reason elaborating the importance of the decision time analysis, especially for evolutionary algorithms, is that only iteration size analysis cannot solely judge the efficiency of one application placement technique. This is because one technique can reach to better solutions in a small number of iterations compared to its counterparts, however, the time spent on each iteration may be far more than other techniques resulting in longer decision time. Hence, although the maximum iteration



(a) Execution time



(b) Energy consumption



(c) Weighted cost

Figure 4.6: Execution cost of workflows with different maximum iteration number values when (LAN: 2000 KB/s, WAN: 500 KB/s)

size analysis is required, the decision time analysis acts as a supplementary analysis to ensure the efficiency of one technique. In this experiment, the population size *PopSize* is

set to 20, and the LAN and WAN bandwidths are 2000 KB/s and 500 KB/s, respectively.

Table 4.3 represents obtained execution times of our proposed solution and COM2019 for four different decision times. Since the execution time result of the ULOOF does not change in different decision times, its respective results are not presented in Table 4.3, however, its average decision time is roughly 30 ms. As the decision time of techniques increases from 100 ms to 400 ms, the execution time of techniques decreases meaning that the higher utility results are obtained. The obtained results of our solution gradually decrease from 100 ms to 400 ms, while the results of COM2019 has a significant decreasing trend in the range of 100-200 ms and 200-300 ms, and gradually decrease between 300-400 ms, which means that the results of COM2019 approximately converged at 400 ms. It can be clearly seen that our technique not only provides better values compared to the COM2019 in the equivalent decision time, but its results at 100 ms also outperform the results of the COM2019 at 400 ms. This demonstrates that, regardless of number of iterations, our technique converges faster to the optimal solutions.

Failure recovery analysis

This experiment analyzes the effect of failure recovery method in application placement techniques. Since the COM2019 and ULOOF do not have any failure recovery method, we present results of our technique with failure recovery mode (FR Mode) when the probability of failure occurrence is 5% in comparison to the local execution, as depicted in Table 4.4. In this experiment, the maximum iteration size I is equal to 100 and values of the rest of parameters are set as same as parameters in decision time analysis.

Table 4.4 shows that obtained results of our technique with FR mode still outperform results of local execution for all workflows and achieve offloading gain. In techniques ignoring failure recovery in their consideration, failed tasks result in incomplete execution of workflows due to dependencies among tasks of one workflow. However, our technique, by accepting a small overhead of failure recovery phase, can achieve a reasonable gain in comparison to local execution.

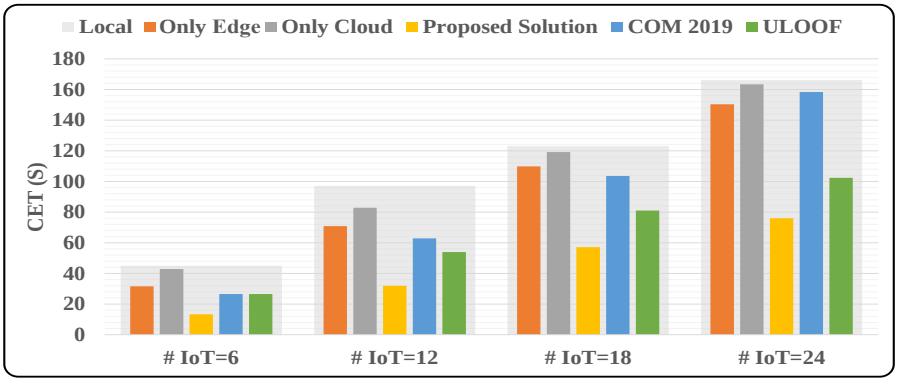
Table 4.4: Failure recovery analysis

Technique	Workflow Execution Time Results					
	WF1	WF2	WF3	WF4	WF5	WF6
Proposed (FR Mode)	2.7132	2.6243	2.8642	3.4125	3.6321	1.4685
Local	6.4354	10.031	5.5194	8.9654	6.0520	8.0180

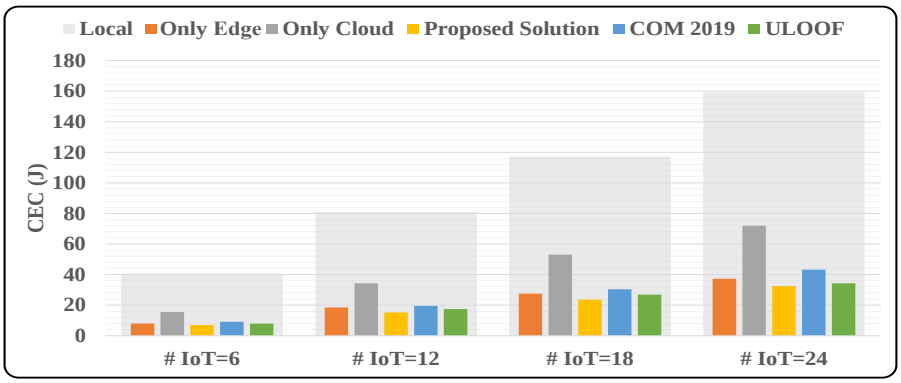
System size analysis

In this experiment, we analyze the effect of system size on different application placement techniques. In our system, each Fog broker makes application placement decisions for its respective IoT devices. Hence, to analyze the performance of our proposed technique, we increase the number of IoT devices and Fog servers per each Fog broker from 6 to 24 by the step of 6. Moreover, in this experiment, we use the same workflows as the previous experiments. In addition, the LAN, and WAN bandwidths are set to 2000 KB/s and 500 KB/s, respectively, and the rest of parameters are as the same as values of Table 4.2.

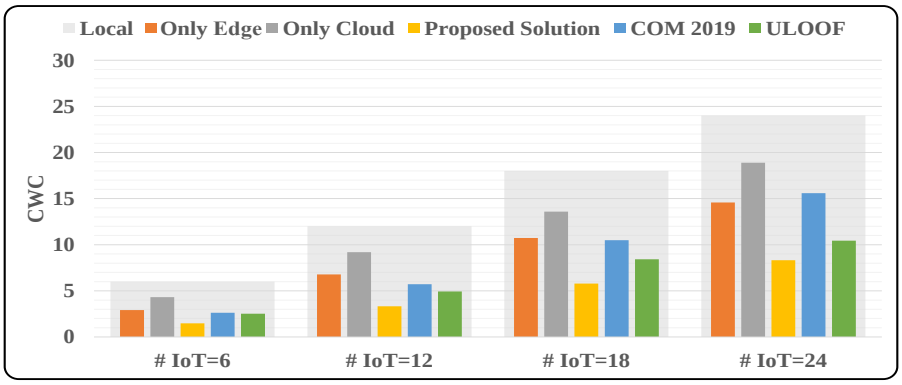
The Fig. 4.7 shows the result of Cumulative Execution Time (CET), Cumulative Energy Consumption (CEC), and Cumulative Weighted Cost (CWC) when different numbers of IoT devices are connected to one Fog broker. The term cumulative refers to the aggregate execution cost of all IoT devices (e.g., the CET shows the aggregate execution time of all IoT devices in scenarios with different number of IoT devices). In Fig. 4.7, the CET, CEC, and CWC increase as the number of IoT devices increases. In all scenarios, the CET, CEC, and CWC of all methods are lower than the local execution cost, however, our proposed technique outperforms other methods in all scenarios and results in lower cost. In addition, the performance of the ULOOF and COM2019 is roughly the same in scenarios with six IoT devices, however the ULOOF shows better performance for the rest of scenarios. This is because ULOOF is independent of maximum number of iteration while the performance of the COM2019 largely depends on the maximum number of iterations.



(a) Cumulative Execution Time (CET)



(b) Cumulative Energy Consumption (CEC)



(c) Cumulative Weighted Cost (CWC)

Figure 4.7: System size analysis with different number of IoT devices per Fog broker

4.6 Summary

We proposed a weighted cost model for optimizing the execution time and energy consumption of IoT devices in a heterogeneous computing environment, in which multiple IoT devices, multiple Fog servers, and multiple Cloud servers are available. We also proposed a batch application placement technique based on the Memetic Algorithm to efficiently place tasks of different workflows on appropriate servers in a timely manner. Besides, a light-weight failure recovery technique is proposed to overcome the potential failures in the execution of tasks in runtime. The effectiveness of our technique is analyzed through extensive experiments and comparisons by the state-of-the-art techniques in the literature. The obtained results demonstrate that our technique improves its counterparts by 65% and 51% in terms of weighted cost in bandwidth analysis and execution time in decision time analysis, respectively. The performance results demonstrate that our technique achieves up to 65% improvement over existing counterparts in terms of the weighted cost.

This chapter proposed a technique for batch placement of IoT applications. In the next chapter, we investigate the scheduling and migration of real-time IoT applications to support their smooth execution for moving IoT devices.

Chapter 5

Real-time Application Placement and Migration Management Techniques

The execution of real-time IoT applications exclusively on one Fog/Edge server may not be always feasible due to limited resources, while the execution of IoT applications on different servers requires further collaboration and management among servers. Moreover, considering user mobility, some modules of each IoT application may require migration to other servers for execution, leading to service interruption and extra execution costs. In this chapter, a new weighted cost model for hierarchical Fog computing environments is proposed to minimize the cost of running IoT applications and potential migrations. Besides, this chapter puts forward a dynamic distributed clustering technique to enable the collaborative execution of application modules. Moreover, application placement and migration management techniques are proposed to minimize the overall cost of executing IoT applications. The performance results show that our technique significantly improves its counterparts in placement deployment time, average execution cost, and the total number of migrations.

5.1 Introduction

Real-time IoT applications can be modeled as a set of lightweight and interdependent application modules in Fog computing environments so that such application modules alongside their allocated resources form the data processing elements of various IoT applications [36, 200]. When the number of IoT applications increases, more requests are

This chapter is derived from:

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Distributed Application Placement and Migration Management Techniques for Edge and Fog Computing Environments", *Proceedings of the 16th Conference on Computer Science and Intelligent Systems (FedCSIS)*, IEEE Press, Pages: 37-56, Online, Poland, September 2-5, 2021.

forwarded to Fog Servers (FSs) that may overload them. Hence, a dynamic application placement technique is required to efficiently place interdependent modules of IoT applications on remote servers while meeting their requirements.

Alongside the importance of suitable application placement techniques, there are yet several issues to be addressed. The coverage ranges of lower-level FSs are limited, and IoT users have different mobility patterns. Besides, interdependent modules of each IoT application may be deployed on several FSs. Hence, as the IoT user moves towards its destination, the application response time and IoT device energy consumption can be negatively affected [81]. Therefore, the migration of interdependent modules of each application among FSs, which incurs service interruption and additional cost, is an important and yet a challenging issue. Several migration techniques decide when, how, and where application modules can migrate when IoT users change their location in the Fog/Edge computing environments, such as [1, 135, 201, 202]. However, these techniques either focus on the migration of a single application module without considering other deployed modules [81] or consider an IoT application as a set of independent application modules. An IoT application may consist of several interdependent modules, and the migration technique should consider the configuration of all interdependent modules when an IoT user moves towards its destination. Hence, the migration of IoT applications, consisting of several interdependent modules, is an important challenge to be addressed, especially in hierarchical Fog computing environments in which modules may be placed on different hierarchical levels.

Also, in Fog computing, there are several studies that assume the application placement and migration management engines (i.e., decision engines) have a global view about topology and resources of all FSs and CSs [136, 203], while there are other studies that assume decision engines only have a local view about resources and topology of servers in their proximity [26, 32, 199]. In these latter techniques, the decision engines act in a distributed manner so that each FS that receives an application placement and/or migration request try to use the available resources in its proximity (which can be accessed with lower latency) to place/migrate the application modules as much as possible. However, if there are no available resources, the rest of the placement and migration will be handled by higher-level FSs in the hierarchy. Considering communi-

cation with higher-level FSs incurs higher latency compared to communication among FSs at the same hierarchical level, the clustering of FSs (if it is possible) at the same hierarchical level can provide sufficient resources (with less latency in comparison to higher-level FSs) to serve real-time IoT applications and reduce the amount of interactions with higher-level FSs.

In this chapter, we address these issues and propose distributed application placement and migration management techniques to satisfy the requirements of real-time IoT applications while users move. The main **contributions** of this chapter are:

- Proposes a new weighted cost model based on IoT applications' response time and IoT devices' energy consumption for application placement and migration of IoT devices in hierarchical Fog/Edge computing environments to minimize cost of running real-time IoT applications.
- Puts forward a dynamic and distributed clustering technique to form clusters of FSs at the same hierarchical levels so that such servers can collaboratively handle IoT application requirements with less execution cost.
- Considering the NP-Complete nature of application placement and migration problems in Fog/Edge computing environments, distributed application placement and migration management techniques are proposed to place/migrate modules of real-time applications on different levels of hierarchical architecture based on their requirements.

The rest of the chapter is organized as follows. In Section 5.2, related researches are reviewed. The system model and problem formulations are presented in Section 5.3. Section 5.4 presents the proposed distributed clustering, application placement, and migration management techniques. The performance of our technique is evaluated in Section 5.5. Finally, Section 5.6 concludes the chapter.

5.2 Related work

In this section, related works that address both application placement and mobility issues at the same time as their main challenges, in the context of Edge/Fog computing,

are studied. These works are categorized into independent and dependent categories based on the dependency mode of their applications' granularity (e.g., modules). In the dependent category, constituent parts of IoT applications (i.e., modules) can be executed only when their predecessor modules complete their execution, while IoT applications that are modeled as a set of independent modules do not have this constraint.

5.2.1 Edge Computing

In the independent category, Wang et al. [87] formulated service migration as a distance-based Markov Decision Process (MDP), which considers the distance between an IoT user and service provider as its main parameter. Then, they proposed a numerical technique to minimize the migration cost of users. Wang et al. [134] and Yang et al [139] considered deterministic mobility conditions, in which the potential paths between source and destination are priori-known, and proposed placement techniques to minimize the application delay. Since paths and available Edge devices are priori-known, as the IoT user moves, the current in-contact Edge device can send the required information to the next Edge device. Ouyang et al. [135] proposed an Edge-centric application placement and mobility management technique that are executed on the network operator and one-hop Edge devices respectively. They proposed a distributed approximation scheme based on the best response update technique to optimize the mobile Edge service performance. Liu et al. [138] proposed a mobility-aware offloading and migration technique to maximize the total revenue of IoT devices by reducing the probability of migration. Zhu et al. [141] proposed a mobility-aware application placement in vehicular scenarios with constraints on service latency and quality loss. In this technique, some of the vehicles generate tasks while other vehicles provide computing services as remote servers. Zhang et al. [83] proposed a deep reinforcement technique to minimize the delay of IoT tasks. Yu et al. [204] proposed a technique to minimize the delay of tasks while satisfying the energy consumption of a single IoT user moving among Edge servers.

In the dependent category, Sun et al. [205] and Qi et al. [140] proposed a mobility-aware application placement technique in which placement decision engines run on IoT devices. The authors of [205] considered a single IoT device and proposed an IoT-

centric energy-aware mobility management technique to minimize the application delay while authors of [140] proposed an Edge-centric and knowledge-driven online learning method to adapt to the environmental changes as vehicles move.

5.2.2 Fog Computing

In the independent category, Wang et al.[202] proposed a solution to place a single service instance of each IoT user on a remote server when multiple IoT users exist in the system. They proposed both offline and online approximation algorithms, performed on the Cloud, to find the optimal and near-optimal solutions respectively. Wang et al. [137] and Wang et al. [81] proposed Edge-centric application placement and mobility management technique when multiple IoT users with a single module exist in the system. The main goal of [137] is maximizing IoT users' gain through offloading and reducing the number of migrations, while the main goal of authors of [81] is minimizing the service delay.

In the dependent category, Shekhar et al. [136] and Bittencourt et al. [199] proposed mobility-aware application placement techniques for IoT application, consisting of multiple interdependent modules while considering prior mobility information. The authors in [136] proposed a Cloud-centric technique, called URMILA, in which the centralized controller makes the placement decision for all IoT applications to satisfy their latency requirements. Besides, whenever the decision is made, even in case the user leaves the range of its immediate server, there is no migration algorithm to migrate modules to new servers, which incurs a higher cost for the users. The authors in [199] proposed an Edge-centric solution based on the edgeward-placement technique [26] for placement of IoT applications while considering their targeted destination. In this work, however, the potential of clustering is not considered. So, whenever the immediate server cannot serve the application modules, the modules are forwarded to the next hierarchical layer for possible placement and migration.

Table 5.1: A qualitative comparison of related works with ours

Techniques	Category	Application Properties			Architectural Properties			Placement and Mobility Management Engines					
		Dependency Model	Module Number	Hetero	IoT Device Number	Hierarchical	Clustering Technique	Placement Engine	Mobility Management Engine	Failure Recovery	Decision Parameters		
											Time	Energy	Weighted
[87]	Edge Computing	Independent	Single	✓	Single	×	×	Edge	Edge	×	✓	×	×
[134]			Multiple	✓	Multiple	×	×	IoT	Edge	×	✓	×	×
[135]			Single	✓	Multiple	×	×	Edge	Edge	×	✓	×	×
[138]			Single	✓	Multiple	×	×	Edge	Edge	×	✓	✓	✓
[139]			Multiple	✓	Multiple	×	×	IoT	Edge	×	✓	×	×
[141]			Multiple	✓	Single	×	×	Edge	Edge	×	✓	×	×
[83]			Single	×	Single	×	×	Edge	Edge	×	✓	×	×
[204]			Multiple	✓	Single	×	×	Edge	Edge	×	✓	✓	×
[205]		Dependent	Multiple	✓	Single	×	×	IoT	IoT	×	✓	✓	×
[140]			Multiple	✓	Multiple	×	×	IoT	Edge	×	✓	×	×
[202]	Fog Computing	Independent	Single	✓	Multiple	×	×	Cloud	Cloud Edge	×	✓	×	×
[137]			Single	✓	Multiple	×	×	Edge	Edge	×	✓	✓	✓
[81]			Single	✓	Multiple	×	×	Edge	Edge	×	✓	×	×
[136]		Dependent	Multiple	×	Single	×	×	Cloud	Cloud	×	✓	×	×
[199]			Multiple	✓	Multiple	✓	×	Edge	Edge	×	✓	×	×
Proposed Solution			Multiple	✓	Multiple	✓	✓	Edge	Edge	✓	✓	✓	✓

The abbreviated terms are as follows: Hetero: Heterogeneity

5.2.3 A Qualitative Comparison

Key elements of related works are identified and presented in Table 5.1 and compared with ours in terms of the main category, IoT application, architectural, and placement and mobility management engines' properties. The IoT application properties identify and compare dependency model mode (either independent or dependent) of IoT applications, modules' number (either single or multiple modules per application), and heterogeneity (whether the specification of modules is same (i.e., homogeneous) or different (i.e., heterogeneous)). Architectural properties contain the number of IoT devices (either single or multiple), whether hierarchical Fog architecture is considered or not, and clustering technique (whether a clustering technique is applied on Edge/Fog servers or not). Placement and mobility management engines contain positions of placement and mobility management engines, failure recovery capability, and the decision parameters used in each work.

Our work proposes an Edge-centric application placement and mobility manage-

ment technique for an environment consisting of multiple IoT devices with heterogeneous applications (consisting of several dependent modules with heterogeneous requirements) and multiple remote servers (either CSs or FSs) deployed in a hierarchical architecture. Considering the potential of the clustering of FSs in the hierarchical Fog computing environment, we propose a weighted cost model of response time and energy consumption for the application placement and migration techniques. The proposed weighted cost model considers the dependency among modules of IoT applications which plays an important role in application placement and migration management. Second, we put forward a distributed and dynamic clustering technique by which FSs of the same hierarchical level can form a cluster and collaboratively provide faster and more efficient service for IoT applications. This latter is because the communication overhead between FSs of the same hierarchical level is usually less than communication with higher-level FSs [36]. Although resources of each lower-level FS is less than each higher-level FS, aggregated resources of lower-level FSs, obtained through clustering, can be used to manage IoT applications modules in lower-level FSs with less response time and energy consumption. Third, we propose a distributed application placement and migration techniques for hierarchical Fog computing environments to minimize the weighted cost of running real-time IoT applications. Finally, due to the highly dynamic nature of such systems, there is a high chance of failures in the system, for which we propose light-weight failure recovery methods in the clustering, application placement, and migration management techniques.

5.3 System Overview

We consider a system consisting of N mobile IoT users (so that each user has one IoT device), F heterogeneous FSs distributed in the proximity of IoT users, and a centralized Cloud. FSs follow a hierarchical topology, in which lower-level FSs can be accessed with lower latency while providing fewer resources in comparison to higher-level FSs that provide more resources but can be accessed with higher latency [32, 36]. Besides, we assume that each IoT device is connected to one FS in the lowest hierarchical level, so that this FS is responsible for the application placement and mobility management of

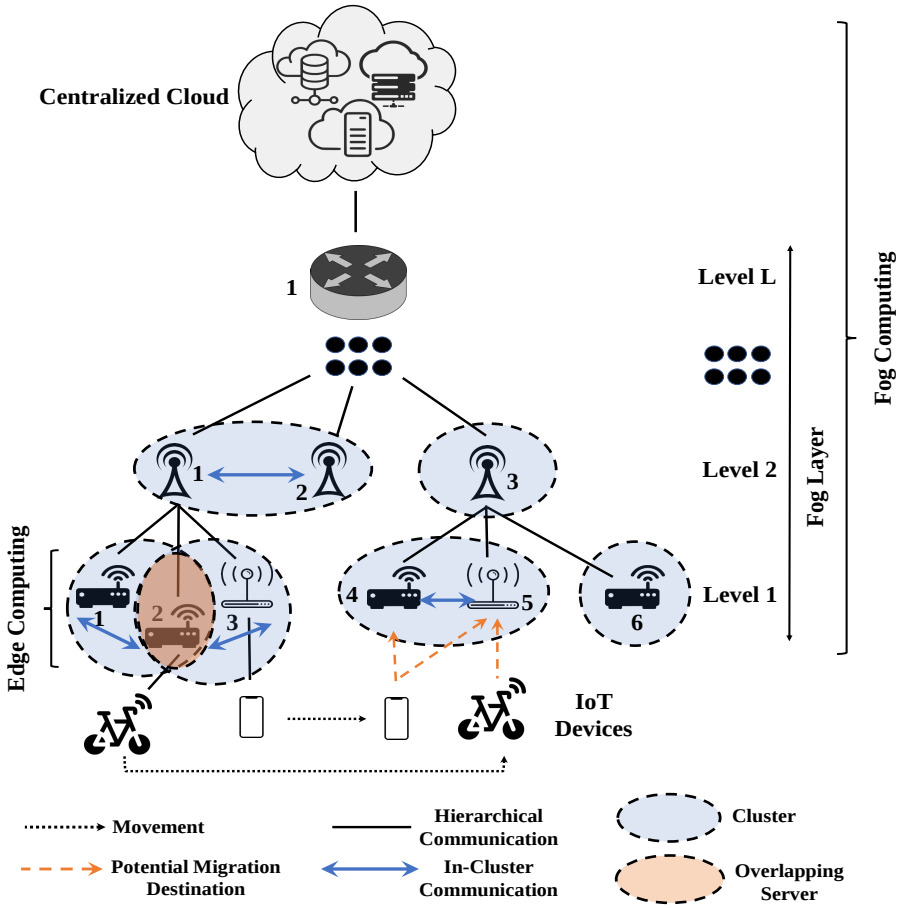


Figure 5.1: A view of our system model

that IoT device. The set of all available servers is represented as $\mathcal{S}$ with $|\mathcal{S}| = M$ and $M > F$. The 2-tuple $(h, i) \in \mathcal{S}$ $(0 \leq h, 1 \leq i)$ represents one server, in which h represents the hierarchical level of the server and i denotes the server's index at that hierarchical level. If we assume there are L hierarchical Fog layers, $(L+1, 1)$ demonstrates the centralized Cloud data-center placed at the top-most level. Moreover, the $(0, n)$ denotes the nth IoT device. Fig. 5.1 represents a view of our system model and how IoT devices move among different FSs. Also, it shows the in-cluster communications (in case clustering is applied) and communications between FSs at different hierarchical levels in this environment.

Each FS can form a cluster either by other nearby FSs at the same hierarchical level or by itself. Moreover, each FS in lth hierarchical level may belong to different clusters in

that hierarchical layer. The cluster member (CM) list of each FS is defined as $List_{cl}(h,i)$, which is empty if the FS (h,i) does not have any CMs. Besides, for each FS, we define a children list, $List_{ch}(h,i)$, containing server specification of immediate lower-level FSs, to which it has direct hierarchical communication links. The sole parent server of each FS is defined as $par(h,i) = (h',i')$ which refers to the immediate higher-level FS. We assume that in-cluster communications are faster than hierarchical communications [36]. Hence, clustering FSs, while incurs additional cost due to running clustering algorithm, can improve the quality of service for IoT users. Moreover, each FS has a list, called $\Omega(h,i)$, containing server specification of itself, its children, and all FSs belonging to the Ω of its children. To illustrate, considering Fig. 5.1, the $\Omega(2,1) = \{(2,1),(1,1),(1,2),(1,3)\}$ and $List_{ch}(2,1) = \{(1,1),(1,2),(1,3)\}$, and $\Omega(2,2) = \{(2,2)\}$ and $List_{ch}(2,2) = \{\}$. If we assume the maximum number of Fog layers is three (i.e., $L = 3$) in this example, then $\Omega(3,1) = \{(3,1),(2,1),(2,2),(2,3),(1,1),(1,2),(1,3),(1,4),(1,5),(1,6)\}$, and the $List_{ch}(3,1) = \{(2,1),(2,2),(2,3)\}$.

We consider that FSs and CSs use container technology to run IoT applications' modules [81, 150]. So, we assume that FSs have access to images of all containers ($Cnts$) while such $Cnts$ may be active if they are running on the server or inactive (i.e., the container images are accessible, but the containers are not running) otherwise [81]. Moreover, for each container, according to the application module that it serves, an amount of ram size at the runtime is assigned to keep the state $Cnt^{ram}_{v_{n,j}}$ [206]. Table 5.2 summarizes the parameters used in this chapter and their respective definitions.

5.3.1 Application Model

We consider real-time IoT applications working based on the Sense-Process-Actuate model, in which sensors transmit tasks periodically according to their sample rate [26, 36]. The emitted sensors' tasks should be forwarded to different modules of the IoT applications for processing based on dependency model among constituent modules. When each module receives tasks from predecessor modules as input, it processes tasks and produces respective tasks as its output to be forwarded to next modules [36]. Finally, results will be forwarded to the actuator as the last module. In this work, we

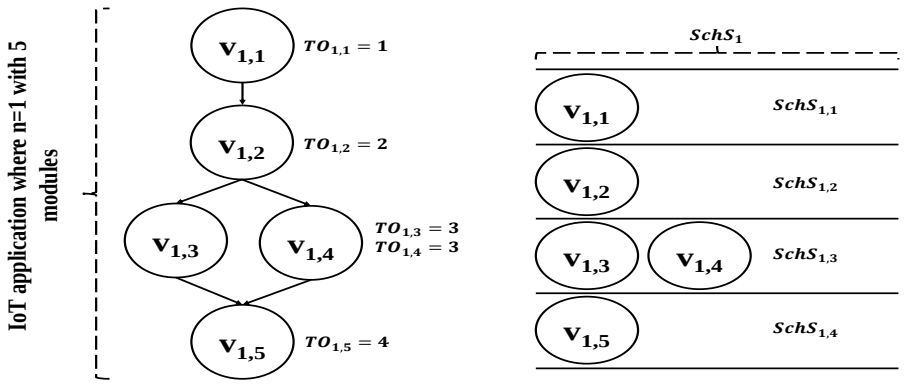
Table 5.2: Parameters and respective definitions

Parameter	Definition	Parameter	Definition
CSs	Cloud Servers	FSs, FS	Fog Servers, Fog Server
$CNTs$, CNT	Containers, Container	N	Number of mobile IoT devices
F	Number of heterogeneous Fog servers (FSs)	$\mathcal{S}$	The set of all available servers
M	Number of available servers	(h,i)	The 2-tuple showing one server in which h represents the hierarchical level of the server and i denotes the server's index at that hierarchical level
$List_{ch}(h,i)$	The list containing server specification of children for the server (h,i)	$par(h,i)$	The sole parent of the server (h,i) in the hierarchical system
$\Omega(h,i)$	The set containing server specification of server (h,i), its children, and all FSs belonging to the Ω of its children	CM	Cluster Member
$List_{cl}(h,i)$	The list containing server specification of cluster members for the server (h,i)	G_n	Directed Acyclic Graph (DAG) of the nth IoT application
$\mathcal{V}_n$	The set of modules belonging to the nth IoT application	$\mathcal{E}_n$	The set of data flows between modules belonging to the nth IoT application
$v_{n,i}$, $v_{n,j}$	The ith and/or jth module belonging to the nth IoT application	$e_{n,i,j}$	The data flow from module $v_{n,i}$ to module $v_{n,j}$ of the nth IoT device
$\mathcal{P}(v_{n,j})$	The set of predecessor modules of the module $v_{n,j}$	$TO_{n,i}=t$	The topological order of ith module of the nth IoT application is equalt to t
$SchS_n$	The schedule set of the nth IoT application consisting of subsets of modules with the same TO value t	$SchSn,t$	A subset of $SchS_n$ showing modules with the same TO value t (i.e., modules that can be executed in parallel)
$e^{ins}_{n,i,j}$	The amount of instructions in terms of Million Instruction that the module $v_{n,i}$ receives from $v_{n,j}$ for processing	$e^{dsize}_{n,i,j}$	The size of data that the module $v_{n,i}$ generates as an output to be sent to module $v_{n,j}$
$v^{mtd}_{n,i}$	The maximum tolerable delay for the module $v_{n,i}$	X_n	The placement configuration of the nth IoT application
$x_{n,i}$	The placement configuration for each module $v_{n,i}$ of the nth IoT application in the X_n	$\Psi(X_n,t)$	The weighted cost of modules in the tth schedule while considering the placement configuration X_n.
$\lvert SchS_n\rvert$	The number of schedules for the nth IoT application	$T_{x_{n,j}}$	The overall delay of each module (i.e., $v_{n,j}$) based on its assigned server
$Cnts_{(h,i)}$	The number of instantiated $Cnts$ on the server (h,i)	$Cap_{(h,i)}$	The maximum capacity of server (h,i) to instantiate $Cnts$.
$\Gamma(X_n,t)$	The weighted cost of modules in the tth schedule while considering the placement configuration X_n	$\Theta(X_n,t)$	The energy consumption of modules in the tth schedule while considering the placement configuration X_n
$T^{lat}_{x_{n,j}}$	The inter-nodal latency between the servers on which module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ are placed	$T^{exe}_{x_{n,j}}$	The computing execution time of tasks, emitted from the $v_{n,j}$ to be executed on the $v_{n,j}$
$T^{tra}_{x_{n,j}}$	The transmission time between between the module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$	$cpu(x_{n,j})$	The computing power of the assigned server (in terms of MIPS) for the module $v_{n,j}$
γ^{tra}	The transmission time between source and destination servers	B_{up}, B_{down}, $B_{cluster}$	The bandwidth of the one server to the parent server, to the child server, and to its CMs, respectively
$NST_i(H)$, $NSE_i(H)$	They define the next intermediate server to reach the destination server	$chRule$	It identifies whether any children of the current server has a route to the destination server or not
$chRule$	It identifies whether any CMs of the current server has a route to the destination server or not	$\Upsilon((\Omega(h,i)),(h',i'))$	It shows whether $\Omega(h,i)$ contains (h',i') or not (i.e., meaning that there is one hierarchical path from (h,i) to the (h',i'))
γ^{lat}	The inter-nodal latency between source and destination servers	$lat(up)$, $lat(down)$, $lat(cluster)$	The inter-nodal latency of one server to the parent server, to the child server, and to its CMs, respectively
$\Psi^{mig}((X_n,X'_n,ts)$	The weighted migration cost of nth IoT application from the current configuration X_n to the new configuration X'_n	$\gamma^{mig}(x_{n,i},x'_{n,j})$	The migration cost of one module from current configuration $x_{n,i}$ to the new configuration $x'_{n,i}$
$\gamma^{lat}_{mig}((h,i),(h',i'))$	The migration latency between current and new servers	$dsize^{mig}$	The size of dump data and states that should be transferred between current and new servers
$e^{ins,r}_{n,i,j}$	The amount of remaining instructions of task $e^{ins,r}_{n,i,j}$ to be executed on the new server after migration	$E(x_{n,j})$	The overall energy consumption of each module (i.e., $v_{n,j}$) based on its assigned server
$E^{exe}_{x_{n,j}}$	The computing energy consumption of tasks, emitted from the $v_{n,i}$to be executed on the $v_{n,j}$	$E^{lat}_{x_{n,j}}$	The energy consumption incurred due to inter-nodal latency between the servers on which module $v_{n,j}$and its predecessors $\mathcal{P}(v_{n,j})$ are placed
$E^{tra}_{x_{n,j}}$	The transmission energy consumption between between the module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$	P_{cpu}, P_i, P_t	The CPU power of the IoT device, the idle power of IoT device, and transmission power of the IoT device
θ^{tra}	The transmission energy consumption between source and destination servers	θ^{lat}	The energy consumption incurred due to inter-nodal latency between servers
$\Gamma^{mig}((X_n,X'_n),t)$	The migration time of nth IoT application from the current configuration X_n to the new configuration X'_n considering schedule t	$\Theta^{mig}((X_n,X'_n),t)$	The migration energy consumption of nth IoT application from the current configuration X_n to the new configuration X'_n considering schedule t

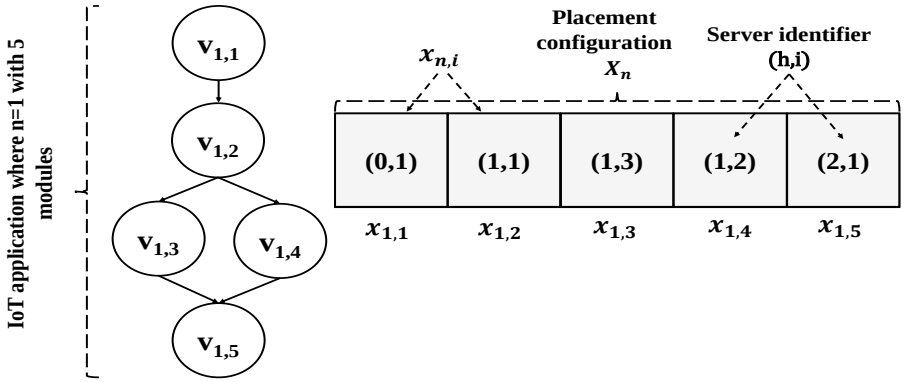
assume that both sensor and actuator modules of IoT applications reside in IoT devices [199].

Real-time IoT application belonging to the nth IoT device is represented as a Directed Acyclic Graph (DAG) of its modules $G_n = (\mathcal{V}_n, \mathcal{E}_n), \forall n \in \{1,2,\cdots,N\}$, where $\mathcal{V}_n = \{v_{n,i} | 1 \leq i \leq |\mathcal{V}_n|\}$ denotes the set of modules belonging to the nth IoT device, and $\mathcal{E}_n = \{e_{n,i,j} | v_{n,i}, v_{n,j} \in \mathcal{V}_n,\ v_{n,i} \in \mathcal{P}(v_{n,j}),\ i \neq j\}$ shows the set of data flows between

modules. Since IoT applications are modeled as DAGs, each module $v_{n,j}$ cannot be executed unless all its predecessor modules, denoted as $\mathcal{P}(v_{n,j})$, finish their execution. To illustrate, $e_{1,1,2}$ represents that execution of module $v_{1,2}$ depends on the execution of the module $v_{1,1}$. Moreover, we define a Topological Order value t for each module i of the nth IoT application as $TO_{n,i} = t$. We define a schedule set for the nth IoT application, called $SchS_n$, consisting of modules with the same TO value t as its subsets. The $SchS_{n,t}$ specify modules with the same TO value t (i.e., modules that can be executed in parallel). In addition, the set of successor modules of module $v_{n,j}$ is defined as $Succ(v_{n,j})$. Fig 5.2a shows an IoT application, the TO value for each module, and the schedule set $SchS_n$ based on the TO values of its modules.



(a) An IoT application and its corresponding schedules



(b) A candidate server configuration for the IoT application

Figure 5.2: An example of IoT application, its schedules and a candidate configuration

Besides, we define the output of each module $v_{n,i}$ as a task consisting of two values to be forwarded to next modules based on data flows of the IoT application. The first

value is the amount of instructions in terms of Million Instruction (MI) that the module $v_{n,j}$ receives from $v_{n,i}$ for processing, shown as $e^{ins}_{n,i,j}$, and the second value is the size of data $e^{dsize}_{n,i,j}$ the module $v_{n,i}$ generates as its output to be forwarded to module $v_{n,j}$ [26].

5.3.2 Problem Formulation

The placement configuration of the application belonging to the nth IoT application is shown as X_n. Also, $x_{n,i} = (h, i)$ denotes the placement configuration for each module $v_{n,i}$ of the nth IoT application in the X_n based on the specification of the server. To illustrate, $x_{n,i} = (1, 3)$ shows that the ith module of nth IoT device is assigned to a server in the first hierarchical level where the server index is 3. Moreover, if the ith module of the nth IoT device is assigned to run locally on itself, $x_{n,i} = (0, n)$. Fig 5.2b presents a sample DAG of an IoT application and a candidate placement configuration.

Placement weighted cost model

The goal of application placement is to find a suitable configuration for modules of each real-time IoT application to minimize the weighted cost $\Psi(X_n, t)$ of running applications in terms of the response time of tasks and energy consumption of IoT devices:

$$\min_{w_1, w_2 \in [0,1]} \sum_{t=1}^{|SchS_n|} \Psi(X_n, t), \quad \forall n \in \{1, 2, \cdots, N\} \tag{5.1}$$

where

$$\Psi(X_n, t) = w_1 \times \Gamma(X_n, t) + w_2 \times \Theta(X_n, t) \tag{5.2}$$

$$\begin{aligned} s.t. \quad & C1: \ Size(x_{n,j}) = 1, \ \forall x_{n,j} \in X_n \ , \quad n \in \{1, 2, \cdots, N\}, 1 \leq i \leq |\mathcal{V}_n| \\ & C2: \ Cnts(h, i) \leq Cap(h, i), \ \forall \ (h, i) \in \mathcal{S} \\ & C3: \ \Psi(x_{n,i}, t) \leq \Psi(x_{n,j}, t), \ \forall v_{n,i} \in \mathcal{P}(v_{n,j}) \end{aligned}$$

where $|SchS_n|$ represents the number of schedules, and $\Gamma(X_n, t)$ and $\Theta(X_n, t)$ show the response time model and energy consumption model, respectively, of modules in the

tth schedule while considering the placement configuration X_n. Moreover, w_1 and w_2 are control parameters to tune the weighted cost model according to user requirements. We assume the number of available servers M is more than or equal to the maximum number of modules in the tth schedule for parallel execution (i.e., $|SchS_{n,t}| \leq M$). We suppose that each module of an IoT application can be exactly assigned to one Cnt of one remote server. $C1$ indicates that each module i of the nth IoT application can only be assigned to one server at a time, and hence the size of $x_{n,j}$ is equal to 1 [3, 123]. $C2$ denotes that the number of instantiated $Cnts$ on the server (h, i) is less or equal to the maximum capacity of that server $Cap(h, i)$. Besides, $C3$ guarantees that the predecessor modules of $v_{n,j}$ (i.e., $\mathcal{P}(v_{n,j})$) are executed before the execution of module $v_{n,j}$ [123].

Response time model. The goal of this model is to find the best possible configuration of servers for each IoT application so that the overall response time for each IoT application becomes minimized. In order to only consider response time model as the main objective, the control parameters of weighted cost model (Eq. 5.2) can be set to $w_1 = 1$ and $w_2 = 0$.

$$\Gamma(X_n, t) = \begin{cases} T(x_{n,j}), & \text{if } |SchS_{n,t}| = 1 \quad \text{(a)} \\ max(T(x_{n,j})), & \text{otherwise} \\ & \text{(b)} \\ \forall x_{n,j} \in X_n | v_{n,j} \in SchS_{n,t} \end{cases} \tag{5.3}$$

The Eq. 5.3.a represents the condition in which the number of modules in the tth schedule is one (i.e, $|SchS_{n,t}| = 1$), and hence, the time of that schedule is equal to the time of that module based on its assigned server $T(x_{n,j})$. Besides, the Eq. 5.3.b refers to the condition in which the number of modules in the tth schedule is more than one (i.e., several modules can be executed in parallel). In this latter case, the time of the tth schedule is equal to the maximum time of all modules that can be executed in parallel.

The overall delay of each module (i.e., $v_{n,j}$) based on its candidate configuration (i.e., $x_{n,j}$) is defined as the sum of inter-nodal latency between servers ($T^{lat}_{x_{n,j}}$), the computing

time per module ($T^{exe}_{x_{n,j}}$), and the data transmission time between $v_{n,j}$ and all of its predecessor modules ($T^{tra}_{x_{n,j}}$). It is formulated as:

$$T(x_{n,j}) = T^{exe}_{x_{n,j}} + T^{lat}_{x_{n,j}} + T^{tra}_{x_{n,j}} \tag{5.4}$$

The computing execution time of module $v_{n,j}$ depends on tasks emitted from its predecessors (i.e., $\mathcal{P}(v_{n,j})$) for processing by $v_{n,j}$. The computing time of $v_{n,j}$ is estimated as:

$$T^{exe}_{x_{n,j}} = \sum \frac{e^{ins}_{n,i,j}}{cpu(x_{n,j})}, \quad \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}) \tag{5.5}$$

where $cpu(x_{n,j})$ shows the computing power of the assigned server (in terms of Million Instruction per Second (MIPS)) for the module $v_{n,j}$. Moreover, the $e^{ins}_{n,i,j}$ shows the amount of instructions in terms of MI that the module $v_{n,j}$ receives from $v_{n,i}$ for the processing. The transmission time between module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ of the application belonging to the nth IoT device is calculated as:

$$T^{tra}_{x_{n,j}} = \max(\gamma^{tra}(e^{dsize}_{n,i,j}, (h,i), (h',i'))), \tag{5.6}$$
$$\forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \quad x_{n,i} = (h,i), x_{n,j} = (h',i')$$

Due to the hierarchical nature of Fog computing, the transmission time of one task (γ^{tra}) between each pair of dependent modules $v_{n,i}$ and $v_{n,j}$ is recursively obtained based on visited servers between source and destination. The (h,i) and (h',i') show server specifications of source and destination servers on which modules $v_{n,i}$ and $v_{n,j}$ are assigned, respectively. By visiting each intermediate server between source and destination servers, the value of source server (h,i) is updated while the value of destination server remains

unchanged. To reduce the length of equations, we consider $(e^{dsize}_{n,i,j}, (h,i), (h',i')) = H$.

$$\gamma^{tra}(H) = \begin{cases} \frac{e^{dsize}_{n,i,j}}{B_{up}} + \gamma^{tra}(H'), & NST_i(H) = NST_1 | NST_4 | NST_6 \\ \frac{e^{dsize}_{n,i,j}}{B_{down}} + \gamma^{tra}(H'), & NST_i(H) = NST_2 \\ \frac{e^{dsize}_{n,i,j}}{B_{cluster}} + \gamma^{tra}(H'), & NST_i(H) = NST_3 | NST_5 \\ 0, & NST_i(H) = NST_7 \end{cases} \tag{5.7}$$

where B_{up}, B_{down}, and $B_{cluster}$ refer to the bandwidth of current server to parent server, to child server, and to cluster server, respectively. Besides, H' is defined as what follows:

$$H' = (e^{dsize}_{n,i,j}, (h^{''}, i^{''}), (h',i')) \tag{5.8}$$

$$(h^{''}, i^{''}) = NST_i(H) \tag{5.9}$$

The Eq. 5.8 shows the data size and destination server of H' is exactly the same as H, and the only difference is the specification of the source server $(h^{''}, i^{''})$ which is obtained from the output of $NST_i(H)$ (i.e., $(h^{''}, i^{''}) = NST_i(H)$. The $NST_i(H)$ defines the next

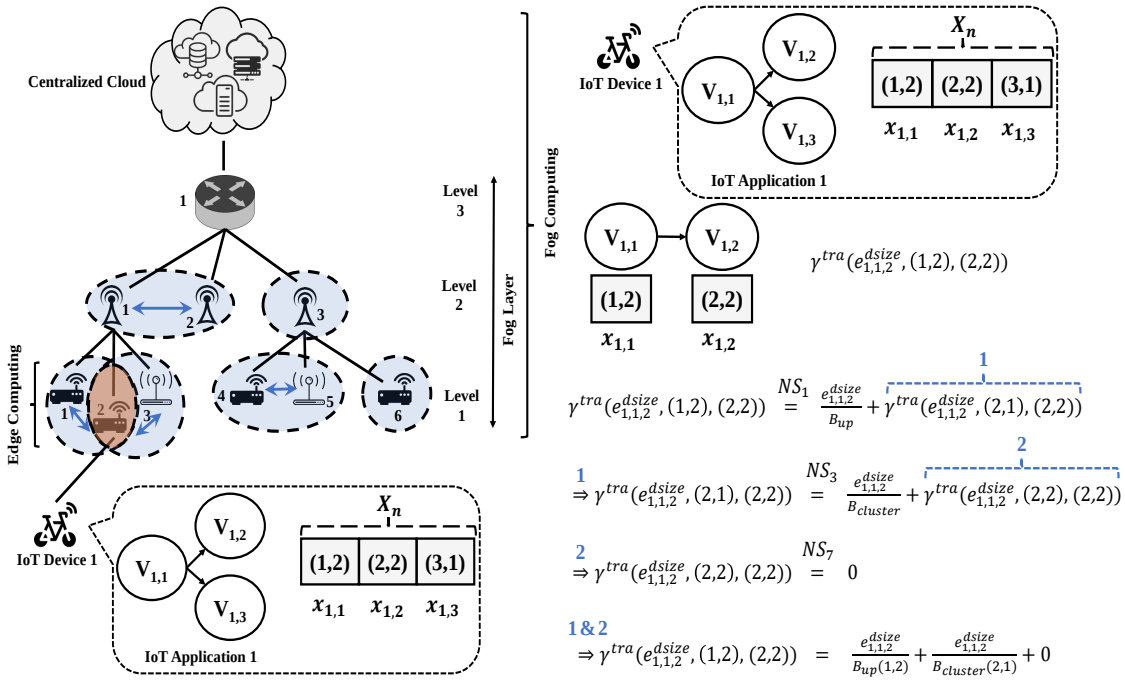
intermediate server to reach the destination server for each edge $e_{n,i,j}$.

$$NST_i(H) = \begin{cases} Par(h,i), & \text{if } h < h' & i = 1 \\ chRule, & \text{if } h > h' & i = 2 \\ & \& \; chRule \neq \varnothing & \\ & \text{if } h \oplus h' = 0 & \\ clRule, & \& \; i \oplus i' \neq 0 & i = 3 \\ & \& \; clRule \neq \varnothing & \\ & \text{if } h \oplus h' = 0 & \\ Par(h,i), & \& \; i \oplus i' \neq 0 & i = 4 \\ & \& \; clRule = \varnothing & \\ clRule, & \text{if } h > h', & i = 5 \\ & \& \; clRule \neq \varnothing & \\ & \text{if } h > h' & \\ Par(h,i), & \& \; chRule = \varnothing & i = 6 \\ & \& \; clRule = \varnothing & \\ (0,0), & \text{if } h \oplus h' = 0 & i = 7 \\ & \& \; i \oplus i' = 0 & \end{cases} \tag{5.10}$$

$$\begin{aligned} chRule = \text{if } \exists (h'', i'') \in List_{ch}(h,i) | \\ \Upsilon((\Omega(h'', i''), (h', i')) = 1, \text{return } (h'', i''), \\ \text{else return } \varnothing \end{aligned} \tag{5.11}$$

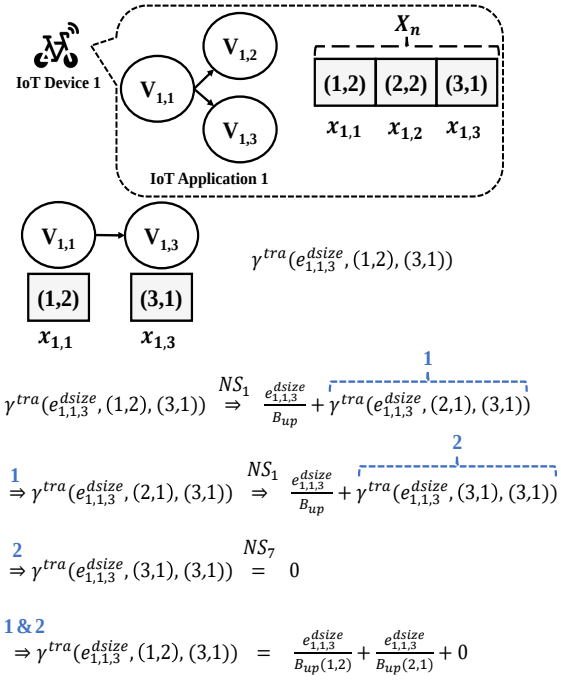
$$clRule = \text{if } \exists(h'', i'') \in List_{cl}(h, i)|$$
$$\Upsilon((\Omega(h'', i''), (h', i')) = 1, \text{return } (h'', i''), \tag{5.12}$$
$$\text{else return } \varnothing$$

The $\Upsilon((\Omega(h'', i'')), (h', i'))$ is equal to 1 if $\Omega(h'', i'')$ contains (h', i') (i.e., meaning that there is one hierarchical path from (h'', i'') to the (h', i')) and is equal to 0 if (h', i') does not exist. Moreover, the $\oplus$ is XOR binary operation. The chRule (Eq. 5.11) says that if the server (h, i) has a children (h'', i'') in its $List_{ch}$ which has a hierarchical path to the destination server (h', i'), the specification of this server (h'', i'') should be returned. The clRule (Eq. 5.12) presents that if the server (h, i) has a CM (h'', i'') in its $List_{cl}(h, i)$ which has a hierarchical path to the destination server (h', i'), the specification of this server (h'', i'') should be returned. Based on the aforementioned rules, $NST(H)$ finds the next server to which the data should be sent and calculates the transmission cost. The NST_1 of Eq. 5.10 states that if the hierarchical level of the current server is less than destination server, the $Par(h, i)$ should be checked in the next step. The NST_2 represents the case that the hierarchical level of the current server is higher than the destination server and the current server has a child through which the destination server can be reached. The NST_3 states the condition that the current and destination servers are in the same hierarchical level, and one of the CMs has a route to the destination server. The NST_4 indicates that if the current and the destination servers are in the same level, and there is no route to destination using CMs, the parent should be checked in the next step. The NST_5 states that if the level of the current server is higher than the destination server, and a CM has a path to the destination server, the cluster server should be selected in the next step. The NST_6 states that if the level of current server is higher than the destination server, and there exists no route from children nor from CMs, the parent server should be traversed. Finally, the NST_7 is the ending condition for this recursive process and states that if the current and destination server is same, the cost is zero. Fig 5.3 represents an example of obtaining transmission time between source and destination servers. Inter-nodal latency $T^{lat}_{x_{n,j}}$ between servers on which module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$



(a) An IoT application and its candidate configuration

(b) Transmission time for the data flow $e_{1,1,2}$



(c) Transmission time for the data flow $e_{1,1,3}$

Figure 5.3: An example of calculating transmission time

are placed is calculated as:

$$\begin{aligned} \Gamma^{lat}_{X_{n,j}} = \max(\gamma^{lat}((h,i),(h',i'))), \\ \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \\ x_{n,i} = (h,i), x_{n,j} = (h',i') \end{aligned} \tag{5.13}$$

where γ^{lat} shows the inter-nodal latency between source and destination servers (i.e., (h,i) and (h',i') respectively) on which $v_{n,i}$ and $v_{n,j}$ are placed. It is calculated similar to the transmission time. To reduce the equation size, we consider $((h,i),(h',i')) = A$.

$$\gamma^{lat}(A) = \begin{cases} lat_{up} + \gamma^{lat}(A'), & NST_i(A) = NST_1 | NST_4 | NST_6 \\ lat_{down} + \gamma^{lat}(A'), & NST_i(A) = NST_2 \\ lat_{cluster} + \gamma^{lat}(A'), & NST_i(A) = NST_3 | NST_5 \\ 0, & NST_i(A) = NST_7 \end{cases} \tag{5.14}$$

where lat_{up}, lat_{down}, and $lat_{cluster}$ correspond to up-link, down-link, and cluster-link inter-nodal latency respectively, and depends on the hierarchical level of servers. Besides, A' is defined as what follows:

$$A' = ((h'',i''),(h',i')) \tag{5.15}$$

The Eq. 5.15 shows the destination server (i.e., (h',i')) of A' is exactly the same as A, and the only difference is the specification of the source server (h'',i'') which is obtained from the output of $NST(A)$. The $NST(A)$ performs exactly the same as $NST(H)$ (i.e., Eq. 5.10) to find the next intermediate server, and all equation from Eq. 5.10 to Eq. 5.12 are valid here.

Energy consumption model. The goal of this model is to find a suitable placement configuration of application modules to minimize the energy consumption of the nth IoT device. To only consider energy consumption model as the main objective, the control

parameters of weighted cost model (Eq. 5.2) can be set to $w_1 = 0$ and $w_2 = 1$.

$$\Theta(X_n, t) = \begin{cases} E(x_{n,j}), & \text{if } |SchS_{n,t}| = 1 \quad \text{(a)} \\ max(E(x_{n,j})), & \text{otherwise} \\ & \text{(b)} \\ \forall x_{n,j} \in X_n | v_{n,j} \in SchS_{n,t} \end{cases} \tag{5.16}$$

where $|SchS_n|$ shows the number of schedules, and $\Theta(X_n, t)$ represents the energy consumption of modules in the tth schedule while considering the placement configuration X_n.

The overall energy consumption of each module (i.e., $v_{n,j}$) based on its candidate configuration (i.e., $x_{n,j}$) is defined as the sum of energy consumed for inter-nodal latency between servers ($E^{lat}_{x_{n,j}}$), the computing of each module ($E^{exe}_{x_{n,j}}$), and the data transmission between $v_{n,j}$ and all of its predecessor modules ($E^{tra}_{x_{n,j}}$). It is formulated as:

$$E(x_{n,j}) = E^{exe}_{x_{n,j}} + E^{lat}_{x_{n,j}} + E^{tra}_{x_{n,j}} \tag{5.17}$$

The computing energy consumption for module $v_{n,j}$ depends on its assigned server and can be derived from:

$$E^{exe}_{x_{n,j}} = \begin{cases} T^{exe}_{x_{n,j}} \times P_{cpu}, & \text{if } x_{n,j} = (h, i) \ \& \ h = 0 \\ T^{idle}_{x_{n,j}} \times P_i, & \text{if } x_{n,j} = (h, i) \ \& \ h \neq 0 \end{cases} \tag{5.18}$$

Because only the energy consumption of IoT devices is considered in this work, whenever application modules run on remote servers, the energy consumption of each IoT device is equal to the idle time $T^{idle}_{x_{n,j}}$ multiplied to the power consumption of IoT device in its idle mode P_i. Besides, P_{cpu} is the CPU power of the IoT device on which module $v_{n,j}$ runs.

The energy consumption for data transmission between module $v_{n,j}$ and its prede-

cessors $\mathcal{P}(v_{n,j})$ of the application belonging to the nth IoT device is calculated as follows:

$$E^{tra}_{x_{n,j}} = \max(\vartheta^{tra}(e^{dsize}_{n,i,j}, (h,i), (h',i'))), \quad \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \quad x_{n,i} = (h,i), x_{n,j} = (h',i') \tag{5.19}$$

where, to reduce the length of equations, we consider $H = (e^{dsize}_{n,i,j}, (h,i), (h',i'))$. Similar to response time model, (h,i) and (h',i') show the specifications of source and destination servers on which modules $v_{n,i}$ and $v_{n,j}$ run. The transmission energy consumption between each pair of dependent modules $(\vartheta^{tra}(H))$ is calculated as follows:

$$\vartheta^{tra}(H) = \begin{cases} (\frac{e^{dsize}_{n,i,j}}{B_{up}} \times P_t) + (\gamma^{tra}(H') \times P_i), & NSE_i(H) = NSE_1 \\ (\frac{e^{dsize}_{n,i,j}}{B_{down}} \times P_t) + (\gamma^{tra}(H') \times P_i), & NSE_i(H) = NSE_2 \\ \gamma^{tra}(H') \times P_i, & NSE_i(H) = NSE_3 \end{cases} \tag{5.20}$$

where P_t presents the transmission power of the IoT device, and the NSE_i shows transmission configuration based on H.

$$NSE_i(H) = \begin{cases} H' = (e^{dsize}_{n,i,j}, Par(h,i), (h',i')), & \text{if } h < h' \ \& \ h = 0, \ i = 1 \\ H' = (e^{dsize}_{n,i,j}, (h,i), Par(h',i')), & \text{if } h > h' \ \& \ h' = 0, \ i = 2 \\ H' = H, & otherwise, \quad i = 3 \end{cases} \tag{5.21}$$

NSE_1 states the data flow is starting from an IoT device as the source server to remote servers as destination. Hence, the respective transmission energy consumption is equal to the required time to send the data to the parent server of IoT device multiplied by P_t, plus the IoT device's idle time (in which the data is transmitted from parent server to the destination) multiplied by P_i. Moreover, NSE_2 represents the invocation starting

from remote servers as the source to the IoT device as the destination. It is important to note that the transmission power of IoT device P_t is active only if one of the modules is assigned to the IoT device and another module run on the remote servers, because we only consider the energy consumption from the IoT device's perspective. In other conditions, the transmission energy consumption is equal to the transmission time γ^{tra} (obtained from Eq. 5.7), in which the IoT device is in idle mode, multiplied by P_i (NSE_3).

The inter-nodal energy consumption $E^{lat}_{x_{n,j}}$ between servers on which module $v_{n,j}$ and its predecessors $\mathcal{P}(v_{n,j})$ are placed is calculated as:

$$\begin{aligned} E^{lat}_{X_{n,j}} = \max(\vartheta^{lat}((h,i),(h',i'))), \\ \forall e_{n,i,j} \in \mathcal{E}_n | v_{n,i} \in \mathcal{P}(v_{n,j}), \\ x_{n,i} = (h,i), x_{n,j} = (h',i') \end{aligned} \tag{5.22}$$

where ϑ^{lat} shows the energy consumption incurred due to inter-nodal delay between source and destination servers on which $v_{n,i}$ and $v_{n,j}$ are placed. This latter is calculated similar to transmission energy consumption based on the $NSE_i(A)$ [3, 123]. To reduce the equation size, $((h,i),(h',i')) = A$.

$$\vartheta^{lat}(A) = \gamma^{lat}(A) \times P_i \tag{5.23}$$

where the $\gamma^{lat}(A)$ is obtained from Eq. 5.14.

Migration weighted cost model

We assume that the migration of modules belonging to the nth IoT device from current servers to new servers only happens due to the mobility of IoT devices. We consider pre-copy memory migration in which the current servers still running while transferring pre-dump to the new servers [81, 206]. The goal of migration cost model is to minimize the downtime plus the required cost of executing remaining instructions on the new servers. The migration weighted cost model is defined as:

$$\min_{w_1,w_2 \in [0,1]} \Psi^{mig}((X_n, X'_n), t), \quad \forall t \in |SchS_n|, \quad \forall n \in \{1,2,\cdots,N\} \tag{5.24}$$

where

$$\Psi^{mig}((X_n, X'_n), t) = w_1 \times \Gamma^{mig}((X_n, X'_n), t) + w_2 \times \Theta^{mig}((X_n, X'_n), t) \tag{5.25}$$

$$s.t. \qquad C1 : \sum_{t=1}^{|SchS_n|} \Psi(X'_n, t) \leq \sum_{t=1}^{|SchS_n|} \Psi(X_n, t) + \epsilon$$

where $\Gamma^{mig}((X_n, X'_n), t)$ and $\Theta^{mig}((X_n, X'_n), t)$ represent the additional time and energy consumption incurred by the migration of modules of tth schedule in the downtime (when the service is interrupted). The C1 states the service cost for tasks emitted from modules of nth IoT device in the new configuration X'_n should be less or roughly the same while considering the previous configuration X_n. The ϵ shows an acceptable additional service cost in the migration. Moreover, constraints C1, C2, and C3 from Eq. 5.1 are valid here as well.

Migration time model. The migration time is considered as the execution time required to finish remaining instructions on the new servers plus the downtime. This latter includes the time for suspending the *Cnts* in current servers, transmission of the dump and states, and *Cnts*' resuming time on the new servers. Since, in the downtime, a specific amount of dump data and states should also be transferred between servers ($dsize^{mig}$), the migration latency $\gamma^{lat}_{mig}((h,i),(h',i'))$ and migration transmission time between current and new servers $\gamma^{tra}_{mig}(dsize^{mig},(h,i),(h',i'))$ to transfer this data are also important [206]. Besides, the *Cnt*s' stopping time plus its resuming time are considered as a constant I^{mig}. The migration time is defined as:

$$\Gamma^{mig}((X_n, X'_n), t) = Max(\gamma^{mig}(x_{n,i}, x'_{n,i})), \tag{5.26}$$
$$\forall x_{n,i} \in X_n, \forall x'_{n,i} \in X'_n | v_{n,i} \in SchS_{n,t},, \quad x_{n,i} = (h,i), x^{'}_{n,i} = (h',i')$$

where

$$\gamma^{mig}(x_{n,i}, x'_{n,i}) = \gamma^{lat}_{mig}((h,i),(h',i')) + I^{mig} \tag{5.27}$$
$$+\gamma^{tra}_{mig}(dsize^{mig},(h,i),(h',i')) + \frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})}$$

where $\gamma^{mig}(x_{n,i}, x'_{n,i})$ represents the migration cost of module $v_{n,i}$ from its current server $x_{n,i}$ to its new server $x'_{n,i}$. The γ^{tra}_{mig} and γ^{lat}_{mig} are calculated based on Eq. 5.7 and Eq. 5.14, respectively. Also, $\frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})}$ shows the execution time of remaining instructions of task $e^{ins,r}_{n,i,j}$ on the new server (h', i').

Migration energy consumption model. The additional energy consumption of IoT device, incurred by the migration, depends on the execution of remaining instructions and the downtime.

$$\Theta^{mig}((X_n, X'_n), t) = Max(\vartheta^{mig}(x_{n,i}, x'_{n,i})), \quad \forall x_{n,i} \in X_n, \forall x'_{n,i} \in X'_n | v_{n,i} \in SchS_{n,t}, \quad x_{n,i} = (h, i), x'_{n,i} = (h', i') \tag{5.28}$$

where

$$\vartheta^{mig}(x_{n,i}, x'_{n,i}) = \vartheta^{lat}_{mig}((h, i), (h', i')) + I^{mig} + \vartheta^{tra}_{mig}(dsize^{mig}, (h, i), (h', i')) + \vartheta^{exe}_{mig}(x'_{n,i}) \tag{5.29}$$

where $\vartheta^{mig}(x_{n,i}, x'_{n,i})$ represents the amount of energy consumed by the IoT device in the migration of each module of application from its current server $x_{n,i}$ to its new server $x'_{n,i}$. The ϑ^{tra}_{mig} and ϑ^{lat}_{mig} represent the energy consumption incurred due to the transmission and migration latency between current and new servers. They are calculated based on Eq. 5.20 and Eq. 5.23, respectively. Also, the $\vartheta^{exe}_{mig}(x'_{n,j})$ shows the energy consumption required for the execution of remaining instructions of task $e^{ins,r}_{n,i,j}$ on the new server (h', i').

$$\vartheta^{exe}_{mig}(x'_{n,i}) = \begin{cases} \frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})} \times P_{cpu}, & \text{if } x'_{n,i} = (h', i') \ \& \ h' = 0 \\ \frac{e^{ins,r}_{n,i,j}}{cpu(x'_{n,i})} \times P_i, & \text{if } x'_{n,i} = (h', i') \ \& \ h' \neq 0 \end{cases} \tag{5.30}$$

5.3.3 Optimal Decision Time Complexity

We assume M servers exist in the hierarchical Fog/Edge computing environment, and the maximum number of modules in each IoT application is K. Each module of an IoT

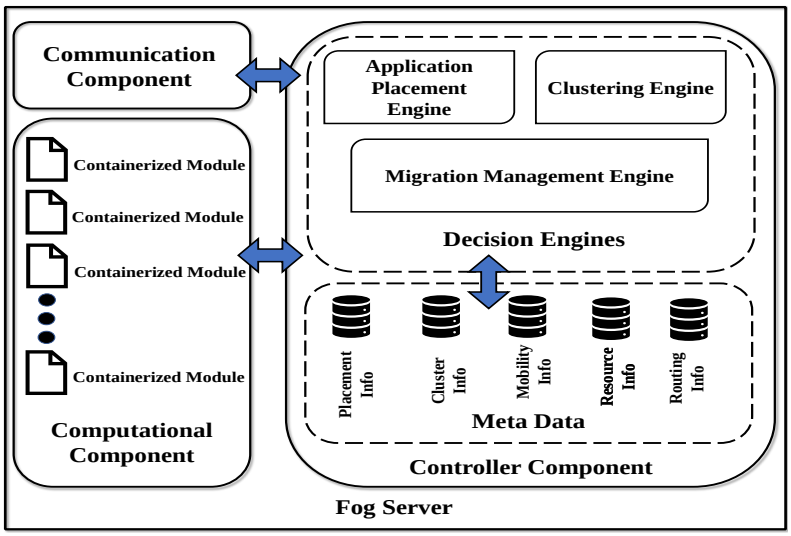


Figure 5.4: A view of Fog server architecture

application can be placed on one of the M servers at a time. Hence, for an IoT application with K modules, the Time Complexity (TC) of finding the global optimal solution for the application placement and migration is $O(M^K)$. This cost is prohibitively high and prevents us from obtaining the global optimal solution in real-time [207]. Thus, we propose distributed algorithms to find an acceptable solution in a polynomial time for application placement and migration techniques.

5.4 Proposed Technique

In this section, we present a Fog server architecture to support distributed application placement, migration management, and clustering (as depicted in Fig. 5.4) by extending the Fog server architecture proposed in [36]. Each FS in [36] is composed of three main components: controller, computational, and communication. We extend this architecture to support clustering and mobility management of IoT users in a distributed manner.

In our FS architecture, the Controller Component monitors and manages the Communication and Computational Components. It consists of three decision engine blocks and several meta-data blocks to store important information. The *Clustering Engine* is

responsible for forming a distributed cluster with its in-range FSs and updating CMs' information in the *Cluster Info* and *Routing Info* meta-data. The *Application Placement Engine* is responsible for placement of IoT applications' modules to minimize the overall cost of running real-time IoT applications. It checks *Cluster Info*, *Resource Info*, and *Routing Info* meta-data for making placement decision, and updates the *Placement Info* and *Resource Info* meta-data blocks to store the configuration of application modules and available resources in this FS, respectively. The *Migration Management Engine* of each FS controls migration process of applications' modules when IoT users move. This module considers all meta-data blocks including the current mobility information of the users (i.,e *Mobility Info*), and decides the migration destination of application modules. Based on its decision, *Placement Info* and *Resource Info* will be updated to store last changes in the configuration of application modules.

The Computational Component provides resources for the execution of application modules that are assigned to this FS based on the container technology. Besides, the Communication Component is responsible for network functionalities such as routing and packet forwarding, just to mention a few [36].

5.4.1 Dynamic Distributed Clustering

Since FSs usually have fewer resources in comparison to CSs, one FS may not be able to provide service for all modules of one application. Moreover, in some scenarios, several IoT devices are connected to the same FS, and hence, the FS may not be able to serve all application modules of different IoT devices due to its limited resources. Thus, other modules of one application should be placed on either CSs or higher-level FSs for the execution. However, in a hierarchical Fog computing environment, in which the potential clustering of FSs is considered, application modules can be placed or migrated to other FSs in the same cluster. It can reduce the placement and migration cost of application modules.

We consider that FSs belonging to the same hierarchical layer can form a cluster by any in-range FSs at the same hierarchical level and swiftly communicate together using the Constrained Application Protocol (CoAP), Simple Network Management Protocol

(SNMP), and so forth. Therefore, the communication delay within a cluster is lower than communication using up-link and down-link [36]. Besides, in a reliable IoT-enabled system, it is expected that the Fog infrastructure providers have applied efficient networking techniques to ensure steady communication among the FSs through less variable inter-nodal latency [36]. Algorithm 9 provides an overview of the dynamic distributed clustering technique.

When an FS joins the network, it receives and stores *CandidParent* control messages from FSs residing in the immediate upper layer. The new FS finds coordinates of its position and estimates the average latency to all candidate parents. It selects the FS with the minimum distance as its parent and sends an acknowledgment to it using *ParentSelection* method. Moreover, the new FS broadcasts a *FogJoining* control message, containing its position and coverage range, to its one-hop neighbors (lines 2-7). FSs receiving this message send back a *replyNewFog* control message with their list of active and inactive $Cnts$, positions' specifications, and their coverage range to the new FS. Besides, they update their CM list $List_{cl}$ with specifications of this new FS (lines 8-14). As the new FS receives *replyNewFog* message, it builds its CM list $List_{cl}$ with specifications of FSs residing in the same hierarchical layer. Alongside storing lists of active and inactive $Cnts$ of its CMs, positions, and their coverage range (lines 15-21). This distributed mechanism helps FSs to dynamically update their CM lists when a new FS joins the network.

We consider that each FS can leave the network in normal conditions (e.g., when the low-level FS is switched off by its user) or due to a failure (such as hardware or software failures). Before an FS leaves the network in normal conditions either permanently or temporarily, we assume that all of its assigned tasks should be finished. Hence, it only needs to send *StartFogLeaving* control message to its CMs to update the $List_{cl}$ of themselves, to its parent server, and to its children to find a new parent (lines 22-25). All FSs that receive *FogLeaving* control message remove all information related to this FS from their entries. Also, the children of the leaving FS that receive this control message call the *ParentSelection* method to update their parent (lines 26-32). In case of a fatal error, in which the leaving FS cannot send a control message to the parent, CMs, and children, the immediate parent runs the *StartFogFailureRecovery* and sends *FogFailureRecovery* control message to its children list $List_{ch}$ so that they can remove entries related to the failed

FS (lines 33-39). It is important to note that this latter process takes more time in comparison to the *FogLeaving* process in normal conditions due to the higher latency of uplink and downlink communications. Besides, if any FS children loose their connection to their parent, they can run the *ParentSelection* method to choose a new parent.

In addition, each FS sends the latest information about its $List_{ch}$ to its parent FS if any changes happen. This helps higher-level FSs to update their Ω.

5.4.2 Application Placement

Due to the time consuming nature of finding the optimal solution (Section 5.3.3) for the application placement problem, a Distributed application placement technique (DAPT) is proposed to find a well-suited solution in a distributed manner (Algorithm 10). The DAPT starts whenever an application placement request arrives, and the serving FS tries to place application modules on appropriate servers so that real-time tasks, emitted from modules, can be processed with the minimum cost. Considering the weighted cost (Eq. 5.1), DAPT attempts to place modules of IoT applications in one/several FSs on the lowest-possible layer while considering the potential of clustering. However, if available resources in that/those FSs are not sufficient, it considers upper layer FSs or/and CSs to place the rest of modules. In this way, DAPT reduces the search space of Eq. 5.1 for each FS by only considering itself, its parent FS, and its CMs, and aims at reducing the overall weighted cost. Moreover, a distributed failure recovery method is embedded in DAPT to recover from possible failures.

The immediate FS that receives the placement request from an IoT device is considered as the application placement controller (*controller*) for that IoT device. If the controller is performing the placement of a set of modules or a parent FS receives placement request from its children and the failure recovery mode is not active (lines 3-28), the *ClusterCheck* method returns the list of CMs and their available resources (line 4). Then, the list of ready servers S_R containing parent FS, current FS, and available CMs is created (line 5). This list contains all servers that current FS considers for the placement of modules in that hierarchical layer. Next, the *FindOrder* method checks either execution order of modules (TO_n) are available or not. If it is available, it loads the execution

Algorithm 9: Dynamic distributed clustering

```
   Input      : RCM: Received Control Message
 1 switch RCM do
 2     case CandidParent do
 3         ParentSelection()
 4         message.add(getPosition(),coverRange)
 5         message.type(FogJoining)
 6         Broadcast(message)
 7     end
 8     case FogJoining do
 9         message.add(getPosition(),coverRange)
10         message.add(getActiveCnts(),getInactiveCnts())
11         message.type(ReplyNewFog)
12         send(RCM.getSourceAddr(), message)
13         List_cl.update(RCM.getData())
14     end
15     case ReplyNewFog do
16         List_cl.update(RCM.getData())
17         MapActiveCnt_cl.put(RCM.getSourceAddr(),
18         message.getListActiveCnts())
19         MapInActiveCnt_cl.put(RCM.getSourceAddr(),
20         message.getListInActiveCnts())
21     end
22     case StartFogLeaving do
23         message.type(FogLeaving)
24         Broadcast(message)
25     end
26     case FogLeaving do
27         List_cl.remove(RCM.getSourceAddr())
28         List_ch.remove(RCM.getSourceAddr())
29         if RCM.getSourceAddr() == this.Parent) then
30             ParentSelection()
31         end
32     end
33     case StartFogFailureRecovery do
34         for i = 1 to List_ch.size() do
35             message.type(FogFailureRecovery)
36             message.setFailedFog(failedFog.getAddr())
37             send(List_ch.get(i).getSourceAddr(),message)
38         end
39     end
40     case FogFailureRecovery do
41         List_cl.remove(RCM.getFailedFogAddr())
42     end
43 end
```

Algorithm 10: An overview of DAPT

Input : $\mathcal{G}_n$: The DAG of nth IoT device, $U_{\mathcal{G}_n}$: A subset of unassigned modules from $\mathcal{G}_n$, X_n: The configuration of assigned modules, $controller_{ID}$: ID of the placement controller
Output : X_n

s_{ID}: this.ID
$List_{cl}$: this.getClusterMembers()
if (controller(n) || ReqFromChild) & !DAPTFailureRecovery(n) **then**
  $List^{A}_{cl}$=ClusterCheck($List_{cl}$)
  S_R=ReadyServers($List^{A}_{cl}$,this.parent,s_{ID})
  $SchS_n$=FindOrder($\mathcal{G}_n$)
  $U_{(\mathcal{G}_n)}$=Sort($U_{(\mathcal{G}_n)}$, $SchS_n$, S_R)
  if $S_R - Par(s_ID) \neq \varnothing$ **then**
    for $i = 1$ to $U_{\mathcal{G}_n}$.size() **do**
      v=$U_{(\mathcal{G}_n),i}$
      ID_{min}=FindMinCost(S_R,$\mathcal{G}_n$,X_n,v)
      if $ID_{min} == s_{ID}$ **then**
        res_v=CalService(v)
        if this.$Cnts$.contains(v) & **then**
          ScaleCnts(v,res_v)
        else
          StartCnt(v)
        end
        UpdateConfig(X_n,v,s_{ID})
      end
      else
        $ReqList$.update(v,ID_{min})
      end
    end
    PlaceReqToServers($ReqList$,$\mathcal{G}_n$,X_n,S_R,TO_n,$SchS_n$)
  else
    PlacePar($\mathcal{G}_n$,$U_{\mathcal{G}_n}$,X_n,TO_n,$SchS_n$)
  end
else if !controller(n) & !DAPTFailureRecovery(n) **then**
  for $i = 1$ to $U_{\mathcal{G}_n}$.size() **do**
    v=$U_{(\mathcal{G}_n),i}$
    res_v=CalService(v)
    if this.$Cnts$.contains(v) & $res_v \leq$ this.Resource **then**
      ScaleCnts(v,res_v)
      UpdateConfig(X_n,v,s_{ID})
      NotifyController(v, s_{ID},$controller_{ID}$)
    else
      if $res_v \leq$ this.Resource **then**
        StartCnt(v)
        UpdateConfig(X_n,v,s_{ID})
        NotifyController(v, s_{ID},$controller_{ID}$)
      else
        SendDAPTFailureRecovery(n,v,$controller_{ID}$,s_{ID})
      end
    end
  end
else
  DAPTFailureRecovery(n,v,S_R,X_n)
end

order and and records it in $SchS_n$ (line 6). Then, *Sort* method defines priority value for modules based on non-increasing order of their rank value, if it is currently not available (line 7). The rank of each module is defined as:

$$Rank(v_{n,j}) = \begin{cases} C^{exe}_{n,j} + \max(C^{tra}_{n,j,z} + Rank(v_{n,z})) & \text{if } v_{n,j} \neq exit \\ \qquad \forall v_{n,z} \in Succ(v_{n,j}), & \\ C^{exe}_{n,j}, & \text{if } v_{n,j} = exit \end{cases} \tag{5.31}$$

where $C^{exe}_{n,j}$ shows the average weighted execution cost of module $v_{n,j}$, and $C^{tra}_{n,j,z}$ depicts the transmission cost of module $v_{n,j}$ and $v_{n,z}$, which are calculated as:

$$C^{exe}_{n,j} = w_1 \times \widetilde{T^{exe}_{x_{n,j}}(S_R)} + w_2 \times \widetilde{E^{exe}_{x_{n,j}}(S_R)} \tag{5.32}$$

$$C^{tra}_{n,j,z} = w_1 \times \widetilde{\gamma^{tra}_{n,j,z}(S_R)} + w_2 \times \widetilde{\vartheta^{tra}_{n,j,z}(S_R)} \tag{5.33}$$

where $\widetilde{T^{exe}_{x_{n,j}}(S_R)}$ and $\widetilde{E^{exe}_{x_{n,j}}(S_R)}$ show the average execution time and energy consumption of each module considering available servers in the S_R. The execution time $T^{exe}_{x_{n,j}}$ and energy consumption $E^{exe}_{x_{n,j}}$ of each module per server are obtained from Eq. 5.5 and Eq. 5.18 respectively. Besides, $\widetilde{\gamma^{tra}_{n,j,z}(S_R)}$ and $\widetilde{\vartheta^{tra}_{n,j,z}(S_R)}$ shows the average transmission time and energy consumption between modules $v_{n,j}$ and $v_{n,z}$ considering available servers in the S_R. The transmission time $\gamma^{tra}_{n,j,z}$ and transmission energy consumption $\vartheta^{tra}_{n,j,z}$ between each pair of servers in the S_R can be obtained from Eq. 5.7 and Eq. 5.20, respectively. Moreover, w_1 and w_2 are control parameters to tune the weighted cost. The rank is calculated recursively by traversing the DAG of application, starting from the exit module. The *Sort* method can find the critical path of the DAG and gives higher priority to the modules that incur higher execution cost among modules that can be executed in parallel. Hence, the probability of placement of these modules on lower-level FSs increases. This latter is important since the resources of lower-level FSs are limited compared to higher-level FSs, but they can be accessed with less communication cost. Hence, if mod-

ules are more communication and latency-sensitive, they can be placed on lower-level FSs with higher priority while if they are computation-intensive modules, that cannot be efficiently executed on the lower-level FSs, they can be forwarded to higher-level FSs with higher priority. If S_R contains any candidate server except its parent, for each module v of $U_{\mathcal{G}_n}$, the *FindMinCost* receives the S_R, $\mathcal{G}_n$, and configuration X_n, as its input and finds the minimum cost for the execution of the module v based on current solution configuration X_n (i.e., based on the assigned servers' configuration to the predecessors of this module). Although in Fog computing environments, a large number of FSs are deployed as candidate servers, the DAPT only considers FSs in the S_R, to which the serving FS can communicate with the lowest possible transmission and inter-nodal cost. Moreover, we assume that FSs do not have a global view of all FSs in the environment. Therefore, the search space in each hierarchical layer is reduced while the suitable candidate servers for real-time and latency-sensitive IoT applications are kept. After prioritizing modules, the execution cost of each module based on the available servers in S_R is calculated using *FindMinCost* method. This method checks the available resources required to run or scale the $Cnts$ to run these modules on available servers. Then, among the servers that meet these requirements, it returns the ID of the selected FS, ID_{min}, that can execute module v while minimizing the overall application cost using Eq. 5.1 (line 11). If the current FS is selected, and it has active Cnt, the *ScaleCnt* method scales the resources so that it can serve this module (line 15). If there is no active Cnt in this FS, it should run a new Cnt, which incurs a Cnt startup cost (line 17). The candidate solution configuration X_n is updated accordingly so that the new configuration can be considered for the placement of the rest of the modules (line 19). If the selected FS is among the CMs or parent FS, the module v and its corresponding assigned server are stored in the request list *ReqList* (line 22) so that it can be forwarded to their destination using the *PlaceReqToServers* (line 25). This method sends modules to assigned serves along with the topological order of this IoT application TO_n, schedules $SchS_n$, and current solution configuration X_n. Finally, in a case that the S_R is empty, meaning that the current controller does not have any resources and also it does not have any candidate servers with sufficient resources, it sends all modules to the parent FS so that the placement can be started in the higher hierarchical levels by means of the *PlacePar* method (line 27). If the

parent FS receives the placement request from its children, it checks the possibility of placement of received modules on its S_R. The background reason is if one FS receives some modules for placement from its children FSs, it means that those modules are either more computation-intensive rather than latency/communication-intensive, or the children FSs did not have sufficient resources for these modules. However, if one FS receives a placement request from its CMs, it starts the deployment of modules on the condition that the available resources meet the modules' requirements.

If serving FS is not the controller FS and the failure recovery mode is not active (i.e, the placement request is forwarded to CMs), it iterates over the received modules (i.e., U_{G_n}) and calculates the required amount of resources for each module *CalService(v)*. If it has enough resources, it starts the module, and using *NotifyController* method sends an acknowledgment for the controller FS. However, if due to any problem this FS cannot place this module, it runs *SendDAPTFailureRecovery* method, which sends a failure message to the controller FS so that the controller can make a new decision (lines 29-47).

If failure recovery mode is active, it means that one or several servers cannot properly execute assigned modules. Hence, the DAPT algorithm calls *DAPTFailureRecovery* method. This method receives failed modules of nth IoT application and finds corresponding FSs from the solution configuration X_n. If it has several candidate servers in S_R, it removes specification of the failed FS from S_R. Then, it iterates over the rest of available servers to finds FSs for these modules that minimize the execution cost. However, if the current FS only has its parent sever in the S_R, *DAPTFailureRecovery* sends a control message to activate *DAPTFailureRecovery* method of the parent FS. (line 48). It helps to check the possibility of placement of these modules in higher hierarchical layers.

5.4.3 Migration Management Technique (MMT)

As the user of nth IoT device is moving away from its current low-level FS (i.e., its controller FS) to a new low-level FS, the current controller FS should initiate the migration process to find a new controller FS, and migrate the current data and states of running $Cnts$ to new FSs. We suppose IoT devices can detect distributed low-level FSs (eg., using

beacons, GPS, etc) and update their list of sensed FSs $List^n_{SFog}$ periodically. Whenever the controller FS realizes that the IoT device n is about to leave (e.g., through the received signal to noise ratio), it receives $List^n_{SFog}$ from the IoT device and initiates the migration process. The goals of the migration management technique (MMT) is to 1) find a new controller FS with the maximum sojourn time for the IoT device and 2) find a set of substitute servers for processing of IoT application's modules while minimizing the migration cost (Eq. 5.24). The Algorithm 11 shows an overview of the distributed migration process.

Whenever a controller FS realizes the nth IoT device is about to leave its coverage range, it initiates *MigrationInitiate* to find a new controller FS for the IoT device. The current controller FS receives the list of sensed low-level FSs $List^n_{SFog}$ from nth IoT device and removes its s_{ID} from this list so that it cannot be selected as a new controller FS (line 4). The mobility information of each user *mobInfo(n)* contains its average speed and its direction. Moreover, in the clustering technique, each FS learns the position and coverage ranges of its CMs. Considering the aforementioned values, the controller FS can estimate the sojourn time of this IoT device for each CM. The *MobilityAnalyzer* method (line 5) receives *mobInfo(n)* and $List^n_{SFog}$ and checks whether the $List^n_{SFog}$ contains any CMs of the current FS controller. Moreover, it finds specifications of other FSs belonging to $List^n_{SFog}$ through its CMs, if possible. The *MobilityAnalyzer* then creates two separate lists for reachable FSs ($List_{reach}$) and unreachable FSs ($List_{unreach}$) from $List^n_{SFog}$. The former one contains any FSs of $List^n_{SFog}$ which are among CMs of the current controller FS or those that can be accessed through its CMs, while the latter one refers to FSs to which the controller FS does not have access either directly or through its CMs. The *MobilityAnalyzer* method gives higher priority to FSs of $List_{reach}$ because the required information for the new controller to start its procedures can be more efficiently transferred to these FSs compared to those FSs to which it does not have direct access. The MMT considers *resources* of FSs belonging to $List_{reach}$, and if they have enough resources to serve modules that are currently assigned to the current controller FS, it estimates the sojourn time of nth IoT device for those candidate FSs. Then, it returns the ID of the FS with sufficient resources and the maximum estimated sojourn time. It is important to note that assigning the controller role to a new FS with maximum sojourn time can reduce

the number of possible future migrations, which leads to fewer service interruptions due to migration downtime. On the condition that no FSs of $List_{reach}$ contains enough resources, it returns the ID of FS with the maximum sojourn time. However, if $List_{reach}$ is empty, this method returns the ID of one of the FSs from $List_{unreach}$ randomly. Then, current controller FS sends a *NewControllerReq* message to $dest_{ID}$, containing the DAG of nth IoT device application $\mathcal{G}_n$, *mobilityInfo(n)*, and the current configuration of assigned servers X_n (lines 7-9).

When an FS receives *NewControllerReq* message, it adds the IoT device n to its *controllerList* to serve this IoT device as its new controller FS (lines 11-12). This new controller FS is responsible for the rest of migration management. It retrieves the current configuration X_n and the previous controller ID, ID_{PreCon}, from the received message RCM (lines 13-14). The *SortCntsSize* method descendingly sorts $Cnts$ based on their allocated runtime Ram $Cnts^{ram}$ (line 15). The background reason is the amount of dump and state to be transferred in the downtime is directly related to $Cnts^{ram}$ [206]. The migration of $Cnts$ with larger $Cnts^{ram}$ incurs higher cost in terms of migration time and energy (Eq. 5.24). Hence, to reduce the total migration cost, MMT gives higher priority to modules with heavier $Cnts^{ram}$ so that the migration decision can be made sooner, and they can be migrated before other modules. Next, *FindPreServersConfig* method retrieves assigned servers' specifications for all application modules and stores them in $MapServer_{pre}$ (line 16). The migration cost (Eq. 5.24) is defined as the maximum migration cost for each application module while considering X_n and its new configuration X'_n. The goal is to minimize this migration cost while it is subject to the condition that the new configuration X'_n provides better application execution cost or roughly the same with previous configuration X_n (Eq.5.26). So, the MMT retrieves modules of each schedule based on $SchS_n$ and send their corresponding information alongside $MapServer_{pre}$ and $List^{sorted}_{Cnts}$ to *sendMigReqToServers* method. It creates a list of modules based on the hierarchical layer on which modules are previously assigned. Modules of each hierarchical layer are also sorted based on allocated Ram size, obtained from $List^{sorted}_{Cnts}$. This method sends $MigrationReq$ messages alongside respective modules' information to FSs that are responsible for making the migration decision. As MMT acts in a distributed manner and FSs at each layer only has information about their parent, children, and

Algorithm 11: Migration management technique

Input : RCM: Received Control Message, $\mathcal{G}_n$: The DAG of nth IoT device, $mobInfo(n)$: The mobility data of the IoT device n, X_n: The configuration of assigned modules, $controller_{ID}$: ID of the controller, $List^n_{SFog}$: Sensed Fog devices' List of IoT device n

```
switch RCM do
    case MigrationInitiate do
        List^n_SFog=List^n_SFog.remove(s_ID)
        dest_Id=MobilityAnalyzer(n,mobInfo,List_cl,List^n_SFog)
        message.add(G_n,mobInfo(n),X_n,TO_n,SchS_n)
        message.type(NewControllerReq)
        send(dest_ID,message)
        controller_pre(n)=true
    end
    case NewControllerReq do
        n=RCM.getIoTDevice
        getcontrollerList().add(n)
        X_n=RCM.getConfig(n)
        ID_PreCon=RCM.getSourceAddr()
        List^sorted_Cnts =SortCntsSize(G_n, Cnts^ram)
        MapServer_pre=FindPreServersConfig(X_n)
        for t = 1 to |SchS_n| do
            sendMigReqToServers(MapServer_pre,List^sorted_Cnts,SchS_n.t)
            WaitForServersNotifications()
        end
    end
    case MigrationReq do
        ReqInfo=RCM.getInfo()
        Modules= ReqInfo.getModules()
        S_R=ReadyServers(this.getCMs(),this.getID(),this.getChildren())
        if !S_R.isEmpty() then
            for i = 1 to Modules.size() do
                SortedCostList=∅
                for j = 1 to S_R.size() do
                    MigCostTemp=CalMigCost(Modules_i,S_R,j)
                    CostList.update(S_R,j,MigCostTemp)
                end
                SortedCostList=Sort(CostList)
                Server_ID=FindMigrationDestination(SortedCostList)
                sendMigrationDestination(Modules_i,X_n,Server_ID)
            end
        end
        else
            SendMigReqToServers(this.Parent(),ReqInfo)
        end
    end
    case MigrationDestination do
        v=RCM.getModule()
        res_v=calService(v)
        if res_v ≤ this.resources then
            sendMigrationStart(v,FS^v_pre,FS^v_new)
            UpdateConfig(X_n,v,s_ID)
            NotifyController(v,s_ID,controller_ID)
        else
            SendMMTFailureRecovery(n,v,controller_ID,s_ID)
        end
    end
    case StartMigration do
        Migrate(v,RCM.FS^v_new)
        UpdateResoure(v)
        if controller_pre(n) & MigrationFinish(n) then
            controller_pre(n)=false
            getControllerList().remove(n)
        end
    end
    case MMTFailureRecovey do
        MMTFailureRecovery(n,v,controller_ID,s_ID)
    end
end
```

CMs, migration decisions for modules of each layer are made by the new controller, its parent, or ancestors in the hierarchy. To illustrate, considering Fig. 5.1, we assume an IoT application has three modules in one of its schedules and two of them were previously assigned on FS (1,3) (prior controller), and one on FS (2,1). If we assume that the new controller is FS (1,4), it makes migration decision for modules that previously assigned on FS (1,3) while $par(1,4)$ (i.e., FS (2,3)) makes migration decision for the module that previously assigned on FS (2,1). After sending migration requests *migrationReq*, FS (1,4) waits to receive notifications and new configuration of modules for that schedule and then iterates over next schedules (lines 17-20).

When an FS receives *MigrationReq* message, the FS retrieves the information and forwarded modules from the received message (lines 23-24). Then, the list of ready servers S_R is created based on CMs, and children. If the S_R does not contain any available servers, all the modules are forwarded to the parent FS for making migration decision (line 39), while if it contains servers, it tries to minimize the migration cost based on the specification of available servers (line 26-37). This FS considers a list of *modules*, sorted descendingly based on $Cnts^{ram}$, for making migration decision. Hence, the migration of modules that incur higher migration costs in each schedule is performed with higher priority, leading to less overall migration costs in that schedule. Then, for each selected module, the migration cost is estimated and stored in the *CostList* (line 29-32). The *Sort* method sorts the migration costs ascendingly so that servers with lower migration cost receives higher priority (line 33). Then, the *FindMigrationDestination* method selects a new server for the module, considering *SortedCostList*, which minimizes the migration cost while it does not negatively affect the application's running cost. Hence, this method iterates over *SortedCostList*, sorted ascendingly based on the migration costs, and selects the server that satisfies the constraints of Eq. 5.24 (line 34). Finally, the *sendMigrationDestination* method sends a *MigrationDestination* message to the selected FS to check its resources and start the migration of the respective module.

The FS receiving *MigrationDestination* checks whether it has enough resources to serve the module v or not (lines 42-44). If this FS can serve the module v, it sends a *StartMigration* message to the FS^v_{pre} so that it can start the migration. Then, it updates the X_n with its s_{ID} and notifies the controller (lines 45-48). If it cannot serve this module

due to any reason, it runs the *SendMMTFailureRecovery* method to send a failure message to the controller FS (lines 49-51).

The *MMTFailureRecovey* is working as the same as *DAPTFailureRecovey*. The only difference is that the migration cost in the MMT is obtained from Eq. 5.24 (lines 59-61).

Whenever an FS receives a *StartMigration* message, it starts the migration and then frees the previously assigned resources (lines 53-55). Moreover, if the FS was previously the controller for the nth IoT device, and it finishes the migration of all assigned modules belonging to that IoT device, the FS removes the nth IoT device from its *controllerList* (lines 56-59).

5.4.4 Complexity Analysis

The TC of the clustering phase (Algorithm 9) depends on the size of $List_{cl}$, $List_{ch}$, and immediate upper-level candidate parent of each FS. In the worst-case scenario, we assume that all FSs reside in one cluster and/or they have only one parent. Hence, the TC of *remove* method belonging to the *FogLeaving* and *FogFailureRecovery* is $O(F)$, and the TC of the *StartFogFailureRecovery* is $O(F)$. Moreover, the TC of *ParentSelection* method of *CandidParent* is $O(F)$ in the worst-case scenario if we assume one FS has $F-1$ candidate parent. Hence, the TC of the clustering step in the worst-case scenario is $O(F)$. Moreover, in the best-case scenario, the number of FSs in $List_{cl}$ and/or the size of the $List_{ch}$ is one, and the TC of the best-case is $O(1)$.

To find the TC of DAPT (Algorithm 10), we suppose that the size of the largest IoT application is K. So, in the worst-case scenario, the size of $U_{\mathcal{G}_n}$ is K. The *FindOrder* method finds the topological order of the DAG using BFS algorithm with the TC of $O(K+|\mathcal{E}|)$, in which $|\mathcal{E}|$ represents the number of data flows. In the dense DAG, the $|\mathcal{E}|$ is of $O(K^2)$. Moreover, the TC of *Sort* algorithm is $O(FK^2)$ in the worst-case scenario. In the worst-case scenario, all FSs reside in one cluster and have enough resources for any requests. Hence, the worst-case TCs of *ClusterCheck*, *ReadyServers*, *FindMinCost*, and *DAPTFailureRecovery* are of $O(F)$, $O(F)$, $O(FK)$, and $O(FK)$, respectively. Hence, the worst-case TC of DAPT Algorithm is $O(FK^2+FK)$. In the best-case scenario, the DAG of the application can be sparse so that the TC of *FindOrder* and *Sort* algorithms become $O(K)$ and

$O(1)$, respectively. Moreover, in the best-case scenario, the number of available servers in one cluster is one, and hence, TCs of *ClusterCheck*, *ReadyServers*, *FindMinCost*, and *DAPTFailureRecovery* are of $O(1)$, $O(1)$, $O(K)$, and $O(K)$, respectively. So, TC of DAPT in the best-case scenario is $O(K)$.

The TC of the *MigrationInitiate* from Algorithm 11 depends on the TC of *MobilityAnalyzer*. In the worst-case scenario, all the FSs reside in one cluster and the IoT device can sense all of them. So, the size of the list of sensed FSs $List^{n}_{SFog}$ is equal to F. Hence, in the worst-case, the TC of creating $List_{reach}$ and $List_{unreach}$ is of $O(F^2)$ while in the best-case scenario, it is of $O(F)$ when there is only one FS in the cluster. Moreover, the worst-case TC of finding maximum sojourn time is $O(F)$. So, the TC of *MigrationInitiate* in the worst-case is $O(F^2)$ while in the best-case, it is of $O(F)$. The TC of the *NewControllerReq* in the worst-case is $O(KLogK + FK)$ while TC of *NewControllerReq* in the best-case scenario is $O(KLogK)$ when there is only one FS in each cluster. The TC of *MigrationReq* in the worst-case scenario depends on the TCs of $CalMigCost$ and $Sort$ which are $O(FK)$ and $O(FKLogF)$ while in the best-case scenario they are O(K). The TC of *MigrationDestination* depends on the TC of $MMTFailureRecovery^{mig}$ and is of $O(F)$ at the worst-case and $O(1)$ in the best-case scenario. Therefore, the TC of the MMT in the worst-case scenario is $O(F^2 + FK + KLogK + FKLogF)$ while in the best-case scenario is $O(KLogK)$.

Considering TCs of all methods, the TC of our technique in the worst-case scenario is $O(F^2 + FK^2 + FKLogF)$ while in the best-case scenario, it is $O(F + KLogK)$.

5.5 Performance evaluation

In this section, the system setup and parameters, and detailed performance analysis of our technique, in comparison to its counterparts, are provided.

5.5.1 System Setup and Parameters

We extended the iFogSim simulator [26] for the implementation and evaluation of distributed mobility management, clustering, and failure recovery techniques. We used DAGs of two real-time applications, namely the Electroencephalography tractor beam

game (EEGTBG) [26, 199] and ECG Monitoring for Health-care applications (ECGMH) [32] to create our DAGs. Both applications consist of a sensor and display modules that are placed in the IoT device (e.g., smartphone, wearable devices, etc). Other modules can be placed either on distributed FSs or CSs based on the distributed application placement decisions and/or the migration technique. Data transmission intervals for ECG and EEG sensors are 10ms and 15ms, respectively [26, 36]. Besides, we assume the amount of RAM allocated to each container at the runtime for state size is randomly selected from 50-75 MBytes [206]. The total amount of data to be transferred in the downtime (i.e., $dsize^{mig}$) is just a few MBytes [206], which is randomly selected from 5-10% of each container's allocated RAM in the runtime.

We simulate a 2km $\times$ 1km area, in which the coverage range of FSs situated in the first and second layers is assumed to be 200m and 400m, respectively. The system consists of one layer of IoT devices, three layers of heterogeneous FSs, and a layer [32, 36, 200]. The IoT device layer consists of 80 IoT devices, while the number of FSs in level 1, level 2, and level 3 are 30, 5, and 1, respectively. The computing power (CPU) of IoT devices is considered as 500 MIPS [123], while the computing power of level 1 FSs is randomly selected from [3000-4000] MIPS [123, 199]. Besides, the total computing power of level 2 FSs, level 3 FSs, and CS are considered as 8000 MIPS, 10000 MIPS, and 80000 MIPS, respectively [32, 200]. Besides, the latency between IoT devices to level 1 FSs, level 1 FSs to level 2 FSs, level 2 FSs to level 3 FSs, and level 3 FSs to Cloud servers are 5ms, 25ms, 50ms, and 150ms, respectively [32, 36, 200]. The upstream and downstream network capacity of IoT devices are 100 Mbps and 200 Mbps, respectively. The upstream, downstream, and clusterlink network capacity for FSs and the CSs are also considered to be 10 Gbps [32, 200]. Moreover, clusters can be formed among the level 1 and level 2 FSs with their in-range FSs of the same hierarchical layer. The communication latency among the FSs residing in level 1 clusters and FSs residing in level 2 clusters are [3-5] ms and [20-25] ms, respectively [32, 36]. The processing power consumption, idle power consumption, and transmission power consumption of IoT devices are 0.9W, 0.3W, and 1.3W, respectively [3, 198]. User trajectories are generated by a variation of the random walk mobility model [87, 205], in which each user selects a direction, chooses a destination anywhere toward that direction, and moves towards it with a uniformly

random speed. The user arriving at the destination can choose a new random direction. Table 5.3 present the evaluation parameters used in experiments.

Table 5.3: Evaluation parameters

Parameter	**Value**
Simulation Time	100,200,300,400 (S)
Area	2km $\times$ 1km
Users' Speed	[0.5-4] m/s
Latency (ms)	
ECG Sensor Data Transmission Interval	10
EEG Sensor Data Transmission Interval	15
ECG and EEG Sensor $\leftrightarrow$ IoT Device	2
IoT Device $\leftrightarrow$ Level 1 FS	5
Level 1 FS $\leftrightarrow$ Level 2 FS	25
Level 2 FS $\leftrightarrow$ Level 3 FS	50
Level 3 FS $\leftrightarrow$ Cloud	150
L1 Clusters	[3-5]
L2 Clusters	[20-25]

5.5.2 Performance Study

We conducted seven experiments evaluating system size analysis, average execution cost of tasks, cumulative migration cost, the total number of migrations, Total number of Interrupted Tasks (TIT) due to the migration, Failure recovery analysis, and optimality analysis. In the experiments, to obtain the weighted cost of placement and migration, the w_1 and w_2 are set to 0.5. To analyze the efficiency of our technique, we extended two other counterparts in the dependent category of Fog computing proposals as follows:

- MAAS: This is the extended version of the technique called Mobility-Aware Application Scheduling (MAAS) [199] working based on edgeward-placement technique. The main concern of this Edge-centric technique is to place dependent modules of IoT applications on remote servers based on their pre-known mobility pattern (i.e., source, destination, and the potential paths between them are known in

advance) of users. In MAAS, if an FS cannot place modules on itself, the modules should be forwarded to the parent server for placement. We extended this technique to support the migration as the users move among remote servers in the runtime while considering the destination and potential paths are not priori-known.

- Urmila: This is the extended version of Ubiquitous Resource Management for Interference and Latency-Aware services (Urmila) [136] which proposes a mobility-aware technique for placement of dependent modules of IoT applications while mobility pattern of users are priori-known. In this technique, the central controller is placed in the highest level FS, and makes placement decisions for IoT applications consisting of dependent modules. We extended this technique so that the central controller helps remote servers to migrate dependent modules of applications as the IoT users move.

System size analysis

In this experiment, we study the effect of number of IoT devices on the Placement Deployment Time (PDT). The PDT shows the period between the start of sending placement requests from IoT devices up to the time the deployment of application modules on FSs are finished. Obviously, the PDT includes the decision time in which FSs make placement decisions and the container startup cost on the servers. Regardless of the quality of solutions that each technique provides, the PDT helps to understand how long the IoT devices should wait until the service can start. In this experiment, the number of IoT devices is increased from 10 to 160 by multiplication of two. Although the number of IoT devices increases, we fixed the number of FSs so that we can analyze how different techniques work when the number of placement requests increases significantly. Besides, it is clear that our technique, due to its distributed manner, can easily manage the increased number of placement requests when the number of FSs increases.

In Fig. 5.5, the PDTs of our proposed solution and MAAS are significantly lower than Urmila, specifically in the presence of larger number of IoT devices. This latter is mainly

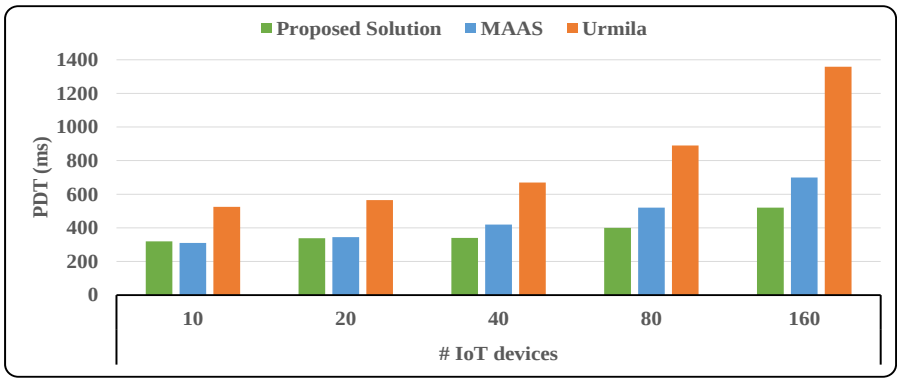


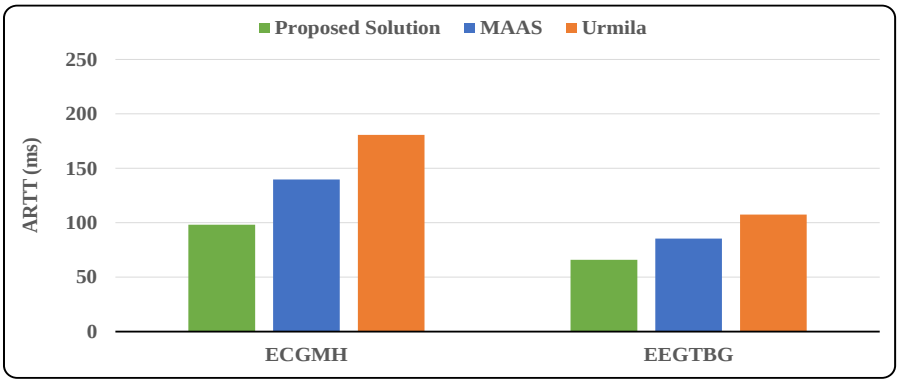
Figure 5.5: Placement Deployment Time (PDT)

because our solution and MAAS use a distributed placement engine while Urmila uses a centralized approach. When the placement decision engine receives incoming placement requests, it should make placement decisions and then manage the deployments of application modules in different servers according to solutions' configuration. In Urmila, all of the placement requests should be forwarded to the centralized entity, meaning that the number of arriving placement requests in the decision engine is larger than the distributed placement techniques. Hence, the processing of these requests on the centralized controller takes more time compared to the distributed placement engines, especially when the number of IoT devices increases. Moreover, our solution outperforms the MAAS since it tries to place more application modules in the lowest hierarchical layer, compared to MAAS, which incurs less deployment time.

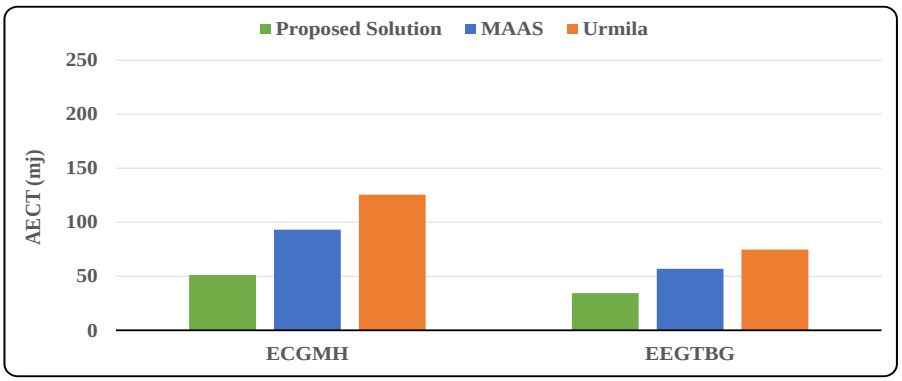
Average execution cost of tasks

This experiment shows the average execution cost of tasks emitted from a sensor module until they arrive at actuator in 400 seconds of simulation.

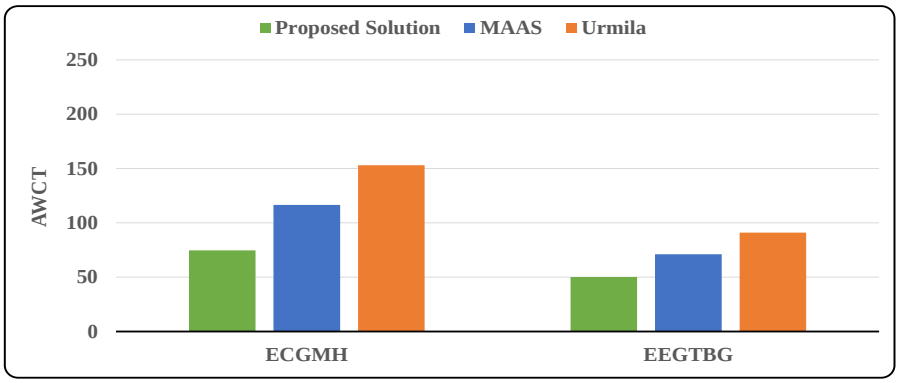
As it can be seen from Fig 5.6, our proposed solution outperforms the MAAS and Urmila in terms of Average Response Time of Tasks (ARTT), Average Energy Consumption of Tasks (AECT), and Average Weighted Cost of Tasks (AWCT). In the MAAS, each FS, from the lowest to the highest hierarchical level, attempts to place modules on itself or forwards them to its parent server for the placement or handling of the migration



(a) Average Response Time of Tasks (ARTT)



(b) Average Energy Consumption of Tasks (AECT)



(c) Average Weighted Cost of Tasks (AWCT)

Figure 5.6: Average execution cost of tasks

process. Therefore, it does not consider other potential servers at the same hierarchical level, which incurs higher transmission and inter-nodal costs. The pure Urmila, on the other hand, does not migrate the application modules to servers that are closer to

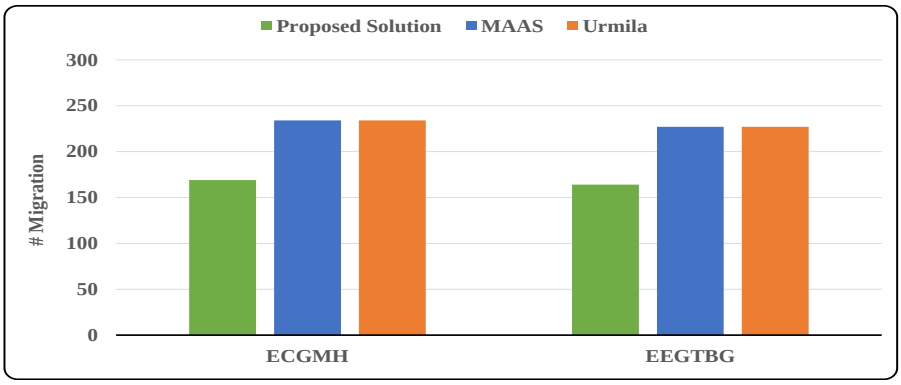


Figure 5.7: Total number of migrations

the moving IoT devices, and hence, the average execution cost of tasks, emitted from IoT devices, increases significantly. In our distributed technique, however, each FS considers potential servers at the same hierarchical level (for placement and migration) if those servers are among its CMs. In this way, we decrease the large search space of centralized techniques, while we use the benefits that servers at the same hierarchical level can provide. Also, since modules with higher costs have higher placement priority, the possibility of their placement on more suitable servers are higher compared to other modules. This latter leads to better placement decisions that minimize the cost of executing tasks. It is important to note that the average execution cost of the EEGTBG is lower than the ECGMH. It is because tasks' instruction number in the EEGTBG is lower than of ECGMH ones.

Total number of migrations

This experiment studies the total number of migrations that occurred during 400 seconds due to the IoT users' movement.

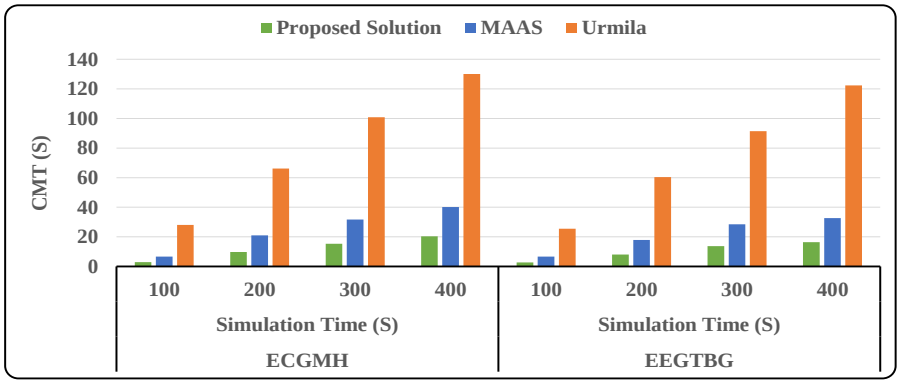
It can be seen from Fig. 5.7 that our technique leads to a smaller number of migrations in comparison to its counterparts. This is because our solution considers the current mobility information of IoT devices such as current speed and direction. Since the controller FS has coordinates of its CMs and current mobility information of leaving IoT devices (e.g., their average speed and their direction while in the range of the current controller

FS), the serving FS can estimate a sojourn time for all candidate remote servers for the migration. Hence, by the migration of modules to the remote server with the highest sojourn time (in case sufficient resources are available), the number of possible migrations decreases. The extended MAAS and Urmila only try to reduce the migration cost by migrating modules to new remote servers, while they do not consider current mobility information of IoT devices and their sojourn time in remote servers. Hence, they may select remote servers in which the IoT devices stay only for a short period.

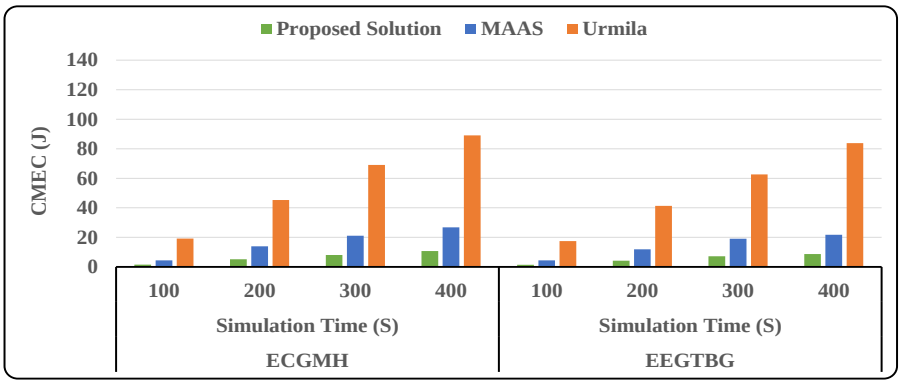
Cumulative migration cost

This experiment analyzes the Cumulative Migration Cost (CMC) of IoT devices for ECGMH and EEGTBG in different simulation times. The term cumulative refers to the aggregate migration cost of all IoT devices.

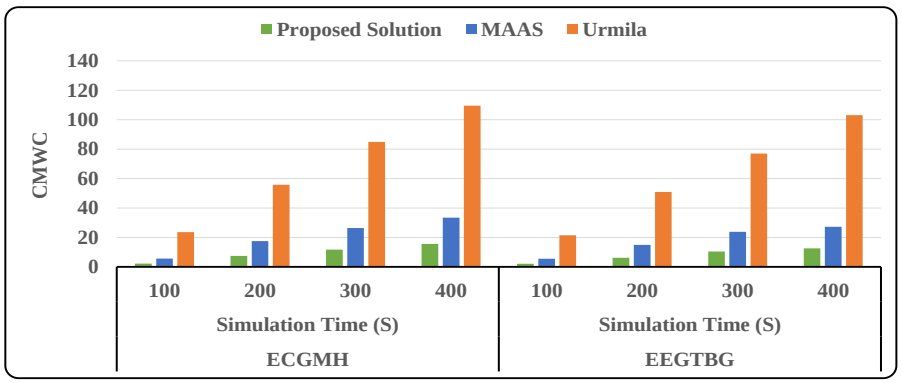
As Fig 5.8 shows, our solution outperforms its counterparts in terms of Cumulative Migration Time (CMT), Cumulative Migration Energy Consumption (CMEC), and Cumulative Migration Weighted Cost (CMWC) for both ECGMH and EEGTBG applications. As the simulation time increases, the cost of all techniques grows, however, Urmila experiences a faster increase in comparison to our solution and MAAS. This latter is because the Urmila's controller is placed at the highest hierarchical layer, which incurs significant inter-nodal and transmission cost when the controller manages migrations between the old and new remote servers in the downtime. Besides, the migration cost of MAAS is more than our solution, since whenever the resources of controller finishes, the MAAS migrates the application modules to higher layers, and hence, the emitted tasks to/from those modules experience higher cost. Also, the total number of migrations in Urmila and MAAS are higher than ours, which apparently increases their cumulative migration costs. The slight difference between cost of ECGMH and EEGTBG is because the tasks generated from the ECGMH's modules are heavier than EEGTBG's ones in terms of their MI. So, the processing time of remaining instructions of tasks (i.e., $e^{ins,r}_{n,i,j}$) that migrated from old server to new server is higher for the ECGMH compared to the EEGTBG (in case the computing powers of old and new servers are roughly the same).



(a) Cumulative Migration Time (CMT)



(b) Cumulative Migration Energy Consumption (CMEC)



(c) Cumulative Migration Weighted Cost (CMWC)

Figure 5.8: Cumulative Migration Cost

Total number of interrupted tasks (TIT)

This experiment analyzes the Total number of Interrupted Tasks (TIT) in the downtime. During migration downtime, there is no active service provider for incoming tasks from

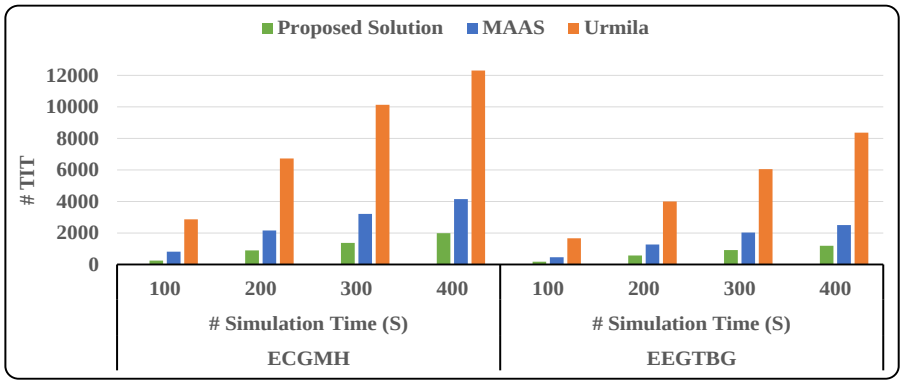


Figure 5.9: Total number of interrupted tasks

the modules deployed on the IoT device for a while. Hence, service interruptions happen in the downtime, in which the generated tasks experience higher delays or even they can be discarded, compared to the tasks that are generated when there is no migration. The IoT users receive smoother results with lower TIT.

Fig. 5.9 presents the TIT of techniques for ECGMH and EEGTBG in different simulation times. It can be seen that our solution outperforms its counterparts in different simulation times for the ECGMH and EEGTBG. The migration time has a direct impact on the TIT, and the techniques with higher migration time lead to larger TIT. This latter is because as the migration time increases, the number of delayed (or even dropped) tasks grows faster. It can be seen from Fig. 5.9 that the Urmila results in larger TIT than two other techniques because of its higher migration time. Moreover, due to our smaller migration time, the TIT of our solution is smaller than other techniques for both ECGMH and EEGTBG applications. It is worth mentioning that the TIT of techniques for EEGTBG applications is smaller than of ECGMH ones. This latter is due to a higher data transmission interval for the EEG sensor in EEGTBG compared to the ECG sensor of ECGMH, which means that the number of emitted tasks per second for the EEGTBG application is smaller than the ECGMH application. Hence, applications with shorter task emission interval (here, the ECGMH application) suffer more from higher migration time.

Table 5.4: Failure recovery analysis

Applications	Experiment	Techniques		
		Proposed Solution (FR Mode)	MAAS (No FR)	Urmila (No FR)
ECGMH	Total Number of Migrations	177	234	234
	Total Number of Interrupted Tasks	2095	4152	12302
EEGTBG	Total Number of Migrations	169	227	227
	Total Number of Interrupted Tasks	1228	2504	8361

Failure recovery analysis

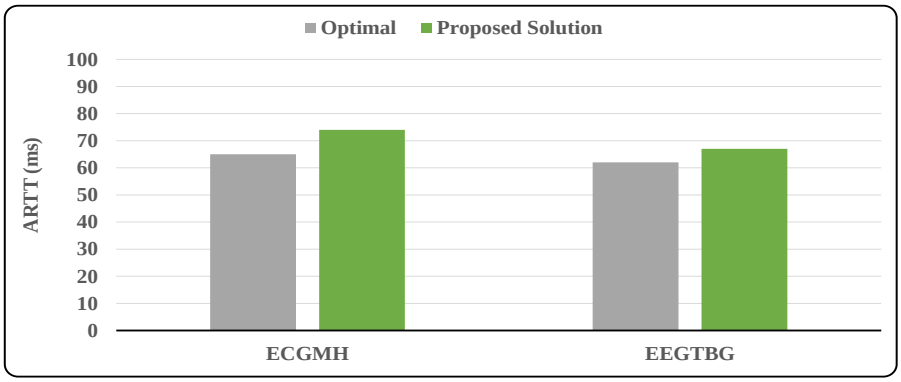
In this experiment, we study the effect of the failure recovery method in the migration process. The MAAS and Urmila do not have any failure recovery methods and their results are just presented here for comparison purposes. The results of our technique with a failure recovery method (FR Mode) are presented in Table 5.4 when there is a 5% probability of failure in the migration process.

Table 5.4 illustrates that our technique with the failure recovery method (FR Mode) can recover from failures while it still outperforms its counterparts in terms of the total number of migrations and TIT. The obtained results of the average execution cost of tasks and cumulative migration cost in the FR Mode are roughly the same with the Non-FR Mode and they are not provided here. Since the Urmila and MAAS do not have any failure recovery methods, in case of any failures, their placement and/or migration process remains incomplete. However, in our technique, we embedded the failure recovery method for which it accepts a small overhead while it does not stop working if any failures occur.

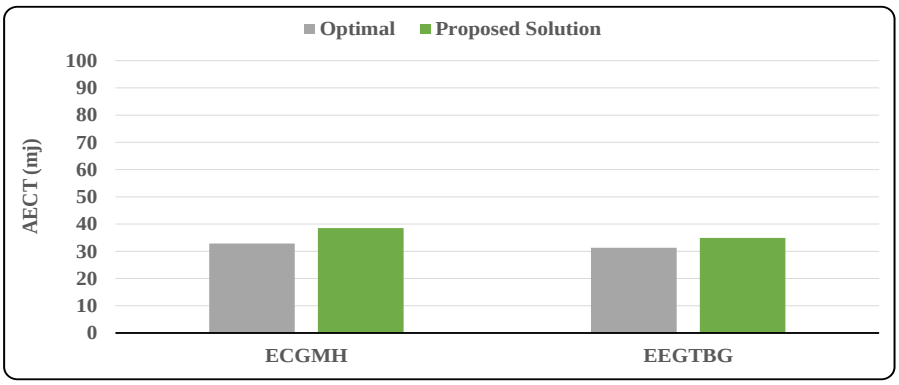
Optimality analysis

In this experiment, we compare the performance of our solution with the optimal values. To obtain the optimal results, we used an optimized version of the Branch-and-Bound algorithm to search all possible candidate configurations for application placement, in which the bounding function helps to faster prune the search space [29]. Since finding the optimal solution is very time consuming, we only consider 20 IoT devices in a

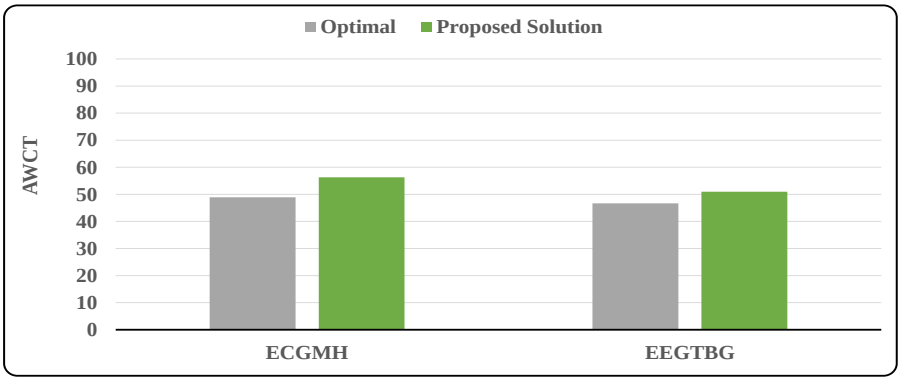
hierarchical Fog computing environment, consisting of 15 candidate servers.



(a) Average Response Time of Tasks (ARTT)



(b) Average Energy Consumption of Tasks (AECT)



(c) Average Weighted Cost of Tasks (AWCT)

Figure 5.10: Optimality analysis results

Fig. 5.10 shows the results of optimality analysis in terms of Average Response Time

of Tasks (ARTT), Average Energy Consumption of Tasks (AECT), and Average Weighted Cost of Tasks (AWCT). The results show that our solution has an average of 12% difference with the optimal results. However, considering the large number of FSs distributed in the proximity of IoT users, obtaining the optimal solutions, due to their large search spaces, is not practically possible, especially for real-time IoT applications.

5.6 Summary

In this chapter, we proposed a new weighted cost model for minimizing the overall response time and energy consumption of IoT devices in a hierarchical Fog computing environment, in which heterogeneous FSs and CSs provide services for IoT devices. In order to enable collaboration among remote servers and provide better services for IoT applications, we proposed a dynamic and distributed clustering technique among FSs of the same hierarchical level. Considering the heterogeneous resources of remote servers and the dynamic nature of such computing environments, we also proposed a distributed application placement technique to place interdependent modules of IoT applications on appropriate remote servers while satisfying their resource requirements. Also, to manage potential migrations of IoT applications' modules among remote servers, due to IoT users' mobility, a distributed migration management technique is proposed. The main goal of this latter is to reduce the migration cost of IoT applications. Finally, we embedded light-weight failure recovery methods to handle possible unpredicted failures that may happen in such dynamic computing environments. The effectiveness of our technique is analyzed through extensive experiments and comparisons by the state-of-the-art techniques in the literature. The obtained results demonstrate that our technique improves its counterparts in terms of placement deployment time, average execution cost of tasks, the total number of migrations, cumulative migration cost of all IoT devices, and the total number of interrupted tasks due to migration.

While this chapter presented heuristic-based algorithms for fast placement and migration of IoT applications, in the next chapter, we explore building a complete distributed learning-based scheduling model using the RL framework.

Chapter 6

Deep Reinforcement Learning-based Application Placement Technique

Several Deep Reinforcement Learning (DRL)-based placement techniques have been proposed in Fog/Edge computing environments, which are only suitable for centralized setups. The training of well-performed DRL agents requires manifold training data while obtaining training data is costly. Thus, these DRL-based techniques lack generalizability and quick adaptability, hence failing to efficiently tackle placement problems. Also, many IoT applications are modeled as Directed Acyclic Graphs (DAGs) with diverse topologies. Satisfying dependencies of DAG-based IoT applications incur additional constraints and increase the complexity of placement problem. To overcome these challenges, we propose an actor-critic-based distributed application placement technique, working based on the IMPortance weighted Actor-Learner Architectures (IMPALA) known for efficient distributed experience trajectory generation that significantly reduces exploration costs of agents. Besides, it uses an adaptive off-policy correction method for faster convergence to optimal solutions. The performance results show that our technique significantly improves execution cost of IoT applications up to 30% compared to its counterparts.

6.1 Introduction

In real-world scenarios, many IoT applications (e.g., face recognition [208], smart healthcare [209],and augmented reality [210]) are modeled as a Directed Acyclic Graph (DAG), in which nodes and edges represent tasks and data communication among dependent

This chapter is derived from:

- **Mohammad Goudarzi**, Marimuthu Palaniswami, and Rajkumar Buyya, "A Distributed Deep Reinforcement Learning Technique for Application Placement in Edge and Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC)*, (in press, DOI: 10.1109/TMC.2021.3123165, accepted on 23 October, 2021).

tasks, respectively. These DAG-based IoT applications incur higher complexity and constraints when making placement decisions. Hence, placement/offloading of IoT applications, comprised of dependent tasks, on/to suitable servers with the minimum execution time and energy consumption is an important and yet challenging problem in Fog computing. Many heuristics, approximation, and rule-based solutions are proposed for this NP-hard problem [11, 199, 211]. Although these techniques work well in general cases, they heavily rely on comprehensive knowledge about the IoT applications and resource providers (e.g., CSs or FSs). The Fog computing environment is stochastic in several aspects, such as arrival rate of application placement requests, dependency among tasks, number of tasks per IoT application, resource requirements of applications, and available remote resources, just to mention a few. Therefore, heuristic-based techniques cannot efficiently adapt to constant changes in the Fog computing environments [212].

Deep Reinforcement Learning (DRL) provides a promising solution by combining Reinforcement Learning (RL) with Deep Neural Network (DNN). Since DRL agents can accurately learn the optimal policy and long-term rewards without prior knowledge of the system [119, 213], they help solve complex problems in dynamic and stochastic environments such as Fog computing, especially when the state space is so large [212, 214]. Although the effectiveness of DRL techniques is shown in several works [71, 99, 130, 131, 215], there are yet several challenges for practical realizations of these techniques in Fog computing environments. In DRL, the agent interacts with the environment using trial and error (i.e., exploration) and records the trajectories of experiences (i.e., sequences of states, actions, and rewards) in large quantities with high diversity. These experience trajectories are used to learn the optimal policy in the training phase. In complex environments, such as Fog computing, DRL agents require a large number of interactions with the environment to obtain sufficient trajectories of experience to capture the properties of the environment. Therefore, the exploration cost of agents increases. Obviously, it negatively affects the user experience in the Fog computing environment, because the training of the DRL agents in such complex environments is a time-consuming process. The centralized DRL agents used in Fog computing environments are not suitable for the highly distributed and stochastic environments [97]. Hence, a key problem is how to adapt distributed DRL techniques to efficiently perform

in Fog computing environments. Considering the distributed nature of Fog computing environments, the application placement engines can be placed on different FSs, that work in parallel and efficiently produce diverse experience trajectories with less exploration costs. However, other challenges may arise such as how these trajectories can be efficiently and practically used to learn the optimal policy.

To address the aforementioned challenges, we propose an EXperience-sharing Distributed Deep Reinforcement Learning-based application placement technique, called *X*-DDRL, to efficiently capture complex dynamics of DAG-based IoT applications and FSs' resources. The *X*-DDRL uses IMPortance weighted Actor-Learner Architectures (IMPALA), proposed by Espeholt et al. [216], which is a distributed DRL agent that uses an actor-learner framework to learn the optimal policy. In IMPALA, several actors interact with the environments in parallel and produce diverse experience trajectories in a timely manner. Then, these experience trajectories are periodically forwarded to the learner for the training and learning of the optimal policy. After each policy update of the learner, actors reset their parameters with the learner's one and independently continue their explorations. As a result of this distributed and collaborative experience-sharing between actors and learners, the exploration costs reduce significantly, and the experience trajectories are efficiently reused. However, due to decoupled acting and learning, a policy gap between actors and learners arises, which can be corrected by V-trace off-policy correction method [216]. Moreover, we use Recurrent Neural Networks (RNN) to accurately identify the temporal patterns across different features of the input. Finally, the *X*-DDRL uses experience replay to break the strong correlation between generated experience trajectories and improve sample efficiency.

The main **contributions** of this chapter are:

- A weighted cost model for application placement of DAG-based IoT applications is proposed to minimize the execution time of IoT applications and energy consumption of IoT devices. Then, this weighted cost model is adapted to be used in DRL-based techniques.

- A pre-scheduling technique is put forward to define an execution order for dependent tasks within each DAG-based IoT application.

- A dynamic and distributed DRL-based application placement technique is proposed for complex and stochastic Fog computing environments, working based on the IMPALA framework. Our technique uses RNN to capture complex patterns across different features of the input. Moreover, it uses an experience replay buffer which remarkably helps sampling efficiency and breaks the strong correlation between experience trajectories.

- Simulation and testbed experiments are conducted using a wide range of synthetic DAGs, derived from the real-world IoT applications, to cover diverse application dependency models, task numbers, and execution costs.

The rest of the chapter is organized as follows. Relevant DRL-based application placement techniques in Edge and Fog computing environments are discussed in Section 6.2. The system model and problem formulations are presented in Section 6.3. Section 6.4 describes the DRL-based model and its main concepts. Section 6.5 presents our proposed distributed DRL-based application placement framework. We evaluate the performance of our technique and compare it with state-of-the-art techniques in Section 6.6. Finally, Section 6.7 concludes the chapter.

6.2 Related Work

Considering the large number of works in application placement techniques, in this section, related works for DRL-based application placement techniques in Fog/Edge computing environments are studied. However, detailed related works for the non-learning-based application placement techniques and frameworks are available in [1, 3, 76].

DRL-based works are first divided into Edge computing and Fog computing. Edge computing works only consider the resources in the proximity of IoT users while Fog computing ones take advantage of both Edge resources and remote Cloud resources. Hence, the heterogeneity of resources is higher in the Fog computing works, which leads to higher complexity for DRL-based application placement techniques to identify the features of the environments. Besides, works are further categorized into independent and dependent categories based on the dependency model of their IoT ap-

plications' granularity (e.g., tasks, modules). In IoT applications with dependent tasks (i.e., DAGs), each task can be executed only when its parent tasks finish their execution, while tasks of independent IoT applications do not have such constraints for execution. Therefore, works in the dependent category have more constraints, and hence the DRL agent requires specific considerations compared to works in the independent category to efficiently learn the optimal policy.

6.2.1 Edge Computing

In the independent category, Huang et al. [73] proposed a DRL-based offloading algorithm to minimize the system cost, in which parallel computing is used to speed up the computation of a single Edge server. Min et al. [69] proposed a fast deep Q-network (DQN) based offloading scheme, combining the deep learning and hotbooting techniques to improve the learning speed of Q-learning. Huang et al. [99] proposed a quantized-based DRL method to optimize the system energy consumption for faster processing of IoT devices' requests. Chen et al. [71] proposed a double DQN-based algorithm to minimize the energy consumption and execution time of independent tasks of IoT applications. Huang et al. [130] also proposed a DRL-based offloading framework based on DQN that jointly considers offloading decisions and resource allocations. Chen et al. [217] proposed a joint offloading framework with DRL to make an offloading decision based on the information of applications' tasks and network conditions where the training data is generated from the searching process of the Monte Carlo tree search algorithm. Lu et al. [131] proposed a Deep Deterministic Policy Gradients (DDPG)-based algorithm for computation offloading of multiple IoT users to a single Edge server to improve the quality of experience of users. To improve the convergence of the DQN algorithm in an Edge computing environment, Xiong et al. [132] proposed a DQN-based algorithm combined with multiple replay memories to minimize the execution time of one IoT application. Qiu et al. [120] studied the distributed DRL in an Edge computing environment with a single Edge server to minimize the energy cost of running IoT applications, consisted of independent tasks. To obtain this goal, they combined deep neuro-evolution and policy gradient to improve the convergence results.

In the dependent category, Wang et al. [119] proposed a meta reinforcement learning algorithm based on the Proximal Policy Optimization (PPO). The main goal of this work is to minimize the execution time of dependent IoT applications, situated in the proximity of a single Edge server.

6.2.2 Fog Computing

In the independent category, Gazori et al. [133] targeted task scheduling of independent IoT applications to minimize long-term service delay and system cost. To obtain this, they used a double DQN-based scheduling algorithm combined with an experience replay buffer. Tuli et al. [97] proposed Asynchronous-Advantage-Actor-Critic (A3C) learning-based technique combined with Recurrent Neural Network (RNN) for the scheduling of independent IoT applications to minimize total system cost.

In the dependent category, Lu et al. [78] proposed a DQN-based algorithm to minimize the overall system cost. Although they consider dependencies among constituent parts of each IoT application, they only consider the sequential dependency model among tasks of an IoT application, where there are no tasks for parallel execution.

6.2.3 A Qualitative Comparison

Table 6.1 identifies and compares the main elements of related works with ours in terms of their IoT application, architectural, and application placement engine properties. In the IoT application section, the dependency model of each proposal is studied, which can be either independent or dependent. Moreover, we study how each proposal models IoT applications in terms of the number of tasks and heterogeneity. This demonstrates whether IoT applications consist of homogeneous or heterogeneous tasks in terms of their computation and data flow. In the architectural properties, the attributes of IoT devices, Fog/Edge servers, and Cloud servers are studied. For IoT devices, the overall number of devices and their type of requests are identified. The heterogeneous request type shows that each device has a different number of requests with various requirements compared to other IoT devices. For Edge/Fog servers, the number of deployed servers between IoT devices and Cloud servers and the heterogeneity of their resources

Table 6.1: A qualitative comparison of related works with ours

Techniques	Category	Application Properties			Architectural Properties					Application Placement Engine Properties				
					IoT Device Layer		Edge/ Layer					Decision Parameters		
		Dependency Model	Task Number	Hetero	Number	Request Type	Number	Hetero	Multi Cloud	Main Method	Task Priority	Time	Energy	Weighted
[73]	Edge Computing	Independent	Multiple	✓	Multiple	×	Single	×	×	-	×	×	×	✓
[69]			Single	×	Multiple	✓	Single	×	×	-	×	×	✓	×
[99]			Single	×	Single	×	Multiple	✓	×	DQN	×	×	✓	×
[71]			Single	✓	Multiple	✓	Single	×	×	Double DQN	×	✓	✓	×
[130]			Multiple	✓	Multiple	✓	Single	×	×	DQN	×	✓	✓	✓
[217]			Single	✓	Multiple	✓	Multiple	✓	×	MCTS	×	✓	✓	×
[131]			Multiple	✓	Multiple	✓	Single	×	×	DDPG	×	✓	✓	✓
[132]			Multiple	✓	Multiple	×	Single	×	×	DQN	×	✓	×	×
[120]			Multiple	✓	Multiple	✓	Single	×	×	Deep Neuro evolution	×	×	✓	×
[119]		Dependent	Multiple	✓	Multiple	✓	Single	×	×	PPO	✓	✓	×	×
[133]	Fog Computing	Independent	Multiple	✓	Multiple	✓	Multiple	✓	×	Double DQN	×	✓	×	×
[97]			Multiple	✓	Multiple	✓	Multiple	✓	×	A3C	×	✓	✓	✓
[78]		Dependent	Multiple	✓	Multiple	×	Multiple	✓	×	DQN	×	×	✓	×
X-DDRL			Multiple	✓	Multiple	✓	Multiple	✓	✓	IMPALA	✓	✓	✓	✓

The abbreviated terms are as follows: Hetero: Heterogeneity

are studied. Moreover, the multi-Cloud shows either these works consider different Cloud service providers with heterogeneous resources or not. In the application placement engine, the main employed DRL methods are identified. Besides, it is studied either these works consider any mechanism to provide priority for the execution of tasks or not. Finally, the decision parameters of these DRL-based techniques are identified.

Considering DRL-based application placement techniques in Edge and Fog computing and their identified properties, the environment with multiple heterogeneous IoT devices, heterogeneous FSs, and heterogeneous multi CSs has the highest number of features. Moreover, DAG-based IoT applications incur more constraints on DRL agents as they need to consider the dependency among tasks within each IoT application. The exploration cost of DRL agents increases as the number of features and complexity of the environment increases. It negatively affects the training and convergence time of DRL techniques, and accordingly users' experience. To address these issues, we pro-

pose a distributed DRL technique based on the IMPALA architecture, called *X*-DDRL, in which several actors independently interact with Fog computing environments and create experience trajectories in parallel. Then, these distributed experience trajectories are forwarded to the learner for training and policy updates. This significantly reduces the exploration and training costs of centralized DRL techniques. Furthermore, since the learner directly uses the batches of experience trajectories of distributed actors, rather than gradients with respect to the parameters of the policy (similar to how the A3C algorithm works), it can more efficiently learn and identify the features of input data [216]. Also, the transmission of gradients among actors and learners is more expensive in terms of data exchange size and time (similar to how A3C works) in comparison to sharing trajectories of experience. Hence, experience-sharing DRL techniques such as IMPALA are more practical and data-efficient in highly distributed and stochastic environments [216], such as Fog computing. Since the policy used to generate the trajectories of experiences in distributed actors can lag behind the policy of the learner in the time of gradient calculations, a V-trace off-policy actor-critic algorithm is used to correct this discrepancy. Besides, to capture the temporal behavior of input data, we embed RNN layers in the network of actors and learners. Moreover, *X*-DDRL uses a replay buffer to improve the sample efficiency for training.

6.3 System Model and Problem Formulation

Fig. 6.1 represents an overview of our system model in Fog computing. IoT devices send their application placement requests to brokers, situated at the edge of the network to be accessed with less latency and higher bandwidth [3, 218]. For each arriving application request, the broker makes a placement decision based on the corresponding DAG of the IoT application, its constraints, and the system status. Accordingly, each task of an IoT application may be assigned to the IoT device for the local execution or one of heterogeneous FSs or CSs for the execution.

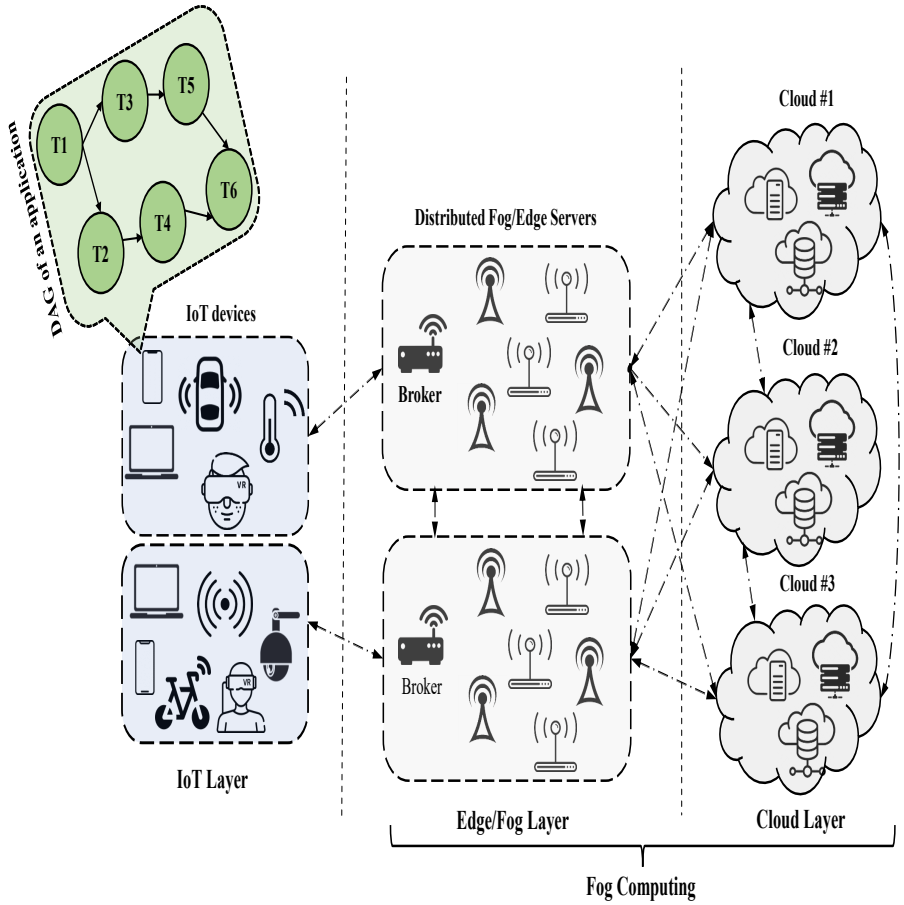


Figure 6.1: An overview of our system model

6.3.1 IoT Application

Each IoT applications is modeled as a DAG $G = (\mathcal{V}, \mathcal{E})$ of its tasks, where $\mathcal{V} = \{v_i | 1 \leq i \leq |\mathcal{V}|\}, |\mathcal{V}| = L$ depicts vertex set of one application, in which v_i denotes the ith task. Moreover, $\mathcal{E} = \{e_{i,j} | v_i, v_j \in \mathcal{V},\ i \neq j\}$ represents edge set, in which $e_{i,j}$ denotes there is a data flow between v_i (i.e., parent), v_j (i.e., child) and hence, v_j cannot be executed before v_i. Accordingly, for each task v_j, a predecessor task set $\mathcal{P}(v_j)$ is defined, containing all tasks that should be executed before v_j. Moreover, for each DAG G, exit tasks are referred to tasks without any children.

The amount of CPU cycles, required for the processing of each task, is represented as v_j^w, while the required amount of RAM for processing of each task is v_j^{ram}. Moreover,

the weight on each edge $e^{w}_{i,j}$ illustrates the amount of data that task v_i sends as its output to task v_j as its input.

6.3.2 Problem Formulation

Each task of an IoT application can either be executed locally on the IoT device or on one of the FSs or CSs. We define the set of all available servers as $\mathcal{M}$ where $|\mathcal{M}| = M$. Each server is represented as $m^{y,z} \in \mathcal{M}$ where y shows the type of server (IoT device ($y = 0$), FSs ($y = 1$), CSs ($y = 2$)) and z denotes the server index. Therefore, the placement configuration of task v_j, belonging to an IoT application, can be defined as:

$$x_{v_j} = m^{y,z} \tag{6.1}$$

and accordingly, the placement configuration of an IoT application $\mathcal{X}$ is defined as the set of placement configurations for all of its tasks:

$$\mathcal{X} = \{x_{v_j} | v_j \in \mathcal{V}, 1 \leq j \leq |\mathcal{V}|\} \tag{6.2}$$

We consider that tasks of an IoT application are sorted in a sequence so that all parent tasks are scheduled for the execution before their children. Hence, the dependencies among tasks are satisfied. Besides, among tasks that can be executed in parallel (i.e., tasks that all of their dependencies are satisfied), the $CP(v_i)$ is an indicator function to demonstrate whether the task is on the critical path of the IoT application or not [140] (i.e., a path containing vertices and edges that incurs the highest execution cost).

Execution time model

The execution time of each task v_j depends on the availability time of required input data for that task $\psi^{input}_{x_{v_j}}$ and its processing time on the assigned server $\psi^{proc}_{x_{v_j}}$:

$$\psi_{x_{v_j}} = \psi^{proc}_{x_{v_j}} + \psi^{input}_{x_{v_j}} \tag{6.3}$$

where $\psi^{proc}_{x_{v_j}}$ depends on the required CPU cycles for that task v^w_j and the processing speed of the corresponding assigned server $f^s_{x_{v_j}}$, calculated as follows:

$$\psi^{proc}_{x_{v_j}} = \frac{v^w_j}{f^s_{x_{v_j}}} \tag{6.4}$$

The $\psi^{input}_{x_{v_j}}$ is calculated as the maximum time that the required input data for the execution of task v_j becomes available on the corresponding assigned server (i.e., x_{v_j}) from its parent tasks:

$$\psi^{input}_{x_{v_j}} = \max((\frac{e^w_{i,j}}{b_{x_{v_i},x_{v_j}}} + l_{x_{v_i},x_{v_j}}) \times SS(x_{v_i}, x_{v_j})), \quad \forall v_i \in \mathcal{P}(v_j) \tag{6.5}$$

where $b_{x_{v_i},x_{v_j}}$ shows the current bandwidth (i.e., data rate) between the servers to which v_i and v_j are assigned, respectively. Moreover, $l_{x_{v_i},x_{v_j}}$ demonstrates the communication latency between two servers. The $SS(x_{v_i}, x_{v_j})$ is equal to 0 if $x_{v_i} = x_{v_j}$ (i.e., same assigned servers) or 1, otherwise. Since Fog computing environments are heterogeneous and stochastic, the $f^s_{x_{v_j}}$, $b_{x_{v_i},x_{v_j}}$, and $l_{x_{v_i},x_{v_j}}$ may be different among IoT devices, FSs, and CSs.

The main goal of the execution time model is to find the best-possible placement configuration for the IoT application so that its execution time becomes minimized. Assuming an IoT application consists of L tasks, the execution time model is defined as:

$$\Psi(\mathcal{X}) = \min(\sum_{j=1}^{L} CP(v_j) \times \psi_{x_{v_j}}) \tag{6.6}$$

where $CP(v_j)$ is 1 if task v_j is on the critical path and 0 otherwise. Due to the parallel execution of some tasks, only the execution time of tasks on the critical path is considered, which incurs the highest execution time and involves the execution time of other parallel tasks as well.

Energy consumption model

We only consider the energy consumption of IoT devices in this work since FSs and CSs are usually connected to constant power supplies [123]. From the IoT devices' per-

spective, the energy consumption that execution of each task v_j incurs depends on the amount of energy the IoT device consumes until the required input data for that task $\omega^{input}_{x_{v_j}}$ becomes ready plus the required energy for the processing of that task $\omega^{proc}_{x_{v_j}}$:

$$\omega_{x_{v_j}} = \omega^{proc}_{x_{v_j}} + \omega^{input}_{x_{v_j}} \tag{6.7}$$

where $\omega^{proc}_{x_{v_j}}$ depends on whether the task is assigned to the IoT device for local execution or not. Hence, we define an IoT Server identifier $IS(x_{v_j})$ to show whether the x_{v_j} refers to an IoT device ($IS(x_{v_j}) = 1$) or other servers ($IS(x_{v_j}) = 0$). Accordingly, the $\omega^{proc}_{x_{v_j}}$ is calculated as follows:

$$\omega^{proc}_{x_{v_j}} = \begin{cases} \psi^{proc}_{x_{v_j}} \times P^{cpu}, & IS(x_{v_j}) = 1 \\ \psi^{proc}_{x_{v_j}} \times P^{idle}, & IS(x_{v_j}) = 0 \end{cases} \tag{6.8}$$

If the task is assigned to the IoT device (i.e., $IS(x_{v_j}) = 1$), the energy consumption of the IoT device is equal to the amount of time that it processes the task multiplied by the CPU power of IoT device P^{cpu}. However, if the task is assigned to the other servers for processing (i.e., $IS(x_{v_j}) = 0$), the energy consumption of the IoT device depends on its idle time and corresponding idle power P^{idle}. The $\omega^{input}_{x_{v_j}}$ depends on the assigned servers to current task (i.e., x_{v_j}) and its predecessors, and is calculated as what follows:

$$\omega^{input}_{x_{v_j}} = \begin{cases} \psi^{input}_{x_{v_j}} \times P^{tra}, & IS(x_{v_j}) = 1 \\ \max(IS(x_{v_i}) \times (\frac{e^w_{i,j}}{b_{x_{v_i},x_{v_j}}} + l_{x_{v_i},x_{v_j}}) & IS(x_{v_j}) = 0 \\ \times SS(x_{v_i}, x_{v_j})) \times P^{tra} + (\psi^{idle} \times P^{idle}), & \\ \forall v_i \in \mathcal{P}(v_j), & \end{cases} \tag{6.9}$$

where $IS(x_{v_j})$ and $IS(x_{v_i})$ demonstrates whether the current task v_j and/or its parent task $v_i \in \mathcal{P}(v_j)$ in each edge are assigned to the IoT device or not, respectively. It is important to note that the transmission energy consumption for each edge in DAG is only considered when one of the tasks is placed on the IoT device. Hence, if the current task is assigned to the IoT device (i.e., $IS(x_{v_j}) = 1$), the $\omega^{input}_{x_{v_j}}$ depends on the $\psi^{input}_{x_{v_j}}$.

However, if the current task is not assigned to the IoT device (i.e., $IS(x_{v_j}) = 0$), it is possible that the predecessor tasks of the current task (i.e., $\forall v_i \in \mathcal{P}(v_j)$) are previously assigned to the IoT device, and hence the IoT device should forward the data to the server on which the current task is assigned (which incurs energy consumption). If none of the tasks are assigned to the IoT device for local execution, the IoT device is in its idle state. Besides, P^{tra}, ψ^{idle} represent the transmission power of the IoT device and its idle time, respectively. Similar to [122, 123, 219], we used constant values for P^{tra}, ψ^{idle}, however, these parameters also can be dynamically configured.

The main goal of the energy consumption model is to find the best-possible placement configuration for the IoT application so that its energy consumption becomes minimized. Assuming an IoT application consists of L tasks, the energy consumption model is defined as:

$$\Omega(\mathcal{X}) = \min(\sum_{j=1}^{L} CP(v_j) \times \omega_{x_{v_j}}) \tag{6.10}$$

Weighted cost model

The weighted execution cost of task v_j is defined based on its assigned server x_{v_j}:

$$\phi_{x_{v_j}} = (w_1 \times \psi_{x_{v_j}}) + (w_2 \times \omega_{x_{v_j}}) \tag{6.11}$$

where $\psi_{x_{v_j}}$ and $\omega_{x_{v_j}}$ refer to the execution time and energy consumption for the execution of task v_j. Moreover, the w_1 and w_2 are control parameters to represent the importance of decision parameters in weighted execution cost of each task. Also, the weighted cost of each task can be changed to execution time or energy consumption cost of each task by assigning $w_1 = 1, w_2 = 0$ or $w_1 = 0, w_2 = 1$, respectively.

Finally, the goal of weighted cost model is to find the best placement configuration for tasks of an IoT application while minimizing the weighted cost of parameters. In this work, we consider execution time of IoT applications and energy consumption of IoT devices as decision parameters, however, this weighted cost can be extended using other decision parameters. The weighted cost model is defined as:

$$\min \Phi(\mathcal{X}) = \min w_1 \times \Psi(\mathcal{X}) + w_2 \times \Omega(\mathcal{X}) \tag{6.12}$$

s.t.

$$C1: \ Size(x_{v_j}) = 1, \ \forall x_{v_j} \in \mathcal{X}$$
$$C2: \ \Phi(v_i) \leq \Phi(v_i + v_j), \ \forall v_i \in \mathcal{P}(v_j)$$
$$C3: \ v_j^{ram} \leq RAM(x_{v_j}), \ \forall v_j \in \mathcal{V}$$
$$C4: w_1 + w_2 = 1$$

where $\Psi(\mathcal{X})$, $\Omega(\mathcal{X})$ are obtained from Eq. 6.6 and Eq. 6.10, respectively. Besides, w_1 and w_2 are control parameters for execution time and energy consumption, by which the weighted cost model can be tuned. $C1$ denotes that each task can only be assigned to one server at a time for processing. Moreover, $C2$ states that the task v_j can only be executed after the execution of its predecessors, and hence the cumulative execution cost of v_j is always larger or equal to execution cost of its predecessors' tasks [123]. Besides, $C3$ states that the assigned server to the task v_j should have sufficient amount of available RAM $RAM(x_{v_j})$ for the processing. Also, $C4$ defines a constraint on the values of control parameters. These constraints are also valid for execution time and energy consumption models. Moreover, the weighted cost model can be changed to execution time or energy consumption model by assigning $w_1 = 1, w_2 = 0$ or $w_1 = 0, w_2 = 1$, respectively.

Since the application placement problem in heterogeneous environments is an NP-hard problem [120], the problem's complexity grows exponentially as the number of heterogeneous servers and/or the number of tasks within an IoT application increases. Thus, the optimal policy of the application placement problem cannot be obtained in polynomial time by iterative approaches. The existing application placement techniques are mostly based on heuristics, rule-based policies, and approximation algorithms [97, 119]. Such techniques work well in general cases, however, they cannot fully adapt to dynamic computing environments where the effective parameters of workloads and computational resources continuously change [119, 220]. To address these issues, DRL-based scheduling/placement algorithms are promising avenues for dynamic optimizations of the system [97, 212].

6.4 Deep Reinforcement Learning Model

The DRL is a general framework that incorporates deep learning to solve decision-making problems with high-dimensional inputs. Formally, learning problems in DRL can be modeled as Markov Decision Processes (MDP), which is extensively used in sequential stochastic decision making problems. A learning problem can be defined by a tuple $< \mathbb{S}, \mathbb{A}, \mathbb{P}, \mathbb{R}, \gamma >$, in which $\mathbb{S}$ and $\mathbb{A}$ denote the state and action spaces, respectively. $\mathbb{P}$ illustrates the state transition probability, and $\mathbb{R}$ is a reward function. Finally, $\gamma \in [0, 1]$ is a discount factor, determining the importance of future rewards. We suppose that the time horizon is separated into multiple time periods, called time steps $t \in \mathbb{T}$. The DRL agent interacts with the environment, and in each time step t, it perceives the current state of the environment s_t, and selects an action a_t based on its policy $\pi(a_t|s_t)$, mapping states to actions. Considering the selected action a_t, the agent receives a reward r_t from the environment, and it can perceive the next state s_{t+1}. The main goal of the agent is to find a policy in order to maximize the expected total of future discounted reward [216]:

$$\mathbb{V}^{\pi}(s_t) = \mathbb{E}_{\pi}[\sum_{t \in \mathbb{T}} \gamma^t r_t] \tag{6.13}$$

where $r_t = \mathbb{R}(s_t, a_t)$ is the reward at time step t, and $a_t \sim \pi(.|s_t)$ is the generated action at time step t by following the policy π. Moreover, when DNN is used to approximate the function, the parameters are denoted as θ.

Considering the application placement in Fog computing environments, we define the main concept of the DRL for our problem as what follows:

- **State space $\mathbb{S}$:** In our application placement problem, the state is the observations of the agent from the heterogeneous Fog computing environment. Thus, the state at time step t (s_t) consists of information about all heterogeneous servers (such as processing speed of CPU, number of CPU cores, CPU utilization, access Bandwidth (i.e., data rate) of servers, access latency of servers, and CPU, transmission, and idle power consumption values of IoT device). For the rest of the servers, their power consumption values are ignored as we only consider energy consumption from IoT devices' perspective [123]. If for each server we have n features to represent its information, the feature vector of all M servers at time step t ($FV_t^{\mathcal{M}}$) can be

presented as:

$$FV_t^{\mathcal{M}} = \{f_i^{m^{y,z}} | \forall m^{y,z} \in \mathcal{M}, 1 \leq \ i \leq n\} \tag{6.14}$$

where $f_i^{m^{y,z}}$ shows the ith feature of the server $m^{y,z}$. Moreover, s_t contains the information about the current task to be processed within a DAG of an IoT application (such as computation requirements of the task, required RAM, amount of output data per parent task, and current placement configuration of all tasks). Since we consider that tasks are sorted and their dependencies are satisfied before their execution, the current placement configuration of tasks contains the information regarding assigned servers to all previous tasks. The values of unprocessed tasks are set to -1. If we assume that each task has b features, the feature vector of task v_j ($FV_t^{v_j}$) can be represented as:

$$FV_t^{v_j} = \{f_i^{v_j} | v_j \in \mathcal{V}, \forall i \ 1 \leq i \leq b\} \tag{6.15}$$

where $f_i^{v_j}$ shows the ith feature of the task v_j. Thus, the system space can be defined as:

$$\mathbb{S} = \{s_t | s_t = (FV_t^{\mathcal{M}}, FV_t^{v_j}), \forall t \in \mathbb{T}\} \tag{6.16}$$

- **Action space $\mathbb{A}$:** Actions are assignments of available servers to tasks of an IoT application. Therefore, the action at time step t (a_t) is equal to assigning a server $m^{y,z}$ to the current task v_j. Considering the placement configuration of each task $x_{v,j}$ in section 6.3.2, a_t can be defined as:

$$a_t = x_{v_j} = m^{y,z} \tag{6.17}$$

Thus, the action space $\mathbb{A}$ can be defined as the set of all available servers, presented as follows:

$$\mathbb{A} = \mathcal{M} \tag{6.18}$$

- **Reward function $\mathbb{R}$:** The goal is to minimize the weighted cost model, defined in

Eq. 6.12. To obtain this, we consider Eq. 6.11 as the weighted cost of each task and define the $\mathbb{R}$ as the negative value of Eq. 6.11 if the task can be executed ($done = 1$). Moreover, we define a constant *penalty* value, which is usually a very large negative number [120]. Furthermore, the *penalty* value can be dynamically set based on the goal of the optimization problem and environmental variables. If the selected action by the agent (i.e., server assignment for the current task) cannot be performed due to any reason ($done = 0$), the reward becomes equal to *penalty*. Accordingly, r_t is defined as:

$$r_t = \begin{cases} -\phi_{x_{v_j}} & done = 1 \\ penalty & done = 0 \end{cases} \tag{6.19}$$

6.5 Proposed Distributed DRL-based Framework

To address the challenges of DAG-based application placement in the heterogeneous Fog computing environment, the *X*-DDRL works based on an actor-critic framework, aiming at taking advantage of both value-based and policy-based techniques while minimizing their drawbacks [140].

Actor-critic framework In an actor-critic framework, the policy is directly parameterized, denoted as $\pi(a_t|s_t;\theta)$, and the θ is updated by calculating the gradient ascent on the variance of the expected total future discounted reward (i.e, $\sum_{k=0}^{\infty} \gamma^k r_{t+k}$) and the learned state-value function under policy π (i.e., $\mathbb{V}^{\pi}(s_t)$) [140]. The actor interacts with the environment and receives state s_t, outputs the action a_t based on $\pi(a_t|s_t;\theta)$, and receives the reward r_t and next state s_{t+1}. The critic, on the other hand, uses rewards to evaluate the current policy based on the Temporal Difference (TD) error between current reward and the estimation of the value function $\mathbb{V}(s_t;\theta)$. Both actor and critic use DNNs as their function approximators, which are trained separately. To improve the selection probability of better actions by the actor, the parameters of the actor network are updated using the feedback of the TD-error, while the network parameters of the critic network are up-

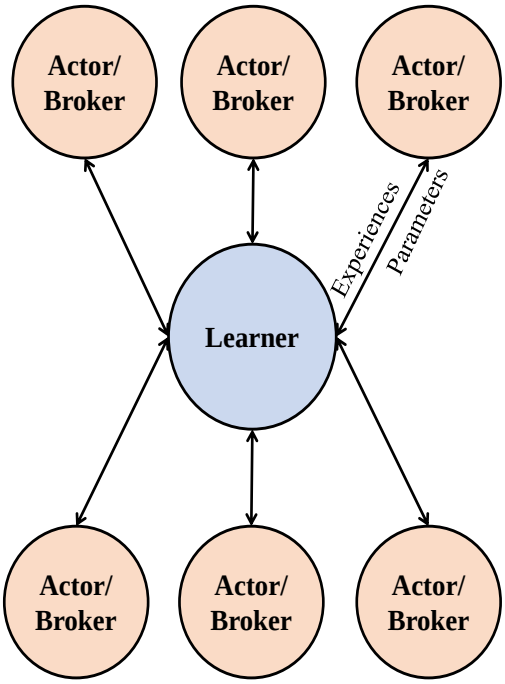
dated to achieve better value estimation. While the actor-critic frameworks work very well in long-term performance optimizations, their learning speeds are slow and they incur huge exploration costs, especially in problems with high dimensional-state space [120]. The distributed learning techniques in which diverse trajectories are generated in parallel can greatly improve the exploration costs and learning speed of actor-critic frameworks.

The *X*-DDRL works based on an actor-learner framework, in which the process of generating experience trajectories is separated from learning the parameters of π and $\mathbb{V}^{\pi}$. Fig 6.2a demonstrates a high-level overview of learner and actors. The distributed actors in Fog computing environments, which can be multiple CPUs within a broker (i.e., FS) or different brokers, interact with their Fog computing environments. Arriving application placement requests to each broker are queued in the $apps_Q$ based on the FIFO policy. As Fig 6.2b depicts, brokers performs pre-scheduling phase for each IoT application. Then, based on features of available servers and current task of selected IoT application, each broker pre-processes the current state and makes an application placement decision. Each broker periodically sends its local experience trajectories to the learner. Besides, the learner updates the target policy π based on collection of received trajectories from different brokers and past trajectories stored in the replay buffer. After each policy update of the learner, brokers update their local policy μ with the policy of the learner π.

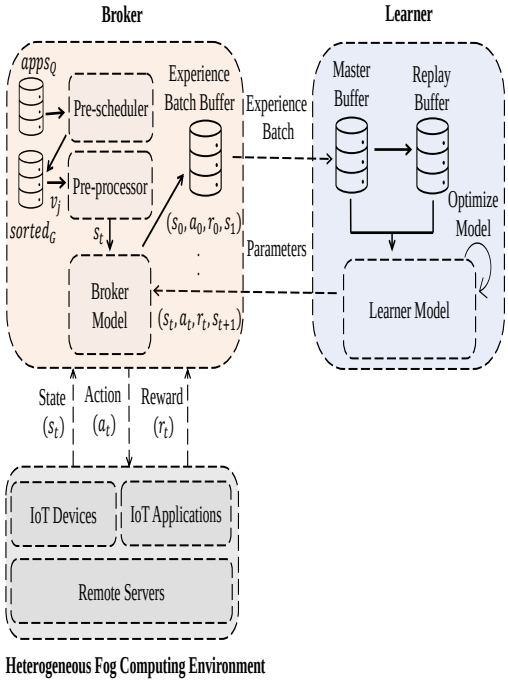
The *X*-DDRL is divided into two phases: pre-scheduling and application placement technique. In the pre-scheduling, tasks of the received IoT application are ranked and sorted in a sequence for the execution. Afterward, for each task of an IoT application, *X*-DDRL makes a placement decision to minimize the execution cost of the IoT application.

6.5.1 *X*-DDRL: Pre-scheduling phase

IoT applications are heterogeneous in terms of the number of tasks per application, the dependency model, and corresponding weights of vertices and edges. Considering the dependency model of an IoT application, tasks should be sorted for execution, so that task v_j cannot be executed before any task $v_i \in \mathcal{P}(v_j)$. Furthermore, there are several



(a) High-level overview of learner and actors/brokers



(b) Communication between learner and each broker

Figure 6.2: An overview of *X*-DDRL framework

tasks that can be executed in parallel, and the order of execution of such parallel tasks are also important and may affect the execution cost of an IoT application. Fig. 6.3 shows a sample IoT application, dependencies among tasks, and parallel tasks with the same colors in each row.

Whenever a broker receives a DAG-based IoT application request from a user, it creates a sequence of tasks for the execution while considering above-mentioned challenges. Tasks within the IoT application are ranked based on the non-increasing order of their rank value. The rank value of a task is defined as:

$$Rank(v_j) = \begin{cases} \widetilde{\phi_{x_{v_j}}} + \max(\widetilde{\phi_{x_{v_i}}}), & \forall v_i \in \mathcal{P}(v_j) \quad \text{if } v_{n,j} \neq exit \\ \widetilde{\phi_{x_{v_j}}}, & \text{if } v_{n,j} = exit \end{cases} \tag{6.20}$$

where $\widetilde{\phi_{x_{v_j}}}$ shows the average weighted execution cost of task $v_{n,j}$ on considering different servers. The rank is calculated recursively by traversing the DAG of the application, starting from the exit module. Using the rank function, the tasks on the critical path

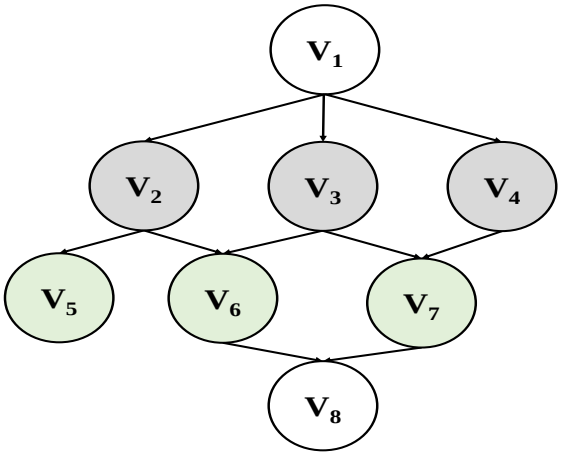


Figure 6.3: A sample IoT application (parallel tasks have same colors in each row)

of DAG (i.e., CP) can also be identified. Hence, not only does the rank function satisfy the dependency among tasks, but it also defines an execution order for tasks that can be executed in parallel. To achieve this, it gives higher priority to tasks that incur higher total execution costs among parallel tasks.

6.5.2 X-DDRL: Application Placement Phase

If we assume that each broker makes placement decisions for tasks of IoT applications, using their local policy μ, for N steps in the time horizon starting at time $i = t$, Algorithm 12 shows how brokers perform application placement decisions and generate experience trajectories. Each broker performs the following steps: At the beginning of each trajectory, the broker updates its policy μ with the policy of the learner (line 3). When broker starts making placement decisions for tasks of a new IoT application G (i.e., when the flag-init=*True*), it receives the current IoT application from the $apps_Q$ (contains all received application requests to this broker) (line 6). Then, the broker performs the pre-scheduling to obtain the sorted list of application' tasks based on the Eq. 6.20 (line 7). Next, the system state is generated using the initial state of the IoT application G and available servers $\mathcal{M}$ (line 8). Moreover, the broker changes the flag-init to *False*, indicating that in the subsequent steps there is no need to re-calculate the ranking and initial state of the G (line 9), and the broker only requires to obtain the current state of

Algorithm 12: The role of each broker/actor

Input : π: The learner policy

/* N: number of steps, μ: the actor's local policy, EBB: expeience batch buffer, $apps_Q$: Queue of all received IoT applications, G: current IoT application */

1 flag-init=True
2 **for** $t = 0$ to ∞ **do**
3 μ=UpdateLocalPolicy(μ, π)
4 **for** $i = t$ to $N + t - 1$ **do**
5 **if** flag-init=True **then**
6 G=$apps_Q$.dequeue()
7 $sorted_G$ = Pre-scheduling (G) % based on Eq. 6.20
8 s_i=ReceiveInitialState(G, $\mathcal{M}$, $sorted_G$)
9 flag-init=False
10 **else**
11 s_i=ReceiveCurrentState()
12 **end**
13 s_i=Pre-processor(s_i)
14 a_i=PlacementEngine(s_i, μ) % calculates the action
%The environment then executes this action
15 r_i=TaskCostCalculator(s_i, a_i) % baed on Eq. 6.19
16 s_{i+1} = BuildNextState(s_i, a_i)
17 EBB.update(s_i, a_i, r_i, s_{i+1})
18 **if** Finish(G) **then**
19 CalculateTotalCost(G) % based on Eq. 6.12
20 flag-init=True
21 **end**
22 **end**
23 **if** size(EBB)==N **then**
24 SendExpeienceToLearner(EBB)
25 **end**
26 **end**

the environment based on Eq. 6.16 (line 11). The current state of the broker's environment s_i consists of feature vectors of servers $FV_t^{\mathcal{M}}$ and the current task of IoT application $FV_t^{v_j}$. The current task of each IoT application is obtained from the ordered sequence of tasks $sorted_G$. Then, the broker pre-processes and normalizes values of the current state (line 13). Considering s_i and current policy μ, an application placement decision (i.e., the assignment of a server for the processing of the current task) is made (line 14). The current task is then forwarded to the assigned server (based on a_i) for processing. After the execution of the task, the broker receives the reward of this action, which is the neg-

ative value of the weighted execution cost of this task Eq. 6.19 (line 15). The next state of the environment is then created using the *BuildNextState* function (line 16). Then, the broker creates an experience tuple (s_i,a_i,r_i,s_{i+1}) and stores it in its local experience batch buffer (line 17). When the broker finishes assignment of servers to all tasks of the current IoT application G, meaning the application placement is done for the current IoT application, the total weighted execution cost of this IoT application is calculated using Eq. 6.12 (line 19). Moreover, the broker sets flag-init to *False* so that the next IoT application in the queue of this broker $apps_Q$ can be served (line 20). After N steps, each broker forwards its experience batch buffer to the learner (lines 23-25). The learner periodically updates its policy (i.e., π) on batches of experience trajectories, collected from several brokers.

Since policies of brokers μ are updated based on the learner's policy (trained on trajectories of different brokers), each broker gets the benefit of trajectories generated by other brokers. It significantly reduces the exploration cost of each broker, and also provides brokers with a more accurate local policy μ. Furthermore, the *X*-DDRL uses an experience-sharing approach, which significantly reduces communication overhead between brokers and learners, in comparison to gradient-sharing techniques such as A3C [216].

Due to the gap between the policy of broker μ (when generating new decisions) and the policy of the learner π in the training time (when the learner estimates the gradients), the learner in the *X*-DDRL uses the off-policy correction method, called V-trace [216], to correct this discrepancy.

- **V-trace off-policy correction method**: We assume that each broker generates an experience trajectory for N steps while following its local policy μ as $(s_t, a_t, r_t)_{t=i}^{i+N}$. The value approximation of state s_i, defined as N-step V-trace target for $\mathbb{V}(s_i)$, is as follows:

$$\overline{\mathbb{V}_i} = \mathbb{V}(s_i) + \sum_{t=i}^{i+N-1} \gamma^{t-i} (\prod_{j=i}^{t-1} c_j) \delta_t \mathbb{V} \tag{6.21}$$

  where $\delta_t \mathbb{V}$ is a TD for $\mathbb{V}$, defined as:

$$\delta_t \mathbb{V} = \rho_t (r_t + \gamma \mathbb{V}(s_{t+1}) - \mathbb{V}(s_t)) \tag{6.22}$$

where $\rho_t = min(\overline{\rho}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)})$ and $c_j = min(\overline{c}, \frac{\pi(a_j|s_j)}{\mu(a_j|s_j)})$ are truncated Importance Sampling (IS) weights, while $\overline{c} \leq \overline{\rho}$. The $\overline{c}$ and $\overline{\rho}$ play different roles in the V-trace. The $\overline{\rho}$ has a direct effect on the value function $\mathbb{V}^{\pi}$ toward which we converge, while $\overline{c}$ has a direct effect on speed of the convergence. Considering ρ, the target policy of the learner π can be defined as:

$$\pi_{\overline{\rho}}(a|s) = \frac{\min(\overline{\rho}\mu(a|s), \pi(a|s))}{\sum_{b \in \mathbb{A}} \min(\overline{\rho}\mu(b|s), \pi(b|s)))} \tag{6.23}$$

We consider: (1) the brokers generate trajectories while following policy μ, (2) the parameterized state-value function under θ as $\mathbb{V}_{\theta}$, (3) the current policy of learner is π_u, and (4) the V-trace target $\overline{\mathbb{V}_i}$ is defined based on Eq. 6.21. The learner updates value parameters θ, at time step i, in the direction of:

$$(\overline{\mathbb{V}_i} - \mathbb{V}_{\theta}(s_i))\nabla_{\theta}\mathbb{V}_{\theta}(s_i) \tag{6.24}$$

Moreover, the policy parameters u are updated in the direction of the policy gradient using Adam optimization algorithm [221]:

$$\rho_i \nabla_u \log(\pi_u(a_i|s_i))(r_i + \gamma\overline{\mathbb{V}_{i+1}} - \mathbb{V}_{\theta}(s_i)) \tag{6.25}$$

Algorithm 13 summarizes the learners' role in the *X*-DDRL. The learner continuously receives and stores experience batches of brokers EB_{broker} and updates the master Buffer MB until the training batch TB becomes full (line 4-10). Then, the learner optimizes the current target policy π based on Eq. 6.24 and 6.25 (line 11). After policy update of the learner, brokers update their local policies μ with the latest policy of the learner π (i.e., brokers set their policies to the new learner policy), and hence, new application placement decisions are made using the updated policy μ in the brokers. The learner in the *X*-DDRL uses the replay buffer RB, which remarkably improves sample efficiency. The *X*-DDRL can easily scale as the number of servers, IoT application requests, and brokers increases, which is a principal factor in highly distributed environments such as Fog computing. If a new broker joins the environment, the broker updates its local policy with the latest policy of the learner, and hence it takes advantage of all trajecto-

Algorithm 13: The role of each learner

Input : EB_{broker}: Experience batch of different brokers

/* $list_{brokers}$: list of brokers, π: the learner's policy, MB: master buffer, MBS: master buffer size, RB: replay buffer, RBS: replay buffer size, TB: training batch, TBS: training batch size */

```
while True do
    flag-training=False
    MB=∅
    while flag-training==False do
        MB.update(EB_broker)
        if TBS ≤ MBS + RBS then
            TB=BuildTrainBatch(MB, RB)
            flag-training==True
        end
    end
    OptimizeModel(TB) % based on Eq. 6.24, 6.25
    UpdateBrokers(list_brokers, π)
end
```

ries that were previously generated by other brokers. Besides, it generates new sets of trajectories which help better diversify the trajectories of the learner. If the number of servers in the environment increases, distributed brokers quickly generate new sets of trajectories, and accordingly, the learner can update its target policy promptly. Such a collaborative distributed broker-learner architecture not only significantly improves the exploration costs but also improves the convergence speed. The other improvement in the *X*-DDRL is using RNN layers since they can accurately identify highly non-linear patterns among different input features, resulting in significant speedup in the learner [97, 222].

6.6 Performance Evaluation

This section first describes the experimental setup, used to evaluate our technique and baseline algorithms. Next, the hyperparameters of our proposed technique *X*-DDR are discussed. Finally, we study the performance of *X*-DDRL and its counterparts in detail.

6.6.1 Experimental Setup

To evaluate the performance of the *X*-DDRL, we use both simulation environment and testbed, which their specification are provided in what follows.

Simulation setup

We developed an event-driven simulation environment in Python using the OpenAI Gym [25] for the application placement in heterogeneous Fog computing environments, similar to [119]. For each of the two learners, we set the number of brokers to 8, which have access to a set of servers, and make application placement decisions accordingly. Hence, we vectorized the Fog computing environment, generated using OpenAI Gym, so that distributed brokers can interact with their Fog computing environments and make application placement decisions in parallel. Unlike prior work [78, 119, 120, 133], we consider a heterogeneous Fog computing environment consisting of IoT devices, resource-constrained FSs, and resource-rich CSs. In Fog computing environment, we used the following server setup, unless it is stated in the experiments: two Raspberry pi 3B (Broadcom BCM2837 4 cores @1.2GHz, 1GB RAM)[1], one Raspberrypi 4B (ARM Cortex-A72 4 cores @1.5GHz, 4GB RAM)[2], and one Jetson Nano (ARM Cortex-A57 4 cores @1.43GHz, 4GB RAM, 128-core Maxwell GPU)[3] as heterogeneous FSs. Besides, to simulate a heterogeneous multi-Cloud environment, we used specifications of six m3.large instances of Nectar Cloud infrastructure (AMD 8 cores @2GHz, 16GB RAM)[4] and two instances of the University of Melbourne Horizon Cloud (Intel Xeon 8 cores @2.4GHz, 24GB RAM, NVIDIA P40 3GB RAM GPU)[5]. For IoT devices, the server type is a single core @1GHz device embedded with 512MB RAM [119]. Besides, the power consumption of each IoT device in processing, idle, and transmission state is 0.5W, 0.002W, and 0.2W, respectively [123]. The bandwidth (i.e., data rate) and latency among different servers and IoT devices are also obtained based on average profiled values from testbed, similar to [97]. Hence, the latency of FSs and CSs are considered as 1ms and 10ms re-

[1]https://www.raspberrypi.org/products/raspberry-pi-3-model-b
[2]https://www.raspberrypi.org/products/raspberry-pi-4-model-b
[3]https://developer.nvidia.com/embedded/jetson-nano-developer-kit
[4]https://nectar.org.au/
[5]https://people.eng.unimelb.edu.au/lucasjb/horizon/

spectively, similar to [97]. The bandwidth between IoT devices and FSs is randomly selected between 10-12MB/s, while the bandwidth between IoT devices and FSs to the CSs is randomly selected between 4-8 MB/s, similar to [122]. Although we obtained these values based on testbed experiments, they are referred to some similar works as well to show the credibility of these values. Also, both w_1 and w_2 are set to 0.5, meaning that the importance of execution time and the energy consumption is equal in the results. However, these parameters can be adjusted based on the users' requirements and network conditions.

Many real-world IoT applications can be modeled by DAGs with a different number of tasks and dependency models. Hence, we generated several synthetic DAG sets with a different number of tasks and dependency models to represent scenarios where IoT devices generate heterogeneous DAGs with different preferences, similar to [119, 223]. The dependency model of each DAG can be identified using three parameters: number of tasks within an application L, fat that controls the width and heights of a DAG, and *density* that identifies the number of edges between different levels of the DAG. Accordingly, we generated different DAG datasets, where each dataset contains 100 DAGs with a similar number of tasks, fat, and density while the weights are randomly selected to represent heterogeneous task requirements in IoT applications with the same DAG structure. To generate heterogeneous DAG datasets, we set task numbers $L \in \{10, 15, 20, 25, 30, 35, 40, 45, 50\}$, $fat \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$, and $density \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$. To illustrate, one dataset of DAGs is $L = 10$, $fat = 0.4$, and $density = 0.4$, containing 100 DAGs. Accordingly, for each task number L, we have 25 different combinations of fat and *density*, resulting in 25 different topologies and 2500 DAGs. Finally, the simulation experiments are all performed on an instance of Horizon Cloud with the above-mentioned specifications.

Testbed setup

To evaluate the performance of *X*-DDRL in a real-world scenario, we created a testbed, similar to [37, 120]. The type of servers are the same as simulation setup while the number of servers of each type is as follows: two Raspberry pi 3B, one Raspberry pi 4B,

one Jetson Nano, one instance of Horizon Cloud, and six m3.large instances of Nectar Cloud infrastructure. As IoT devices, we created several single-core VMs within a PC (HP Elitebook 840 G5 with Intel Core i7-8550U 8 cores @2GHz and 16GB RAM). These VMs are used to send application placement requests, using described DAG datasets, to the brokers. Moreover, to estimate the energy consumption of IoT devices, we used computing power, transmission power, and idle power as discussed in section 6.6.1, similar to the approach in [120]. For the connectivity, we set up a virtual network using VPN among IoT devices, FSs, and CSs, as described in [1, 164]. Due to the limited CPU and RAM of the IoT devices' VMs, they can send application placement requests, using a message-passing protocol (implemented using HTTP requests), to the broker that is the Jetson Nano in this testbed. The broker runs a multi-threaded server application that receives application placement requests from different IoT devices and puts them in the queue based on the FIFO policy. The broker dequeues requests and makes placement decisions for tasks according to its policy μ. According to the placement configuration for each IoT application, each server that receives a task for processing assigns that task to one of its threads for processing. The thread is kept busy according to the weight of task and processing speed of the server. After the execution of each task, the size of output results that should be forwarded to the children tasks is obtained based on the weights of the task's outgoing edges in each DAG. Since weights of edges in each DAG (i.e., data to be transferred between tasks) are different, we generate files with different sizes to represent the weights on edges. Finally, the broker logs the execution cost of each IoT application and all of its constituent tasks in terms of selected evaluation metrics.

Baseline algorithms

We evaluate the performance of the *X*-DDRL with a greedy heuristic algorithm, and two DRL-based techniques from the literature that proposed DRL-based solutions for DAG-based IoT applications. In what follows, we briefly describe how these techniques are implemented, while their detailed specifications are provided in section 6.2.

- PPO-RNN: It is the extended and adapted version of the technique proposed in [119][6]. We extended this technique so that it can be used in multi-objective scenarios to minimize the weighted cost of execution. Besides, this technique is extended to be used in heterogeneous Fog computing environments where several IoT devices, FSs, and CSs are available. This technique uses PPO as its DRL framework while the networks of the agent are wrapped by the RNN. Besides, we used the same hyperparameters as [119].

- PPO-No-RNN: This technique is the same as PPO-RNN, while the networks are not wrapped by the RNNs.

- Double-DQN: Many works in the literature uses standard Deep Q-Learning (DQN) based RL approach such as [78, 99, 132, 133]. We implemented the optimized Double-DQN technique with an adaptive exploration for application placement in heterogeneous Fog computing environments[7]. The hyperparameters of this technique are set based on [78], which is a DQN-based application placement technique for DAG-based IoT applications.

- Greedy: In this technique, tasks are greedily assigned to the servers if their execution cost is less than the estimated local execution cost, similar to [119].

6.6.2 *X*-DDRL Hyperparameters

In the implementation of *X*-DRRL, where the standard implementation of IMPALA is used[8], the DNN structure of all agents is similar, consisting of two fully connected layers followed by two LSTM layers as recurrent layers. Moreover, we performed a grid search to tune hyperparameters. According to tuning experiments, we set the learning rate lr to 0.01, the discount factor γ to 0.99. Besides, values of $\overline{\rho}$ and $\overline{c}$, controlling the performance of V-trace are set to 1 [216] to obtain the best result. Table 6.2 summarizes the setting of hyperparameters.

[6]https://github.com/linkpark/metarl-offloading
[7]https://docs.ray.io/en/master/
[8]https://docs.ray.io/en/master/

Table 6.2: The DNN and training hyperparameters

Parameter	Value	Parameter	Value
Fully Connected layers	2	Learning Rate lr	0.01
LSTM Layers	2	Discount Factor γ	0.99
Optimization Method	Adam	V-trace $\overline{\rho}$	1
Activation Function	Tanh	V-trace $\overline{c}$	1

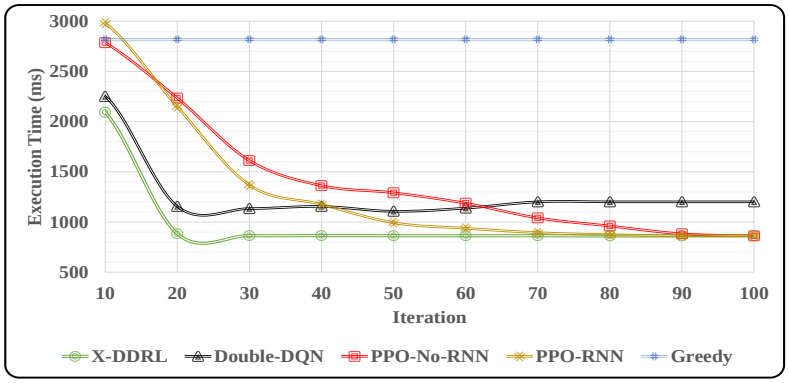
6.6.3 Performance Study

In this section, four experiments are conducted to evaluate and compare the performance of *X*-DDRL with other techniques in terms of weighted execution cost, execution time of IoT applications, and energy consumption of IoT devices.

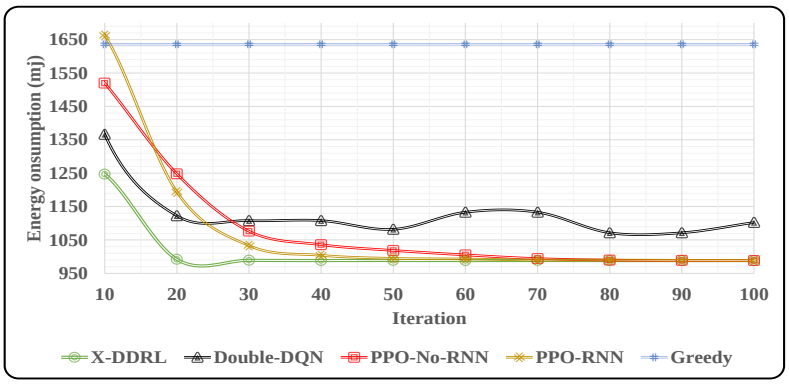
Execution cost vs policy update analysis

In this experiment, we study the performance of application placement techniques in different iterations of the policy updates. We consider two scenarios for datasets of IoT applications to analyze how efficiently these techniques can extract features of different datasets of IoT applications and optimize their target policy. In the first scenario, we consider the number of tasks within IoT applications $L = 30$. Hence, 25 datasets of IoT applications with the same task number and different fat and *density* are used, among which 20 datasets are used for the training and 5 datasets are used for the evaluation. In the second scenario, for the training $L \in \{10, 15, 25, 30\}$ while for the evaluation $L = 20$. Therefore, the training and evaluation are performed on datasets with a different number of tasks. Fig 6.4 and Fig 6.5 show the obtained results of this study in terms of the average execution time of IoT applications, the energy consumption of IoT devices, and weighted cost for the above-mentioned two scenarios.

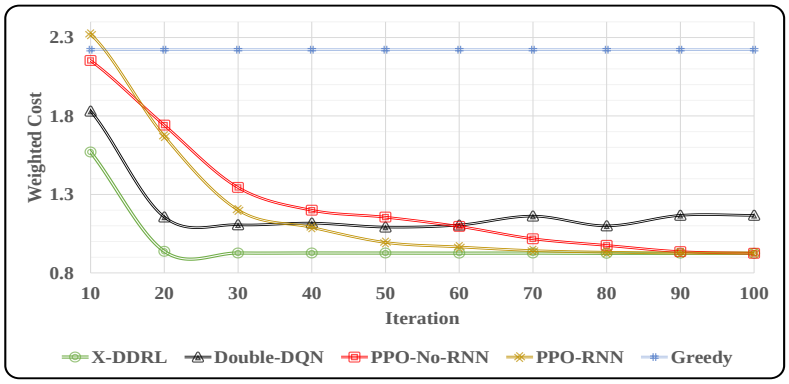
As Fig. 6.4 and Fig. 6.5 show, the average execution cost of all techniques, except the greedy, decreases in different scenarios as the iteration number increases. However, the *X*-DDRL converges faster and to better placement solutions in comparison to other techniques. This is mainly because the V-trace function embedded in the *X*-DDRL uses n-step state-value approximation rather than 1-step state-value approximation [216], im-



(a) Execution time



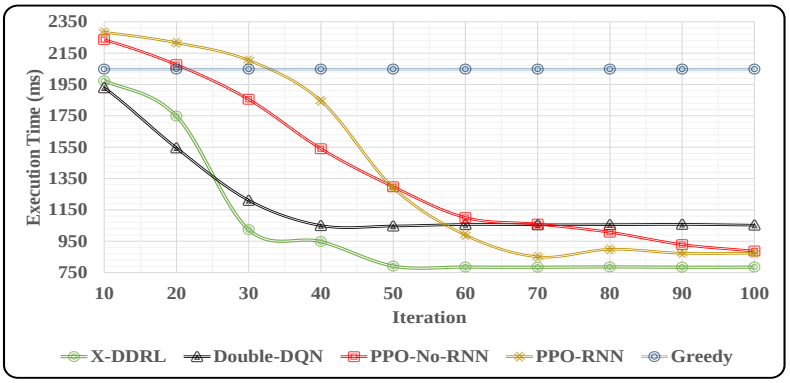
(b) Energy consumption



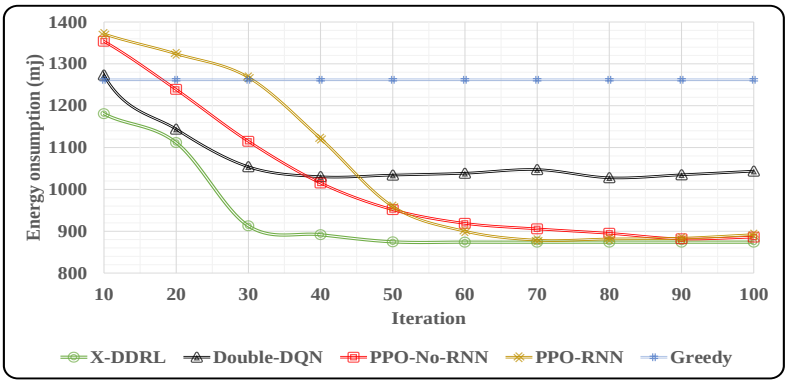
(c) Weighted cost

Figure 6.4: Execution cost vs policy update analysis: Scenario 1, training and evaluations are performed on datasets where $L = 30$

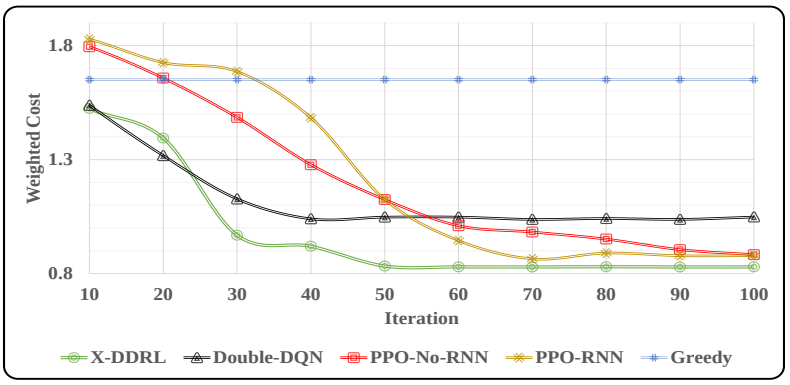
proving convergence speed of *X*-DDRL to better solutions. Moreover, trajectories generated by distributed brokers are diverse, leading to a more efficient learning process. The



(a) Execution time



(b) Energy consumption



(c) Weighted cost

Figure 6.5: Execution cost vs policy update analysis: Scenario 2, training is performed on datasets where $L \in \{10, 15, 25, 30\}$ and the evaluation is performed on datasets where $L = 20$.

execution cost of the greedy technique is fixed and does not change with different iteration numbers, but it can be used as a baseline technique to compare the performance of DRL-based techniques. The convergence speed of PPO-RNN and PPO-NO-RNN techniques is slower than the Double-DQN technique however, they finally converge to better placement solutions. In addition, the obtained results of the second scenario (Fig. 6.5a, 6.5b, 6.5c) shows that all DRL-based techniques has lower convergence speed in comparison to the obtained results of first scenario (Fig. 6.4a, 6.4b, 6.4c). However, still *X*-DDRL outperforms other techniques in terms of execution time, energy consumption, and weighted cost. This proves that the *X*-DDRL can more efficiently adapt itself with different DAG structures (i.e., task numbers, and dependency model), and hence it makes better application placement decisions in unforeseen scenarios.

System size analysis

In this experiment, the effect of different numbers of servers on application placement techniques is studied. The number of candidate servers has a direct effect on the complexity of application placement problems because the larger number of servers leads to a bigger search space. Hence, to analyze the performance of *X*-DDRL, the default number of servers in this experiment is multiplied by two and four; i.e, we have 24 and 48 servers respectively. Moreover, in this experiment, the training and evaluation datasets are specified as the same as the first scenario in Section 6.6.3; i.e., a total of 25 datasets where $L = 30$ and different fat and *density* values. Due to the space limit and the fact that patterns for execution time, energy consumption, and weighted cost were roughly the same, only the obtained results from the weighted cost are provided in this experiment.

Fig. 6.6 shows the weighted cost of different techniques, where brokers in the system have access to 24 and 48 candidate servers when making application placement decisions. It is crystal clear that the weighted cost of the greedy technique is steady for 24 and 48 servers as the number of iterations increases. All DRL-based techniques perform better than greedy technique either when the number of servers is 24 or 48. Also, it can be seen that the weighted execution costs of techniques are higher when

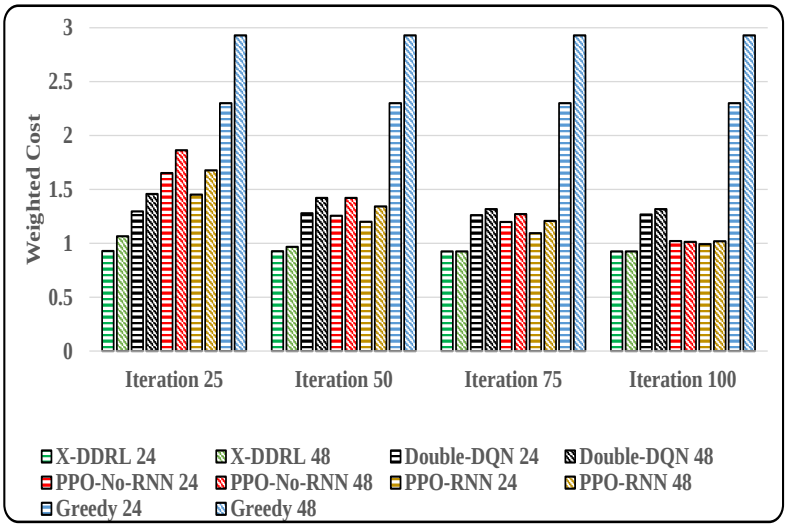


Figure 6.6: System size analysis

the number of servers is 48 than weighted costs when the servers' number is 24. As the number of iterations increases, the DRL-based techniques can more accurately make placement decisions, leading to less weighted execution cost. However, the *X*-DDRL always outperforms other techniques and converges faster to better solutions. It shows that the *X*-DDRL has better scalability when the system size grows. This helps *X*-DDRL to make better application placement decisions in a fewer number of iterations. Among other DRL-based techniques, PPO-RNN performs better than PPO-No-RNN and Double-DQN and makes better placement decisions as the iteration numbers increases.

Speedup and placement time overhead analysis

In this section, we study the speedup and placement time overhead of different DRL-based techniques. We follow the same experimental setup as the first scenario in Section 6.6.3. We define the average Placement Time Overhead (PTO) as the average required amount of time for each technique to make an application placement decision divided by the average local execution time of IoT applications on IoT devices. To obtain the local execution time of IoT applications on IoT devices, we assume that tasks

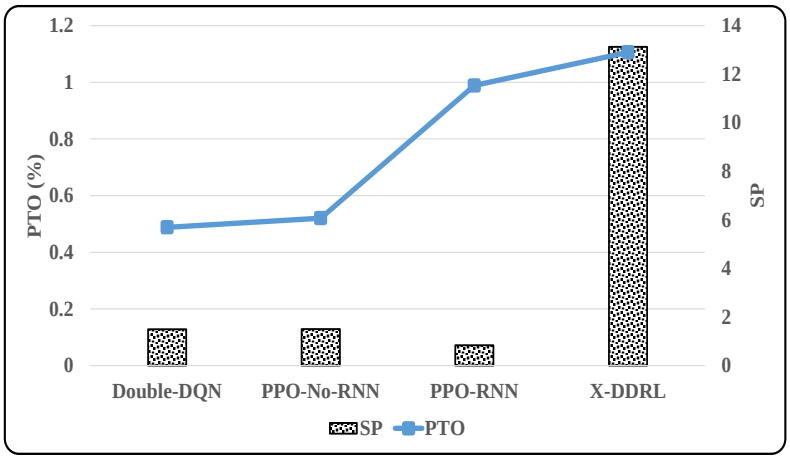


Figure 6.7: Placement time overhead and speedup analysis

within an IoT application are executed sequentially, similar to [123]. Besides, we define the time taken by the *X*-DDRL technique with one broker to reach the value 1.1 from the weighted execution cost as $Time_R$. The reason why 1.1 is considered as the reference weighted execution cost is that this value is the minimum weighted execution cost that all DRL-based techniques can obtain. Moreover, the time taken by each technique to reach the reference weighted execution cost is defined as $Time_T$. Accordingly, similar to [97], the Speedup value of each technique (SP) is defined as $SP = \frac{Time_R}{Time_T}$.

Fig 6.7 shows results of PTO and SP for all DRL-based techniques. The placement time overhead of techniques using RNN (i.e., *X*-DDRL and PPO-RNN) is usually higher than techniques that do not use RNN (i.e., Double-DQN, and PPO-No-RNN). The PTO of the *X*-DDRL is higher than other DRL-based techniques by less than 1% in the worst-case scenario, which is not significantly large. However, the obtained results of SP show that *X*-DDRL performs 8 to 16 times faster than other techniques. Hence, considering the speedup performance and execution cost results of the *X*-DDRL, its placement time overhead is negligible, and *X*-DDRL can more efficiently perform application placement decisions compared to other techniques for heterogeneous Fog computing environments.

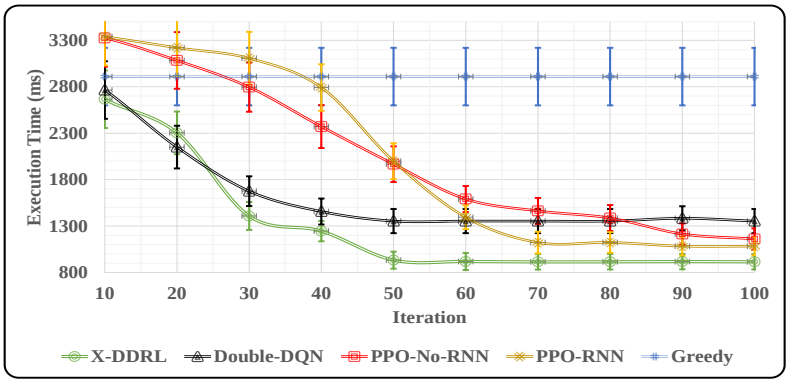
Evaluation on Testbed

To evaluate the performance of *X*-DDRL in real-world scenarios, we conducted experiments on the testbed whose configuration is discussed earlier in Section 6.6.1. In this experiment, for the training $L \in \{30, 35, 45, 50\}$ while for the evaluation $L = 40$.

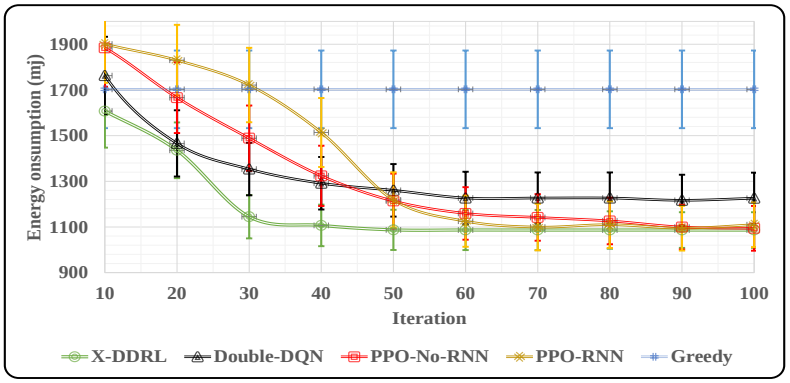
Fig 6.8 shows the execution cost of different techniques in terms of execution time, energy consumption, and weighted cost by 95% confidence interval. It can be observed that, similar to the simulation results, *X*-DDRL can outperform other techniques in terms of execution time, energy consumption, and weighted cost. Moreover, even after 100 iterations, where all techniques converged, there are no techniques that obtain better results in comparison to *X*-DDRL. It demonstrates that not only does *X*-DDRL converge faster, and its training time is significantly less than other techniques, but it also provides better results. As the results depict, the optimized Double-DQN technique converges faster than PPO-RNN and PPO-No-RNN, but it cannot obtain results as well as them. Overall, compared to converged results of other DRL-techniques, achieved results of *X*-DDRL show an average performance gain up to 30%, 11%, and 24% in terms of execution time, energy consumption, and weighted cost, respectively.

6.7 Summary

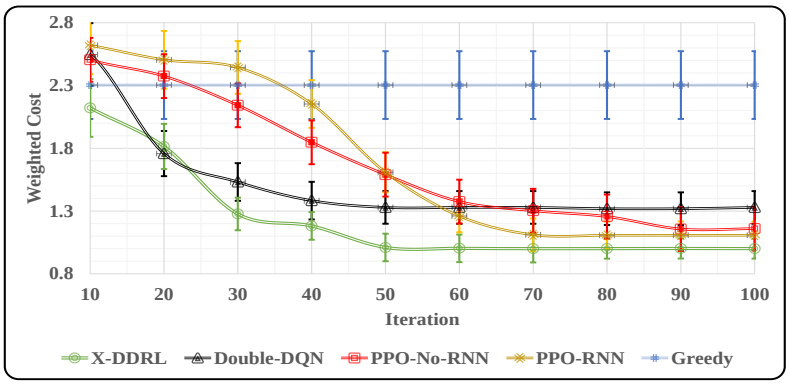
In this chapter, a distributed DRL-based technique, called *X*-DDRL, is proposed to efficiently solve the application placement problem of DAG-based IoT applications in heterogeneous Fog computing environments, where Edge and Cloud servers are collaboratively used. First, a weighted cost model for optimizing the execution time and energy consumption of IoT devices with DAG-based applications in heterogeneous Fog computing environments is proposed. Besides, a pre-scheduling phase is used in the *X*-DDRL, by which dependent tasks of each IoT application are prioritized for execution based on the dependency model of the DAG and their estimated execution cost. Moreover, we proposed an application placement phase, working based on the IMPALA framework for the training of distributed brokers, to efficiently make application placement decisions in a timely manner. Distinguished from existing works, the *X*-DDRL



(a) Execution time



(b) Energy consumption



(c) Weighted cost

Figure 6.8: Evaluation on testbed

can rapidly converge well-suited solutions in heterogeneous Fog computing environments with a large number of servers and users. The effectiveness of *X*-DDRL is analyzed through extensive simulation and testbed experiments while comparing with the

state-of-the-art techniques in the literature. The obtained results indicate that *X*-DDRL performs 8 to 16 times faster than other DRL-based techniques. Besides, compared to other DRL-based techniques, it achieves a performance gain up to 30%, 11%, and 24% in terms of execution time, energy consumption, and weighted cost, respectively.

This chapter proposed a machine learning-based scheduling technique based on DDRL for the dynamic placement of IoT applications. In the next chapter, we investigate building a software system for the scheduling of real-time IoT applications.

Chapter 7

A Software System for Scheduling IoT Applications

Chapter 5 introduced a scheduling technique to minimize the execution cost of real-time DAG-based IoT applications. This chapter presents the implementation of the proposed ranking-based scheduling technique using our FogBus2 framework in an environment consisting of multiple Cloud datacenters and Fog servers. We also extend the FogBus2 framework to integrate new scheduling techniques and implement several new containerized applications to be integrated with the FogBus2 framework. The design and implementation of the framework and IoT applications in this environment followed by evaluation and validation are described in detail in this chapter.

7.1 Introduction

There are different Cloud Service Providers (CSPs) with a wide variety of services, where each CSP provides a particular set of services such as computing, database, and data analysis in an optimized way. Hence, no CSP can satisfy the full functional requirements of different IoT applications in an optimized manner [224]. As a result, each IoT application can be particularly serviced by a specific CSP or simultaneously by different CSPs, which is often called hybrid Cloud computing [224]. Although a hybrid Cloud computing platform provides IoT devices with unlimited and diverse computing and storage resources, CSs are residing multi-hops away from IoT devices, which incurs high prop-

This chapter is derived from:

- **Mohammad Goudarzi**, Qifan Deng, and Rajkumar Buyya, "Resource Management in Edge and Fog Computing using FogBus2 Framework", *Managing Internet of Things Applications across Edge and Cloud Data Centres*, Rajiv Ranjan, Karan Mitra, Prem Prakash Jayaraman, Albert Y. Zomaya (eds), ISBN: 978-1785617799, IET Press, Hertfordshire, UK, June 2022.

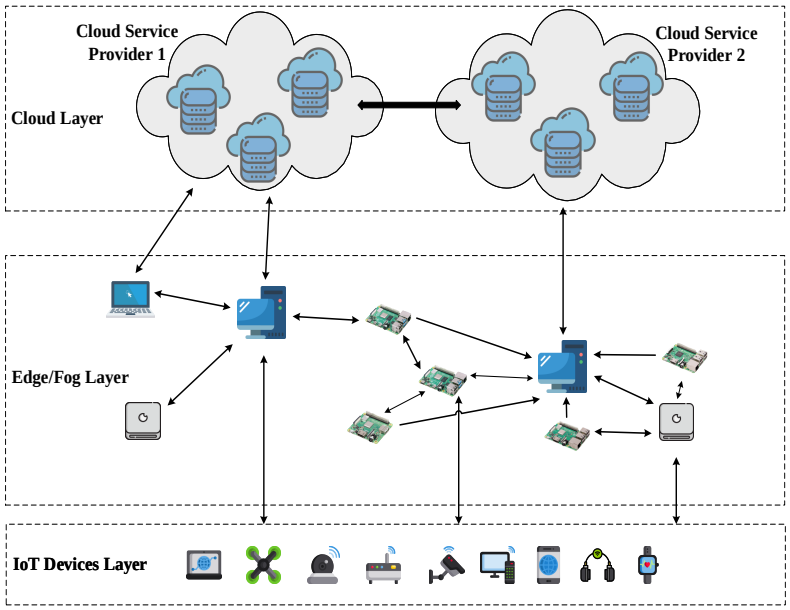


Figure 7.1: Heterogeneous computing environment containing multiple Cloud servers, Fog servers, and IoT devices

agation and queuing latency. Thus, CSs cannot solely provide the best possible services for latency-critical and real-time IoT applications (e.g., intelligent transportation, smart healthcare, emergency, and real-time control systems) [3, 225]. Besides, forwarding the huge amount of data generated by distributed IoT devices to CSs for processing and storage may overload the CSs [30].

In Fog computing environments, the geographically distributed and heterogeneous Fog servers (FSs) (e.g., access points, smartphones, Raspberry pis (Rpi)), situated in the vicinity of IoT devices, can be used for processing and storage of IoT devices' data. These FSs can be accessed with lower latency, which makes them a potential candidate for latency-critical IoT applications, and reduce the traffic of the network's backbone [163]. However, the computing and storage resources of FSs are limited compared to CSs, so they cannot efficiently execute computation-intensive tasks. Therefore, to satisfy the resource and Quality of Service (QoS) requirements of diverse IoT-enabled systems, a seamlessly integrated computing environment with heterogeneous Fog and different Cloud infrastructures is required, as depicted in Fig. 7.1.

The computing and storage resources in such an integrated environment are highly

heterogeneous in terms of their architecture, processing speed, RAM capacity, communication protocols, access bandwidth, and latency, just to mention a few. Furthermore, there are a wide variety of IoT-enabled systems with various QoS and resource requirements. Accordingly, to satisfy the requirements of IoT applications in such an integrated environment, scheduling and resource management techniques are required to dynamically place incoming requests of IoT applications on appropriate servers for processing and storage [226]. In order to develop, test, deploy, and analyze different IoT applications and scheduling and resource management techniques in real-world scenarios, lightweight and easy-to-use frameworks are required for both researchers and developers. There are some existing frameworks for integrating IoT-enabled systems with Edge and Fog computing such as [226–232]. However, they only focus on one aspect of IoT-enabled systems in Edge and Fog computing and often do not support distributed containerized applications.

In this chapter, we extend our FogBus2 framework [1], an open-source python-based, distributed, and containerized framework supporting distributed execution of containerized IoT applications. We extend this framework with a new scheduling technique. Besides, we design and implement new DAG-based and containerized IoT applications and integrate them with the FogBus2 framework. Also, we follow best practices to design a heterogeneous computing environment, consisting of multi-Cloud, multiple FSs, and IoT devices, to evaluate the extended framework.

The remainder of this chapter is organized as follows: Section 7.2 presents the details of framework's design and its implementation, where we describe main components of FogBus2 framework, communication protocol, extensions to scheduling policies, and several containerized IoT applications in subsections 7.2.1, 7.2.2, 7.2.3, and 7.2.4, respectively. Section 7.3 presents the design of our integrated computing environment. Evaluation and validation of the system are conducted in Section 7.4. Finally, Section 7.5 concludes this chapter.

7.2 Extended Framework's Design and Implementation

FogBus2[1] [1] is a container-based framework based on docker containers, developed in Python. To enable the integration of various IoT application scenarios in highly heterogeneous computing environments, FogBus2's components can be simultaneously executed on one or multiple distributed servers in any computing layer. This feature significantly helps researchers and developers in the development and testing phases because they can develop, test, and debug their desired IoT applications, scheduling, and resource management policies on one or a small number of servers. Furthermore, in the deployment phase, they can run and test their IoT applications, scheduling, and resource management techniques on an unlimited number of servers.

7.2.1 Main Components

FogBus2 consists of five containerized components, namely *User*, *Master*, *Actor*, *Task Executor*, and *Remote Logger*. Among these components, the *User* should run on IoT devices or any servers that directly interact with users' sensory or input data. The rest of the components can run on any server with sufficient resources. Each of the containerized components contains several sub-components (sub-C) with specific functionalities. Fig. 7.2 presents FogBus2's main components and their respective sub-Cs. Since the components of the FogBus2 can run on geographically distributed servers, a *message handler* Sub-C is embedded in each component to handle sending and receiving of messages. In what follows, we briefly describe the main functionalities and sub-Cs of each component.

- **User**: This component controls the IoT device's requests for surrogate resources and contains two main sub-Cs, namely *sensor* and *actuator*, alongside with *message handler*. The *sensor* is responsible for capturing and management of raw sensory data and configuring sensing intervals based on IoT application scenarios. Besides, the *actuator*'s main function is collecting the incoming processed data and executing a respective action. The *actuator* can be configured by its users to perform real-time actions based on incoming processed data or periodic actions based

[1] https://github.com/Cloudslab/FogBus2

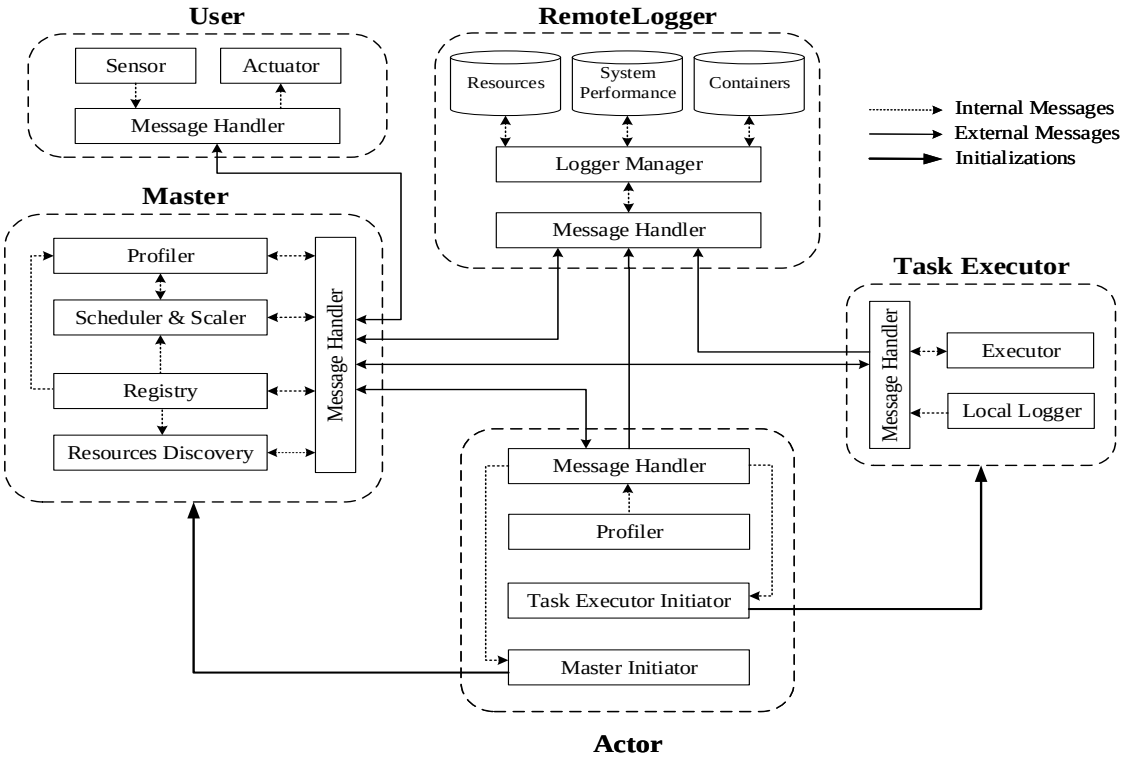


Figure 7.2: FogBus2 main components, sub-components, and their interactions [1]

on a batch of processed data. Researchers and developers can configure the *sensor* and *actuator* to implement different application scenarios.

- **Remote Logger**: The main functionality of this component is to collect and store the contextual information of servers, IoT devices, IoT applications, and networking. It contains the *logger manager* sub-C that can connect to different databases, receives logs of other components, and stores logs in persistent storage. By default, the *Remote Logger* connects to databases to store logs, which is easier to manage and maintain. However, logs can be stored in files as well.

- **Master**: In a real-world computing environment, one or multiple *Master* components may exist. This component contains four main sub-Cs, called *profiler*, *scheduler* & *scaler*, *registry*, and *resource discovery*, alongside with the *message handler*. When the *Master* starts, the *resource discovery* sub-C periodically search the network to find available *Remote Logger*, *Master*, and *Actor* components in the network. If

new components can be found in the network, the *resource discovery* advertise itself to those components, so that they can send a request and register themselves in this *Master*. If the *Master* receives any requests for registration or placement requests from IoT devices (i.e., *User* components), the *registry* sub-C will be called. This sub-C records the information of IoT devices and other components and assigns them a unique identifier. Besides, when the incoming message is a placement request from *User* components, it initiates the *scheduler* & *scaler* sub-C. The *scheduler* & *scaler* sub-C receives the placement request from the *registry* sub-C, the contextual profiling information of all available servers, and networking information from the *profiler* sub-C. Next, if it has enough resources to run the scheduling technique and its placement queue is not very large (configurable queue size), it runs one of the scheduling policies implemented in the FogBus2 framework to assign tasks/containers of the IoT application on different servers for the execution. According to the outcome of the scheduling technique, the *Master* component forwards required information to the selected *Actors* to execute tasks/containers of the IoT application. If due to any reason the *Master* component cannot run its scheduling technique, it runs the *scalability* mechanism to forward the placement request to other available *Master* components, or it initiates a new *Master* component on of the available servers.

- **Actor**: The main responsibility of this component is to start different *Task Executor* components on the server on which it is running. To illustrate, available surrogate servers in the environment should run *Actor* component. Then, these *Actor* components will be automatically discovered and registered by one or several *Master* components in the environment. The *Actor* component profiles the hardware and networking condition of the server on which it is running using the *profiler* sub-C. Besides, when a *Master* component assigns a task of an IoT application to an *Actor* for the execution, it calls the *task executor initiator* sub-C which initiates different *Task Executor* components on the server according to different IoT applications. This sub-C also defines the destination to which the result of each *Task Executor* should be forwarded based on the dependency model of the IoT application. Finally, in order to scale *Master* components in the environment, each *Actor* is

embedded with a *master initiator* sub-C. When an *Actor* receives a scaling message from one of the available *Master* components in the environment, the *master initiator* sub-C will be called. This sub-C starts a *Master* component on the server, which can independently serve incoming IoT application requests. In addition, it can be seen that each server simultaneously can run different components (e.g., *Master*, *Actor*, *Task Executor*, etc) and play different roles.

- **Task Executor**: IoT applications can be represented as a set of dependent or independent tasks or services. In the rest of this chapter, tasks and services are used interchangeably. In the dependent model, the execution of tasks has constraints and each task can be executed when its predecessor tasks are properly executed. In FogBus2, each *Task Executor* component is responsible for the execution of a specific task; i.e., each task or service can be containerized as a *Task Executor*. To illustrate, an IoT application with three decoupled tasks should have three separate *Task Executor* components, so that each *Task Executor* corresponds to one IoT application's task. Considering the granularity level (e.g., task, service) of IoT applications in FogBus2, an application can be deployed on distributed servers for execution. The *Task Executor* consists of two sub-Cs, called *executor* and *local logger*. The former Sub-C initiates the execution of one task and forwards the results to the next *Task Executor* components if the IoT application is developed using the dependent model. It is crystal clear that in the independent model, the results will be forwarded to the *Master* component for the aggregation or directly to the corresponding *User* component. Besides, the *local logger* sub-C records the contextual information of this task, such as its execution time.

7.2.2 Communication Protocol

Different components of the FogBus2 framework can communicate together by passing messages. Therefore, understanding the communication protocol of this framework is important. The communication protocol of FogBus2 is implemented in JSON format and messages contain eight main elements, as depicted in Fig. 7.3.

The *source* and *destination* are JSON objects containing the metadata of source and

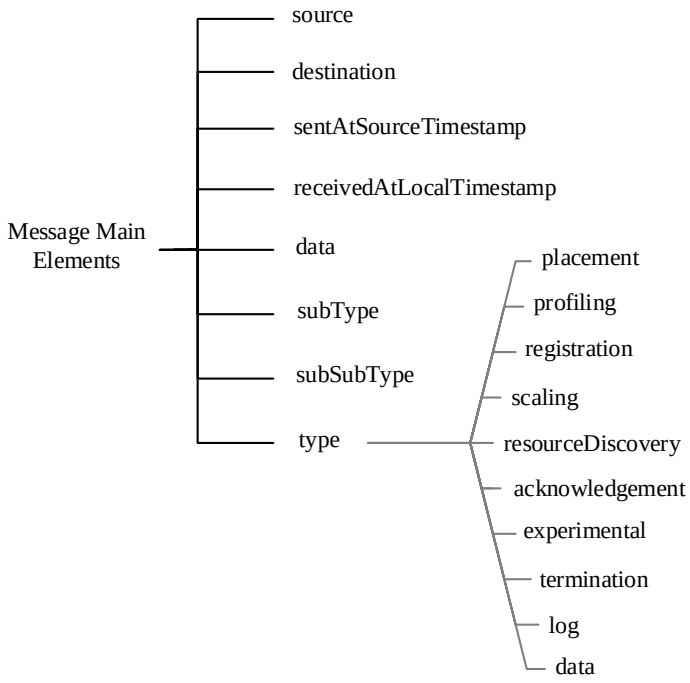


Figure 7.3: FogBus2 communication protocol format

destination of one message, respectively. The *sentAtSourceTimestamp* and *receivedAtLocalTimestamp* elements are embedded to calculate the networking delay. Furthermore, each message can carry any types of information, stored in *data*. Besides there are three other elements, namely *type*, *subType*, and *subSubType*, which are used to categorize messages. There are 10 types of messages in the current version of FogBus2 framework, shown in Fig. 7.3, where each *type* can be further divided into 41 *subType* and 5 *subSubType*. Hence, *type*, *subType*, and *subSubType* elements logically provides a hierarchical structure for the categorization of the messages. Due to the page limit we cannot describe all the messages here, however, the most important messages and their respective description are provided in Table 7.1. Also, code snippet 7.1 presents a sample message used for sending the log information ($type = log$) of server resources ($type = hostResources$) from an *Actor* component (*source role* = *Actor*) to the *Remote Logger* component (*destination role* = *RemoteLogger*). Accordingly, the message contains the resources information in the *data* element.

Table 7.1: Important communication messages

Sender	Receiver	Type	SubType	SubSubType	Description
Master	Actor	placement	runTaskExecutor	-	Master has finished the scheduling and sends this message in a no-reuse scenario
TaskExecutor	Master	placement	lookup	-	Task Executor requests the address of its children Task Executors (in the dependent model)
Master	TaskExecutor	placement	lookup	-	Master responds to the lookup message of Task Executors
TaskExecutor	Master	acknowledgement	ready	-	Task Executor has received its children's information, and use this message to acknowledge the Master that it is ready
Master	User	acknowledgement	serviceReady	-	When the service is ready and User can start sending sensory data
User	Master	data	sensoryData	-	sensoryData forwarded from the User
Master	TaskExecutor	data	intermediateData	-	Master sends sensory data to Task Executor(s) for processing
TaskExecutor	TaskExecutor	data	intermediateData	-	Task Executor finishes its execution and send intermediate data to other Task Executor(s)
TaskExecutor	Master	acknowledgement	waiting	-	Task Executor asks Master whether it can go into the cool off period
Master	TaskExecutor	acknowledgement	wait	-	Master asks Task Executor to start its cool off period immediately
Master	TaskExecutor	placement	reuse	-	Master has finished the scheduling and sends this message in reuse scenario
TaskExecutor	Master	data	finalResult	-	Task Executor sends final results to Master
Master	User	data	finalResult	-	Master sends final results to User
Master A	Master B	scaling	getProfiles	-	Master A send request to get profiles from the Master B
Master B	Master A	scaling	profilesInfo	-	Master B sends profiles to Master B
Master	Actor	scaling	initNewMaster	-	Master asks Actor to initiate a new Master
RemoteLogger	Master	log	allResourcesProfiles	-	This message is sent in response to requestProfiles message of the Master
Master A	Master B	resourcesDiscovery	requestActorsInfo	-	Master A asks Master B the information of Actors registered at Master B for further advertisement
Master B	Master A	resourcesDiscovery	actorsInfo	-	Master B sends its registered Actors' information to Master A
Master	Actor	resourcesDiscovery	advertiseMaster	-	Master advertises itself to Actor
Any Components	Any Components	resourcesDiscovery	probe	try	Any component receiving probe message should provide its component role, such as Master, Actor, etc to the sender
Any Components	Any Components	resourcesDiscovery	probe	result	The response to the probe message received from one component

```
1 {'data': {'resources': {'cpu': {'cores': 8,  // Message type is log, subtype
    is hostResources. Thus, data contains resources
2                 'frequency': 2400.0,
3                 'utilization': 0.052,
```

```
                    'utilizationPeak': 1.0},
                    'memory': {'maximum': 17179869184,
                    'utilization': 0.075,
                    'utilizationPeak': 1.0}}},
 'destination': {'addr': ['127.0.0.1', 5000],
                 'componentID': '?',
                 'hostID': 'HostID',
                 'name': 'RemoteLogger-?_127.0.0.1-5000',
                 'nameConsistent': 'RemoteLogger_HostID',
                 'nameLogPrinting': 'RemoteLogger-?_127.0.0.1-5000',
                 'role': 'RemoteLogger'},
 'receivedAtLocalTimestamp': 0.0,
 'sentAtSourceTimestamp': 1625572932123.89,
 'source': {'addr': ['127.0.0.1', 50000],
                    'componentID': '2',
                    'hostID': '127.0.0.1',
                    'name': 'Actor',
                    'nameConsistent': 'Actor_127.0.0.1',
                    'nameLogPrinting': 'Actor-2_127.0.0.1-50000_Master-?_127
    .0.0.1-5001',
                    'role': 'Actor'},
 'subSubType': '',
 'subType': 'hostResources',
 'type': 'log'}
```

Code Snippet 7.1: An Example of FogBus2 Message Format

7.2.3 Implementation of New Scheduling Technique

One of the most important challenges for resource management in Edge/Fog and CSPs is the proper scheduling of incoming IoT application requests. FogBus2 provides a straightforward mechanism for the scheduling of various types of IoT applications. In this section, we implement the ranking-based scheduling technique discussed in chapter 5 by extending the FogBus2 framework.

To integrate a new scheduling technique, a *BaseScheduler* class is provided in *containers/master/sources/utils/master/scheduler/base.py*. We inherit from *BaseScheduler* class and

override the *_schedule* function. Besides, if the utilization of the current *Master* component, which is responsible for the scheduling of IoT applications, goes beyond a threshold, the new request should be forwarded to another *Master* component. The *getBestMaster* function handles this process and can be overridden with different policies for the selection of another *Master*. Finally, scaling technique uses the *prepareScaler* function. The following steps describe how to define and integrate the new scheduling technique:

1. Navigating to *containers/master/sources/utils/master/scheduler/policies*, and create a new file named *schedulerRankingBased.py*:

```
$ pwd
/home/ubuntu/fogbus2/containers/master/sources/utils/master/scheduler/
    policies
$ > schedulerRankingBased.py
```

2. Implementing the ranking based scheduling technique based on the Algorithm 10 *schedulerRankingBased.py*. The *_schedule* function contains the logic of scheduling policy.

```
$ cat schedulerRankingBased.py

from random import randint
from time import time
from typing import List
from typing import Union

from ..base import BaseScheduler as SchedulerPolicy
from ..baseScaler.base import Scaler
from ..baseScaler.policies.scalerRandomPolicy import ScalerRandomPolicy
from ..types import Decision
from ...registry.roles.actor import Actor
from ...registry.roles.user import User
from ....types import Component


class SchedulerRankingBased(SchedulerPolicy):
    def __init__(
            self,
```

```
            isContainerMode: bool,
            *args,
            **kwargs):
        """
        :param isContainerMode: Whether this component is running in
    container
        :param args:
        :param kwargs:
        """
        super().__init__('RankingBased', isContainerMode, *args, **kwargs
    )

    def _schedule(self, *args, **kwargs) -> Decision:
        """
        :param args:
        :param kwargs:
        :return: A decision object
        """
        user: User = kwargs['user']
        allActors: List[Actor] = kwargs['allActors']
        # Get what tasks are required
        taskNameList = user.application.taskNameList

        startTime = time()
        indexSequence = ['' for _ in range(len(taskNameList))]
        indexToHostID = {}

        # Ranking of tasks belonging to an application
        rankedTasksList = self.rankApplicationTasks(
        indexSequence, **kwargs)
        indexToHostID = self.tasksAssignment(
        rankedTasksList, allActors, **kwargs)

        schedulingTime = (time() - startTime) * 1000

        # Create a decision object
        decision = Decision(
            user=user,
```

```
            indexSequence=rankedTasksList,
            indexToHostID=indexToHostID,
            schedulingTime=schedulingTime
        )
        # A simple example of cost estimation
        decision.cost = self.estimateCost(decision, **kwargs)
        return decision

    @staticmethod
    def estimateCost(decision: Decision, **kwargs) -> float:
        # You may develop your own with the following used values
        from ..estimator.estimator import Estimator
        # Get necessary params from the key args
        user = kwargs['user']
        master = kwargs['master']
        systemPerformance = kwargs['systemPerformance']
        allActors = kwargs['allActors']
        isContainerMode = kwargs['isContainerMode']
        # Init the estimator
        estimator = Estimator(
            user=user,
            master=master,
            systemPerformance=systemPerformance,
            allActors=allActors,
            isContainerMode=isContainerMode)
        indexSequence = [int(i) for i in decision.indexSequence]
        # Estimate the cost
        estimatedCost = estimator.estimateCost(indexSequence)
        return estimatedCost

    def getBestMaster(self, *args, **kwargs) -> Union[Component, None]:
        """

        :param args:
        :param kwargs:
        :return: A Master used to ask the user to request when this
    Master is busy
        """
```

```
93          user: User = kwargs['user']
94          knownMasters: List[Component] = kwargs['knownMasters']
95          mastersNum = len(knownMasters)
96          if mastersNum == 0:
97              return None
98          return knownMasters[randint(0, mastersNum - 1)]
99
100     def prepareScaler(self, *args, **kwargs) -> Scaler:
101         # Create a scaler object and return
102         scaler = ScalerRandomPolicy(*args, **kwargs)
103         return scaler
```

First, the information of *user* and all available actors *allActors* are retrieved (lines 36-37). Then, the tasks corresponding to the requested application are retrieved and stored in *taskNameList* (line 39). The *rankApplicationTasks* considers the dependency model of tasks (if any) and satisfies the dependency among tasks while defining an order for the tasks that can be executed in parallel. Different ranking policies can be defined in this function. We consider the average execution time of tasks on different servers as criteria for the ranking. Hence, among tasks that can be executed in parallel, the tasks with higher execution time receive higher priority. This eventually helps to reduce the overall response time of the application (lines 46-47). Next, *tasksAssignment* function receives the ordered *rankedTasksList* and assigns a proper actor to each task to minimize its execution time (line 48-49). According to the scheduling decision, a *decision* object will be created, storing the ordered list of the application's tasks, the list of server/host mapping, scheduling time, and the cost of scheduling, to be returned (lines 54-59). To illustrate how the execution cost of each task and overall response time of an application can be estimated, a *estimateCost* function is defined (lines 65-84). As mentioned above, the *getBestMaster* and *prepareScaler* also can be defined in *schedulerRankingBased.py*. To reduce the complexity, these functions are working based on random policy.

3. The new scheduling technique is then added to the *schedulerName* options by configuring the *initSchedulerByName* function of the *containers/master/sources/utils/master/scheduler/tools/initSchedulerByName.py*.

```
$ pwd
/home/ubuntu/fogbus2/containers/master/sources/utils/master/scheduler/
    tools
$ nano initSchedulerByName.py

def initSchedulerByName(
        .
        .
        .
    # New Added Block
    elif schedulerName == 'RankingBased':
        from ..policies.schedulerRankingBased import \
            RankingBasedPolicy
        scheduler = SchedulerRankingBased(isContainerMode=isContainerMode
    )
        return scheduler

    return None
```

4. The *Master* component can be executed using the following command while the *schedulerName* option shows the name of the selected scheduling technique:

```
$ pwd
/home/ubuntu/fogbus2/containers/master
$ docker-compose run --rm --name TempContainerName fogbus2-master  --
    containerName TempContainerName --bindIP 192.0.0.8 --schedulerName
    RankingBased
```

7.2.4 Implementation of New IoT Applications

Every containerized IoT application can be implemented and integrated with the FogBus2 framework. Alongside the implementation of new IoT applications, there are several required steps to follow in order to implement and integrate the new IoT applications with the FogBus2 framework, such as building docker images and defining dependencies between different tasks.

Overall, we implement four new IoT applications with different dependency models, namely *FaceDetection*, *FaceAndEyeDetection* and *NaiveFormulaParallelized*, and *Naive-*

FormulaSerialized. Table 7.2 presents the list of new IoT applications, their descriptions, dependency model, and the logical tasks of applications. In the application logic's tasks, only the specific tasks of each application are mentioned. The FogBus2 framework manages the rest of the steps, such as receiving input data using *User* component and communication with the database using *RemoteLogger* component. As the required steps for the integration of these applications are the same, we only describe one of these applications, called *NaiveFormulaParallelized* as it has a larger number of tasks. As Table 7.2 shows, this mathematical application contains three different tasks, called *naive_formula0*, *naive_formula1*, and *naive_formula2*, that can be executed in parallel. The corresponding equation of each task is described in Eq. 7.1.

$$\begin{cases} naive\_formula0 & a+b+c \\ naive\_formula1 & \frac{a^2}{b^2+c^2} \\ naive\_formula2 & \frac{1}{a}+\frac{2}{b}+\frac{3}{c} \end{cases} \tag{7.1}$$

To integrate this application into the FogBus2 framework, these tasks should be dockerized and prepared to be integrated as *Task Executor* components. Besides, we need a *User* component to receive inputs (using *Sensor* sub-C) and show outputs (using *Actuator* sub-C). The input will be forwarded to the *Master* component of the framework, and this component forwards inputs to corresponding *Task Executor* components based on the outcome of the scheduling algorithm. The following steps demonstrate how to implement and integrate the new application with the FogBus2 framework:

1. Create three python files as three different tasks with the desired naming convention. We name these files as *naiveFormula0.py*, *naiveFormula1.py*, and *naiveFormula2.py* which contain the logic of tasks.

```
$ pwd
/home/ubuntu/fogbus2
$ cd containers/taskExecutor/sources/utils/taskExecutor/tasks
$ > naiveFormula0.py
$ > naiveFormula1.py
$ > naiveFormula2.py
```

Table 7.2: List of applications

Application	Description	Dependency Model	Application Logic's Tasks
FaceDetection	Detecting human face from video stream, either realtime or from recorded files	Sequential	face_detection
FaceAndEyeDetection	Detecting human face and Eyes from video stream, either realtime or from recorded files	Sequential	face_detection, eye_detection
NaiveFormulaParallelized	Tasks process different parts of an equation in parallel	Parallel	naive_formula0, naive_formula1, naive_formula2
NaiveFormulaSerialized	Tasks process different parts of an equation sequentially	Sequential	naive_formula0, naive_formula1, naive_formula2

2. Edit the corresponding python files of each task and insert the required logic. For each task, a unique identifier *taskID* is required.

   (a) The logic of task *naiveFormula0.py*:

```
$ nano naiveFormula0.py
    from .base import BaseTask

    class NaiveFormula0(BaseTask):
        def __init__(self):
            super().__init__(taskID=108, taskName='NaiveFormula0')

        def exec(self, inputData):
            a = inputData['a']
            b = inputData['b']
            c = inputData['c']

            result = a + b + c
            inputData['resultPart0'] = result

            return inputData
```

   (b) The logic of task *naiveFormula1.py*:

```
$ nano naiveFormula1.py
    from .base import BaseTask
```

```
3
4     class NaiveFormula1(BaseTask):
5         def __init__(self):
6             super().__init__(taskID=109, taskName='NaiveFormula1')
7
8         def exec(self, inputData):
9             a = inputData['a']
10            b = inputData['b']
11            c = inputData['c']
12
13            result = a * a / (b * b + c * c)
14            inputData['resultPart1'] = result
15
16            return inputData
```

(c) The logic of task *naiveFormula2.py*:

```
1 $ nano naiveFormula2.py
2     from .base import BaseTask
3
4     class NaiveFormula2(BaseTask):
5         def __init__(self):
6             super().__init__(taskID=110, taskName='NaiveFormula2')
7
8         def exec(self, inputData):
9             a = inputData['a']
10            b = inputData['b']
11            c = inputData['c']
12
13            result = 1 / a + 2 / b + 3 / c
14            inputData['resultPart2'] = result
15            return inputData
```

(d) The return value of *exec* functions in the above mentioned tasks will be managed by *Task Executor*. If it is none, the return value will be ignored, otherwise, it will be forwarded to next *Task Executor* components based on the specified dependencies among tasks.

3. Configure arguments:

(a) Configure *__init__.py*:

```
$ pwd
/home/ubuntu/fogbus2/containers/taskExecutor/sources/utils/
    taskExecutor/tasks
$ nano containers/taskExecutor/sources/utils/taskExecutor/tasks/
    __init__.py

from .base import BaseTask
...
from .naiveFormula0 import NaiveFormula0
from .naiveFormula1 import NaiveFormula1
from .naiveFormula2 import NaiveFormula2
...
```

(b) Configure *initTask.py*:

```
$ pwd
/home/ubuntu/fogbus2/containers/taskExecutor/sources/utils/
    taskExecutor/tools/initTask.py
$ nano containers/taskExecutor/sources/utils/taskExecutor/tasks/
    __init__.py

from typing import Union
from ..tasks import *

def initTask(taskName: str) -> Union[BaseTask, None]:
    task = None
    if taskName == 'FaceDetection':
        task = FaceDetection()
    ...
    elif taskName == 'NaiveFormula0':
        task = NaiveFormula0()
    elif taskName == 'NaiveFormula1':
        task = NaiveFormula1()
    elif taskName == 'NaiveFormula2':
        task = NaiveFormula2()

    return task
```

4. Prepare docker images:

   (a) Prepare the required libraries:

```
$ pwd
/home/ubuntu/fogbus2/containers/taskExecutor/sources
$ cat requirements.txt

psutil
docker
python-dotenv
pytesseract
editdistance
six
```

   (b) Create dockerfiles: For each task, a docker file should be created. Considering *NaiveFormula0*:

```
$ pwd
/home/ubuntu/fogbus2/containers/taskExecutor/dockerFiles/
    NaiveFormula0

$ nano Dockerfile

# Base
FROM python:3.9-alpine3.14 as base
FROM base as builder

## Dependencies
RUN apk update
RUN apk add --no-cache \
    build-base clang clang-dev ninja cmake ffmpeg-dev \
    freetype-dev g++ jpeg-dev lcms2-dev libffi-dev \
    libgcc libxml2-dev libxslt-dev linux-headers \
    make musl musl-dev openjpeg-dev openssl-dev \
    zlib-dev curl freetype gcc6 jpeg libjpeg \
    openjpeg tesseract-ocr zlib unzip openjpeg-tools

RUN python -m pip install --retries 100 --default-timeout=600  --no-
    cache-dir --upgrade pip
```

```
RUN python -m pip install --retries 100 --default-timeout=600  numpy
     --no-cache-dir

## OpenCV Source Code
WORKDIR /workplace
RUN cd /workplace/ \
    && curl -L "https://github.com/opencv/opencv/archive/4.5.1.zip"
    -o opencv.zip \
    && curl -L "https://github.com/opencv/opencv_contrib/archive
    /4.5.1.zip" -o opencv_contrib.zip \
    && unzip opencv.zip \
    && unzip opencv_contrib.zip \
    && rm opencv.zip opencv_contrib.zip

## Configure
RUN cd /workplace/opencv-4.5.1 \
    && mkdir -p build && cd build \
    && cmake \
        -DOPENCV_EXTRA_MODULES_PATH=../../opencv_contrib-4.5.1/
    modules \
        -DBUILD_NEW_PYTHON_SUPPORT=ON \
        -DBUILD_opencv_python3=ON \
        -DHAVE_opencv_python3=ON \
        -DPYTHON_DEFAULT_EXECUTABLE=$(which python) \
        -DBUILD_TESTS=OFF \
        -DWITH_FFMPEG=ON \
        ../

## Compile

RUN cd /workplace/opencv-4.5.1/build && make -j $(nproc)
RUN cd /workplace/opencv-4.5.1/build && make install

## Python libraries
COPY ./sources/requirements.txt /install/requirements.txt
RUN python -m pip install --retries 100 --default-timeout=600  \
    --prefix=/install \
    --no-cache-dir \
```

```
    -r /install/requirements.txt

## Copy files
FROM base
COPY --from=builder /install /usr/local
COPY ./sources/ /workplace

## Install OpenCV
COPY  --from=builder /usr/local/ /usr/local/
COPY --from=builder /usr/lib/ /usr/lib/

# Hostname
RUN echo "NaiveFormula0" > /etc/hostname

# Run NaiveFormula0
WORKDIR /workplace
ENTRYPOINT ["python", "taskExecutor.py"]
```

(c) Create docker files for *NaiveFormula1* and *NaiveFormula2* similar to *NaiveFormula0*, as described in step (b).

(d) Create docker-compose files: For each task, a docker-compose file should be created. Considering *NaiveFormula0*:

```
$ pwd
/home/ubuntu/fogbus2/containers/taskExecutor/dockerFiles/
    NaiveFormula0
$ nano docker-compose.yml

version: '3'

services:

  fogbus2-naive_formula0:
    image: fogbus2-naive_formula0
    build:
      context: ../../
      dockerfile: dockerFiles/NaiveFormula0/Dockerfile
    environment:
```

```
      PUID: 1000
      PGID: 1000
      TZ: Australia/Melbourne
    network_mode:
      host
```

(e) Create docker-compose files for *NaiveFormula1* and *NaiveFormula2* similar to *NaiveFormula0*, as described in step (d).

(f) Build docker images: The docker images corresponding to the tasks of new application can be built using the provided automated script (*demo.py*).

```
$ pwd
/home/ubuntu/fogbus2/demo
$ python3.9 demo.py --buildAll
```

(g) Verify new docker images:

```
$ docker images

REPOSITORY                TAG       IMAGE ID       CREATED
    SIZE
...
fogbus2-naive_formula1    latest    5e9ad6999801   2 minutes ago
    xxx
fogbus2-naive_formula0    latest    74cfbb128699   2 minutes ago
    xxx
fogbus2-naive_formula2    latest    924d6bc0f281   3 minutes ago
    xxx
...
```

5. Prepare *User* side code:

```
$ pwd
/home/ubuntu/fogbus2/containers/user/sources/utils/user/applications
$ nano naiveFormulaParallelized.py

from time import time
from pprint import pformat
from .base import ApplicationUserSide
```

```
8  from ...component.basic import BasicComponent
9
10
11 class NaiveFormulaParallelized(ApplicationUserSide):
12
13     def __init__(
14             self,
15             videoPath: str,
16             targetHeight: int,
17             showWindow: bool,
18             basicComponent: BasicComponent):
19         super().__init__(
20             appName='NaiveFormulaParallelized',
21             videoPath=videoPath,
22             targetHeight=targetHeight,
23             showWindow=showWindow,
24             basicComponent=basicComponent)
25
26     def prepare(self):
27         pass
28
29     def _run(self):
30         self.basicComponent.debugLogger.info(
31             'Application is running: %s', self.appName)
32
33         # get user input of a, b, and c
34         print('a = ', end='')
35         a = int(input())
36         print('b = ', end='')
37         b = int(input())
38         print('c = ', end='')
39         c = int(input())
40
41         inputData = {
42             'a': a,
43             'b': b,
44             'c': c
45         }
```

```
46
47          # put it in to data uploading queue
48          self.dataToSubmit.put(inputData)
49          lastDataSentTime = time()
50          self.basicComponent.debugLogger.info(
51              'Data has sent (a, b, c): %.2f, %.2f, %.2f', a, b, c)
52
53          # wait for all the 4 results
54          while True:
55              result = self.resultForActuator.get()
56
57              responseTime = (time() - lastDataSentTime) * 1000
58              self.responseTime.update(responseTime)
59              self.responseTimeCount += 1
60
61              if 'finalResult' in result:
62                  break
63
64          for key, value in result.items():
65              result[key] = '%.4f' % value
66          self.basicComponent.debugLogger.info(
67              'Received all the 4 results: \r\n%s', pformat(result))
```

6. Define dependencies among tasks of a new application in the database. Considering MariaDB is running on 192.0.0.1 as an example:

   (a) Connect to the database:

```
1 $ mysql -h 192.0.0.1 -uroot -p
2 Enter password:
```

   (b) The *EntryTasks* contains the root tasks of this application, where the sensory data should be forwarded.

```
1 mysql> SELECT entryTasks FROM FogBus2_Applications.applications
      WHERE name='NaiveFormulaParallelized';
2
3 [
4     "NaiveFormula0",
```

```
    "NaiveFormula1",
    "NaiveFormula2"
]
```

(c) The *TaskWithDependency* contains the dependencies among tasks. For each task, we define an array of *parents* and *children*, representing predecessor and successor tasks.

```
mysql> SELECT tasksWithDependency FROM FogBus2_Applications.
    applications WHERE name='NaiveFormulaParallelized';

{
"NaiveFormula0": {
    "parents": [
        "Sensor"
    ],
    "children": [
        "Actuator"
    ]
},
"NaiveFormula1": {
    "parents": [
        "Sensor"
    ],
    "children": [
        "Actuator"
    ]
},
"NaiveFormula2": {
    "parents": [
        "Sensor"
    ],
    "children": [
        "Actuator"
    ]
}
}
```

(d) Considering the FogBus2 framework is running, the *NaiveFormulaParallelized*

can be executed using the following command:

```
$ pwd
/home/ubuntu/fogbus2/containers/user/sources

$ python user.py --bindIP 192.0.0.9 --masterIP 192.0.0.2 --
    masterPort 5001 --remoteLoggerIP 192.0.0.1 --remoteLoggerPort
    5000 --applicationName NaiveFormulaParallelized
```

7.3 Design of Computing Environment

In this section, we describe the integrated computing environment designed for this work. Overall, we consider three tiers, namely, IoT, Edge/Fog, and multi-Cloud. Fig. 7.4 presents an overview of our computing environment, consisting of heterogeneous servers.

7.3.1 IoT Tier

This tier consists of heterogeneous types of resource-limited IoT devices (such as smartphones, laptops, surveillance cameras, and any types of sensors such as ECG) that interact with the environment to collect data and perform some actions. Using the FogBus2 framework, IoT devices are able to connect and forward their requests to distributed servers in the environments. Hence, the IoT data can be processed and stored on resourceful surrogate servers, which significantly helps to reduce the processing time of data generated from IoT devices.

7.3.2 Fog Tier

In the Fog tier, we consider two clusters consisting of heterogeneous servers at the proximity of IoT devices. These servers include different Rpis (Rpi) models (such as Rpi4B, Rpi3B) and Nvidia Jetson platform (such as Jetson Nano) that are placed on-premises in the private subnet. Besides, these servers can communicate together and collaboratively handle incoming IoT requests.

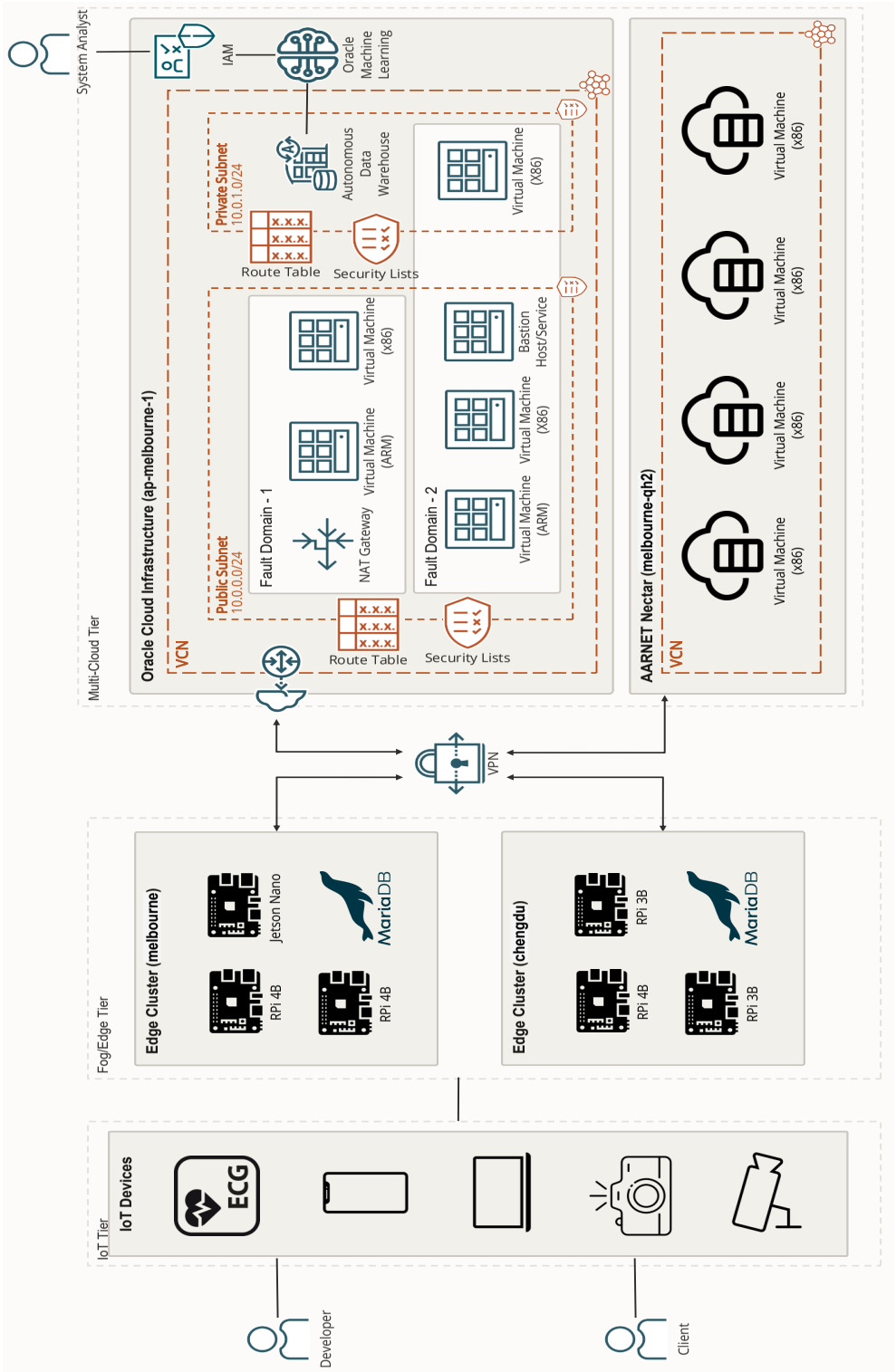


Figure 7.4: Design of computing environment

Also, The Fog tier contains local databases (e.g., MariaDB), to persist the required local information and for faster access to required data for resource management. The local data can also be transferred periodically to the centralized database which is placed in the Cloud.

7.3.3 Multi-Cloud Tier

The computing and storage resources of IoT devices can be expanded by supporting CSPs in different geo-location areas, bringing location independency for IoT applications. In this chapter, we design a multi-Cloud environment by exploiting the resources from Oracle Cloud Infrastructure (OCI) and Nectar Cloud Infrastructure (NCI) [2].

In the design of the multi-Cloud environment, the following design criteria are considered.

- **Multiple Virtual Cloud Networks (VCN):** Two different VCNs are considered for OCI and NCI, as depicted in Fig. 7.4. The resources and services of each Cloud service provider are defined within the respective VCN.

- **Public and Private subnets:** Within each VCN, the resources can be categorized and created in multiple public and private subnets. A public subnet has an outbound route that sends all traffic through **Internet Gateway**. The resources in the public subnets can also receive inbound traffic through Internet Gateway. The resources in the private subnet do not have direct access to the Internet, and they require **Network Address Translation (NAT) Gateway** to forward data to the Internet. Besides, the resources within each subnet can be protected using a set of **Security Groups/Lists/Rules** that act as a virtual firewall to control the inbound and outbound traffic.

- **Bastion Host/Service:** To further increase the security level and protection mechanism of resources in the private subnets, the bastion Host/Service can be used as a controlled entry point to access the resources. The topology has a single, known entry point that you can monitor and audit regularly. So, you can avoid exposing

[2]https://nectar.org.au/

the more sensitive components of the topology without compromising access to them.

- **Heterogeneous Virtual Machines (VMs):** In the computing environment, we have used computing instances with different architectures (i.e., x86 and ARM) in OCI and NCI.
- **Fault Domain (FD):** To increase the availability of the framework, we use several FDs when deploying the resources, so that in case of any failure in one FD, the framework can continue working without any issues.
- **Database:** Alongside the local databases at the Fog tier, we use a centralized database in the Cloud to store all the detailed data of the IoT devices, users, framework, and resources. The local databases also periodically forward their batch of data to the database. Hence, for the long-term analysis of the system, the centralized database can be used to find anomalies, data trends, and behavior of users and resources using different analysis techniques and tools.

7.4 Evaluation and Validation

To evaluate the performance of the extended FogBus2 framework, an integrated computing environment consisting of multiple Cloud instances and Fog servers is prepared. Table 7.3 depicts the full configuration of servers and corresponding running components. All the tests are performed using the new scheduling technique and new IoT applications, described in the previous sections.

Fig. 7.5 represents the average docker size of components in compressed and uncompressed formats. The compressed docker image size is obtained from the average size of docker images stored in the docker hub for multiple architectures, while uncompressed docker image size is obtained from the average size of extracted docker images on different instances. The size of the compressed docker image shows that extended FogBus2 components are lightweight to be downloaded on different platforms, ranging from a few megabytes to roughly 100 MB at maximum. Besides, the uncompressed docker image size proves that extended FogBus2 components are not resource-hungry

Table 7.3: Configuration of resources

Server Tag	Server Name	Computing Layer	Public IP Address	Private IP Address	Port	Component Role	Environment Preparation
A	Oracle1	Cloud	168.138.9.91	192.0.0.1	5000	RemoteLogger, Actor_1	docker and docker-compose
B	Oracle2	Cloud	168.138.10.94	192.0.0.2	automatically assign	Actor_2	docker and docker-compose
C	Oracle3	Cloud	168.138.15.110	192.0.0.3	automatically assign	Actor_3	docker and docker-compose
D	Oracle4	Cloud	168.138.15.111	192.0.0.4	automatically assign	Actor_4	docker and docker-compose
E	Oracle5	Cloud	168.138.15.118	192.0.0.5	automatically assign	Actor_5	docker and docker-compose
F	Nectar1	Cloud	45.113.235.222	192.0.0.6	automatically assign	Actor_6	docker and docker-compose
G	Nectar2	Cloud	45.113.232.187	192.0.0.7	automatically assign	Actor_7	docker and docker-compose
H	Nectar3	Cloud	45.113.232.245	192.0.0.8	automatically assign	Actor_8	docker and docker-compose
I	Rpi 4B 4GB	Fog	-	192.0.0.9	5000	RemoteLogger, Actor_9	docker and docker-compose
J	Rpi 4B 4GB	Fog	-	192.0.0.10	automatically assign	Actor_10	docker and docker-compose
K	Rpi 3B 1GB	Fog	-	192.0.0.11	automatically assign	Actor_11	docker and docker-compose
L	Jetson Nano 4GB	Fog	-	192.0.0.12	5001	Master	docker and docker-compose
M	VM on a Laptop	IoT	-	192.0.0.13	automatically assign	User	Python3.9

and do not occupy the storage. The reason why the image sizes of *User* and *Task Executor* components are not provided is that the docker image sizes of these components heavily depend on the logic of IoT applications.

Fig. 7.6 represents the average runtime RAM usage of the extended FogBus2 components on servers with different architectures. It illustrates that the average resource usage of the FogBus2 components on different architectures is low, ranging from 25 MB to 45 MB.

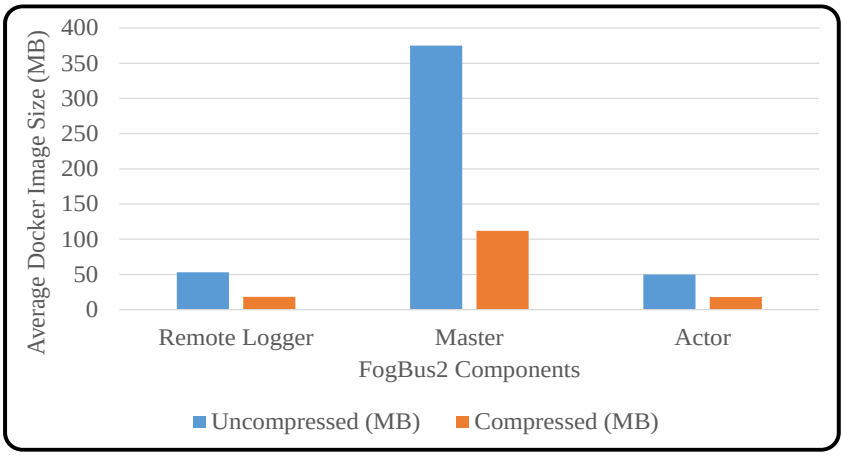


Figure 7.5: Average docker image size of FogBus2 components

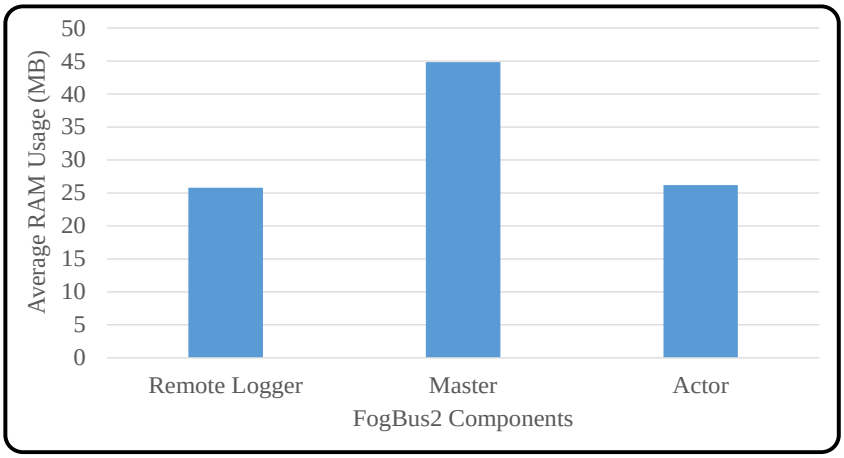


Figure 7.6: Average runtime RAM usage of FogBus2 components

Fig. 7.7 demonstrates the average startup time of FogBus2 components on different architectures (i.e., x86 and different ARM architectures). It contains the amount of time required to start containers until they become in a completely functional state for serving incoming requests. Therefore, the extended FogBus2 framework only requires a few seconds to enter into its fully functional state. It significantly helps IoT developers in the development and testing phase as they require to re-initiate the framework several times to test and debug their applications.

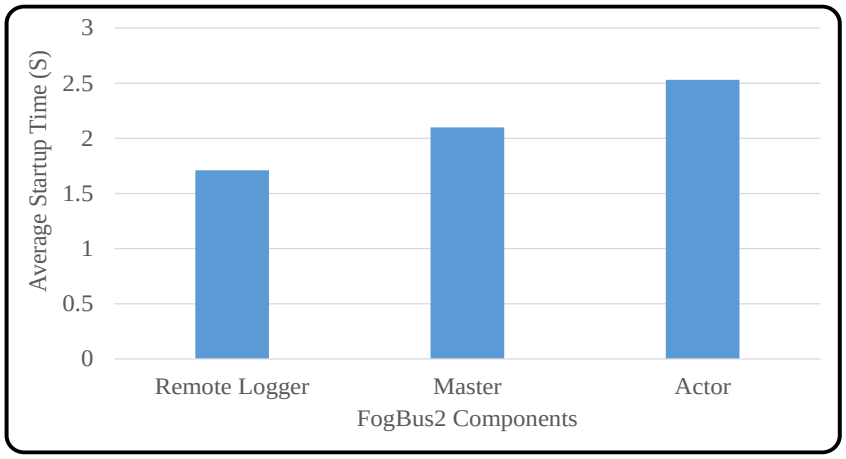


Figure 7.7: Average startup time of FogBus2 components

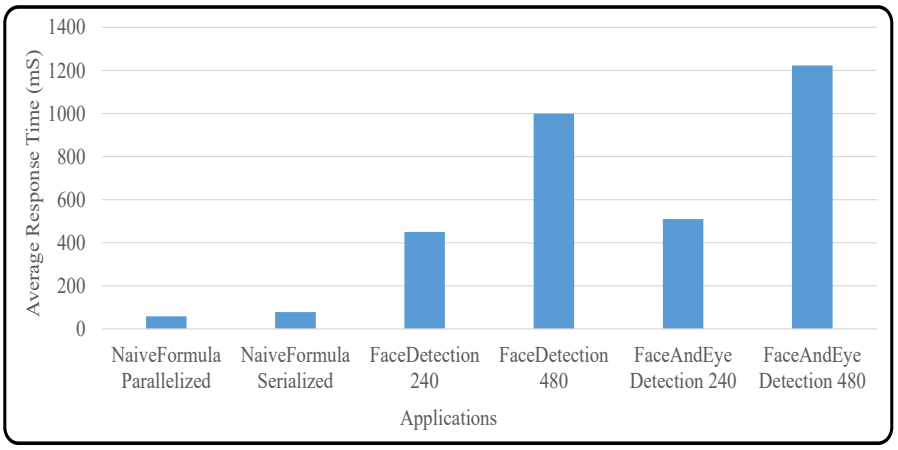


Figure 7.8: Average response time of IoT applications

Fig. 7.8 depicts the average response time of *NaiveFormulaParallelized*, *NaiveFormulaSerialized*, *FaceAndEyeDetection* and *FaceDetection application* with different resolutions based on the server configuration derived from ranking-based scheduling technique. It also validates the correct implementation and integration of new scheduling technique and IoT applications with the FogBus2 framework.

7.5 Summary

In this chapter, key components of the FogBus2 framework and its communication protocol are described. Besides, we extended this framework with a new scheduling technique for real-time IoT applications. Also, we implemented and integrated new IoT applications with this framework. Next, an integrated computing environment, containing multiple Cloud service providers and Fog devices is designed. Finally, we studied the performance of the extended FogBus2 framework in the designed computing environment.

Chapter 8
Conclusions and Future Directions

This chapter concludes the thesis and describes a summary of works and key contributions. Next, it identifies and discusses several future research directions for further improvement of Fog computing concepts.

8.1 Summary of Contributions

The Internet of Things (IoT) paradigm has become an integral part of our daily life, thanks to the continuous advancements of hardware and software technologies and ubiquitous access to the Internet. IoT spans across various application scenarios with heterogeneous resource requirements, ranging from computation-intensive to latency-sensitive. Fog computing has been emerged as a distributed computing paradigm, containing not only servers at the proximity of IoT devices but also distant remote servers. Hence, it provides heterogeneous resources to support a wide variety of applications. It has already drawn significant attention from both industry and academia. However, the number of resources is limited compared to the ever-increasing demand for numerous IoT devices. Also, the execution of one IoT application on some resources may affect the execution of other applications and degrade users' QoE and overall system performance. Hence, the smooth execution of different IoT applications in this highly heterogeneous and dynamic environment is not quite simple. In Fog computing environments, these issues can be addressed by identifying appropriate scheduling techniques for IoT applications. In this thesis, we investigated scheduling techniques of IoT applications in highly distributed, heterogeneous, and dynamic Fog computing environments.

Chapter 1 presented the basic concepts of Edge and Fog computing and discussed

their main difference from our perspective. Next, important challenges of Fog computing environments are identified and described. This chapter also presented the research questions addressed in this thesis.

Chapter 2 investigated the existing scheduling techniques for IoT applications in Fog computing from different perspectives, namely IoT application structure, environmental architecture, optimization characteristics, decision engines' characteristics, and performance evaluation. Then, considering each perspective, a taxonomy and survey of the recent literature are provided. Finally, the research gaps of each perspective are described.

Chapter 3 investigated efficient distributed scheduling policies for network resources to optimize the total throughput of the network while mitigating the interference in dense and ultra-dense Edge and Fog computing environments. Firstly, the throughput model of hierarchical Edge and Fog computing environments is formulated. Secondly, a distributed dynamic clustering algorithm is proposed to solve the intra-cluster interference. Afterward, the inter-cluster interference is modeled using a Fog-driven graph formation strategy. Then, a graph coloring technique is proposed to distribute network resources while decreasing inter-cluster interference. Finally, a policy-aware scheduling technique is proposed to distribute network resources to Edge servers.

Chapter 4 Puts forward a distributed batch placement scheduling technique for concurrent DAG-based IoT applications to optimize the execution cost of IoT applications. At first, the weighted cost of running concurrent IoT applications in terms of the execution time of IoT applications and the energy consumption of IoT devices is formulated. Then, different tasks of concurrent IoT applications are prepared as several batches for the scheduling while the dependency of tasks within each application is considered. Afterward, an optimized meta-heuristic technique based on the Memetic Algorithm is proposed for the batch scheduling to solve the minimization problem. Also, to solve the stochastic failures, a fast failure recovery mechanism is embedded in the scheduling technique to assign failed tasks to appropriate servers in a timely manner

Chapter 5 proposed distributed techniques for scheduling and migration management of real-time IoT applications in hierarchical Edge and Fog computing environments. Firstly, mathematical models to minimize the weighted cost of running and mi-

gration of real-time applications in terms of the execution time of IoT applications and the energy consumption of IoT devices in hierarchical Edge and Fog computing environments are proposed. Next, a fast ranking-based distributed scheduling policy for heterogeneous IoT applications is proposed to reduce the weighted cost. Afterward, to address the mobility of users, a distributed migration management technique is proposed to minimize downtime and service interruption in the pre-copy migration model. Also, failure recovery techniques for clustering of resources, scheduling, and migration of IoT applications are embedded to solve random failures.

Chapter 6 presented a distributed DRL-based scheduling framework to learn and optimize complex scheduling of DAG-based IoT applications in highly dynamic Edge and Fog computing environments. At first, the weighted cost of running IoT applications in terms of the execution time of IoT applications and the energy consumption of IoT devices is formulated as a minimization problem. Next, a DRL model for the scheduling of DAG-based IoT applications is proposed. Then, action, reward, and state management methods for our DRL framework are defined. Afterward, a distributed actor-critic DRL-based model for off-policy learning of optimal policy for Edge and Fog computing environments is put forward to improve exploration costs and convergence rate. Finally, the proposed technique is evaluated and validated using simulation and testbed experiments, consisting of heterogeneous Fog and Cloud servers.

Chapter 7 investigated a software system for scheduling IoT applications in Edge and Fog computing environments. Firstly, the system configuration, consisting of multiple Cloud datacenters, multiple Edge/Fog servers, and different IoT applications, was described. Next, the important modules of our software system for scheduling have been described. Afterward, we implemented and integrated several containerized DAG-based IoT applications, such as face and eye detection. Then, the implementation and integration of the scheduling algorithm in this environment were presented. Finally, evaluation and validation of scheduling algorithm's performance in real Fog computing environments were described.

The chapters mentioned above collectively present multiple scheduling techniques in highly heterogeneous Edge and Fog computing environments, which is a timely contribution to the state-of-the-art.

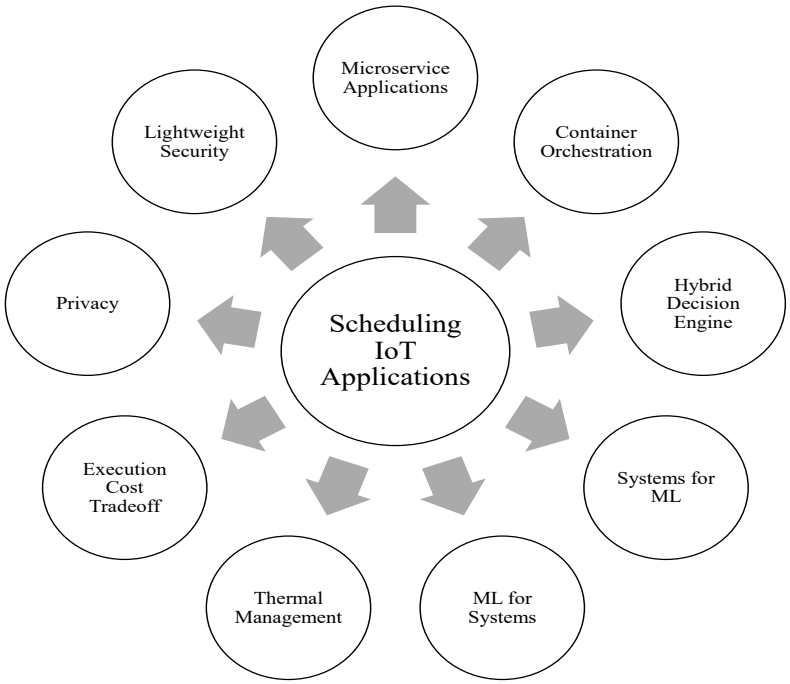


Figure 8.1: Summary of future directions

8.2 Future Research Directions

This thesis addressed several challenges of scheduling IoT applications in Edge and Fog computing environments. However, Edge and Fog computing paradigms can be further improved by addressing several key issues requiring further investigation. An overview of future directions discussed in this section is presented in Fig. 8.1 and described in the following.

8.2.1 Microservices-based applications

The popularity of microservices for the deployment of IoT applications is due to their loosely-coupled design, modularity, and the capability of microservices to be shared among multiple IoT applications. But, it may incur data consistency and data privacy challenges. To overcome these challenges, the placement techniques should consider the context of applications and data before sharing microservices. Also, the loosely-coupled and lightweight design of microservices enables the efficient migration of microservice-

based applications. Hence, considering the type and structure of IoT applications ranging from strictly latency-sensitive to strictly computation-intensive, different migration models (e.g., pre-copy, post-copy, hybrid) should be completely investigated in real Edge and Fog computing environments for a smooth migration.

8.2.2 Practical container orchestration in Fog computing

Orchestrating container-based IoT applications is well studied in the cloud computing paradigm. However, in Fog computing, in which CSs and FSs collaborate to run an application, several deployment models of orchestration techniques are available. To illustrate, the master node can either be deployed on a FS or CS. When the master node runs on a FS, the communication overhead and latency for end-users will be reduced. However, the master node will use the most of resources on the FS for the cluster management, especially for resource-limited FSs. Also, when the master runs on a CS, the startup time and application latency will be negatively affected. Thus, based on the application structure and its goal, different container orchestration models should be studied to find the best deployment model according the application scenario. Several practical studies can be conducted to find which deployment model is suitable for each IoT application scenario in terms of communication overhead, the startup time of services, memory footprint, failure management, load balancing, and scheduling.

8.2.3 Hybrid scheduling decision engines

Usually, decision engines only use one placement technique for different IoT applications. However, the requirements of IoT applications are heterogeneous, where one application is sensitive to startup time while the extremely high accuracy is not important, or vice versa. Besides, decision engines should be adapted to work with either single or batch placement approaches. Hence, context-aware decision engines with a suite of placement techniques can be implemented to address the requirements of different IoT applications. Moreover, current scheduling techniques solely use heuristic, metaheuristic, or ML-based algorithms for making the decision. However, these algorithms can be integrated for the efficient scheduling of applications. To illustrate, the training

of DRL models is costly and time-consuming, while after training the inference time is very low. Hence, heuristics or meta-heuristics algorithms can be used in the training phase of DRL algorithms for more efficient training of DRL models.

8.2.4 Systems for ML

Due to advancements in ML techniques and their rapid adoptions across many IoT applications, it creates new demand for specialized hardware resources and software frameworks (e.g., Nvidia GPU-powered Jetson, Google Coral Edge Tensor Processing Unit (Edge TPU)) for Fog computing. New systems and software frameworks should be built to support the massive computational requirement of these AI workloads. Besides, these systems can be a potential target for the deployment of decision engines due to their high computational capacity.

8.2.5 ML for systems

While ML systems themselves are becoming mature and adopted into many critical application domains, it is equally important to use these ML techniques to design and operate large-scale systems. Adopting the ML techniques to solve different resource management problems in Edge/Fog and Cloud is crucial to managing these complex infrastructures and workloads. Moreover, majority of ML techniques are not optimized to run on resource-constrained devices. To illustrate, consider an efficient ML model trained for resource management. Many resource-constrained devices require full integer quantization to run the trained model. However, post quantization of trained models is not always possible and in some cases they cannot be efficiently converted. As a result, a study on requirements for the efficient execution of resource management ML models on resource-limited FSs should be conducted.

8.2.6 Thermal management

The temperature of FSs (e.g., racks of Rpi or Nvidia Jetson platform), especially those executing large workloads, increases significantly. So, the cooling systems should be em-

bedded to avoid system breakdown. Hence, a study on the temperature of these devices based on their main processing and communication modules can be conducted to find the respective temperature dynamics in different application scenarios and workloads. Moreover, lightweight thermal management software systems for FSs can be designed to control the temperature dynamics of devices. Also, the thermal index can be added as an important optimization/decision parameter alongside other currently available parameters (e.g., time, energy, cost) for the placement techniques.

8.2.7 Execution cost trade-off

The goal of scheduling algorithms is to minimize the execution cost of applications either from IoT or resource providers' perspectives. However, some parameters such as energy consumption or carbon footprint should be considered from both perspectives. Hence, not only is minimizing these parameters from either perspective critical to reducing total energy consumption, but a trade-off parameter between the execution cost of IoT devices and resource providers can be designed, aiming at total energy or carbon footprint minimization.

8.2.8 Privacy aware and adaptive decision engines

Data-driven and distributed scheduling approaches are gaining popularity due to their high adaptability and scalability. However, sharing raw data of users or systems incurs privacy issues. To illustrate, in DDRL-based scheduling techniques, sharing experiences of multiple agents significantly reduce the exploration costs and improve convergence time of DDRL agents while incurring privacy concerns when raw agents' experiences are shared. Accordingly, privacy-aware mechanisms for sharing such data (e.g., agents' experiences) can be integrated with these highly adaptive distributed scheduling techniques.

8.2.9 Lightweight security mechanisms

Due to the highly distributed nature of Edge and Fog computing paradigms, FSs are situated at different geo-locations. Besides, FSs are highly exposed to users. Hence, the process of protecting FSs under the global security umbrella is challenging. Moreover, security mechanisms usually add overhead either for initiating IoT-enabled systems in Fog computing or the execution of applications in these environments. Hence, lightweight and distributed security mechanisms should be designed for Edge and Fog computing environments. Blockchain technology, due to its distributed approach, is recently regarded as one of the main security enablers for Edge and Fog computing paradigms. However, it has considerable computation overhead for FSs, especially the resource-constrained ones. Hence, different approaches for the deployment of Blockchain technology in Edge and Fog computing environments should be studied. Besides, developing a lightweight Blockchain technique for these environments is required.

8.2.10 Single-Sign-On mechanism

Single-Sign-On (SSO) is a property of access control of multiple related but independent software systems. It offers users the ability to securely access and use a variety of distributed resources without the need for multiple usernames/passwords or authentication challenges/responses. In the Grid computing world, this is typically obtained using the trust of the certification authority that issued the certificate and local policy on whether that individual with that certificate is allowed access. In many Grid computing environments, this can be achieved by mapping of the distinguished name associated with the certificate to a local system account. Grid Computing has many overlaps with Edge and Fog computing environments. Hence, different mechanisms such as SSO, which is not currently available for Edge, Fog, and Cloud computing environments, can be implemented in these novel large-scale distributed systems.

8.2.11 Software Systems for Resource Management

Resource management software systems help the smooth execution of IoT applications in a real-world environment. We have developed the FogBus2 framework as a distributed and container-based software system for resource management in large-scale distributed systems. Currently, the communication module of FogBus2 works based on TCP sockets. However, other messaging platforms and libraries such as Apache Kafka, ZeroMQ, and RabbitMQ can be integrated with the communication module of this framework to support diverse communication methods for different IoT applications. Besides, FogBus2 can be further extended to act as a broker for other commercial technologies and services for linking IoT and computing resources, such as AWS Lambda functions (as a serverless service) and Amazon IoT Greengrass (which enables local execution of AWS Lambda functions, Docker containers, native OS processes, or custom runtimes), just to mention a few.

8.3 Final Remarks

The Fog computing paradigm has become the backbone of today's digital world, enabling IoT-driven solutions to be deployed for different use cases such as healthcare, transportation, science, and entertainment, just to mention a few. To fully utilize the potential of Fog computing, efficient and dynamic scheduling of IoT applications is a major concern, affecting the execution cost of IoT applications, users' QoE, and operational costs. In this thesis, we investigated how to efficiently schedule networking and computing resources for the smooth execution of heterogeneous IoT applications over resource-constrained distributed FSs and resourceful CSs. The algorithms, mathematical models, and system architectures proposed in this thesis optimize the execution time of IoT applications, the service start time of IoT applications, the energy consumption of IoT devices, the overhead of scheduling techniques, and enhance users' QoE of users. Research on scheduling IoT applications, such as presented in this thesis, will enable Edge and Fog service providers to successfully and efficiently perform scheduling in highly heterogeneous, dynamic, and complex Edge and Fog computing environments

at scale. Moreover, these research outcomes can further advance innovations and developments of IoT, Edge, and Fog computing systems.

Bibliography

[1] Q. Deng, M. Goudarzi, and R. Buyya, "FogBus2: a lightweight and distributed container-based framework for integration of IoT-enabled systems with Edge and Cloud computing," in Proceedings of the International Workshop on Big Data in Emergent Distributed Environments (BiDEDE'21) in conjunction with the 2021 ACM SIGMOD/PODS Conference. ACM, 2021, pp. 1–8.

[2] F. Khodadadi, A. Dastjerdi, and R. Buyya, "Internet of Things: an overview," in Internet of Things, R. Buyya and A. Vahid Dastjerdi, Eds. Morgan Kaufmann, 2016, pp. 3–27.

[3] M. Goudarzi, H. Wu, M. Palaniswami, and R. Buyya, "An application placement technique for concurrent IoT applications in Edge and Fog computing environments," IEEE Transactions on Mobile Computing, vol. 20, no. 4, pp. 1298–1311, 2021.

[4] R. Ranjan, O. Rana, S. Nepal, M. Yousif, P. James, Z. Wen, S. Barr, P. Watson, P. P. Jayaraman, and D. Georgakopoulos, "The next grand challenges: Integrating the Internet of Things and data science," IEEE Cloud Computing, vol. 5, no. 3, pp. 12–26, 2018.

[5] "Cisco annual internet report (2018–2023) white paper," https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, Mar. 2020, [Online; accessed 20-October-2021].

[6] Norton, "The future of IoT: 10 predictions about the Internet of Things," https://us.norton.com/internetsecurity-iot-5-predictions-for-the-future-of-iot.html, 2019, [Online; accessed 20-October-2021].

[7] "The Internet of Things 2020," https://www.businessinsider.com/internet-of-things-report, Mar. Business Insider. 2020, [Online; accessed 20-October-2021].

[8] "Bain & company," https://www.bain.com/about/media-center/

press-releases/2018/bain-predicts-the-iot-market-will-more-than-double-by-2021/, [Online; accessed 20-October-2021].

[9] "Statista: Internet of Things spending worldwide 2023," https://www.statista.com/statistics/668996/worldwide-expenditures-for-the-internet-of-things/, [Online; accessed 20-October-2021].

[10] "IoT growth demands rethink of long term storage strategies," https://www.idc.com/getdoc.jsp?containerId=prAP46737220, IDC. 2020, [Online; accessed 20-October-2021].

[11] M. Goudarzi, Z. Movahedi, and M. Nazari, "Mobile Cloud computing: a multisite computation offloading," in Proceedings of the 8th International Symposium on Telecommunications (IST). IEEE, 2016, pp. 660–665.

[12] E. G. Petrakis, S. Sotiriadis, T. Soultanopoulos, P. T. Renta, R. Buyya, and N. Bessis, "Internet of Things as a service (itaas): Challenges and solutions for management of sensor data on the Cloud and the Fog," Internet of Things, vol. 3, pp. 156–174, 2018.

[13] H. Madsen, B. Burtschy, G. Albeanu, and F. Popentiu-Vladicescu, "Reliability in the utility computing era: Towards reliable Fog computing," in Proceedings of the 20th International Conference on Systems, Signals and Image Processing (IWSSIP). IEEE, 2013, pp. 43–46.

[14] J. Yao and N. Ansari, "QoS-aware Fog resource provisioning and mobile device power control in IoT networks," IEEE Transactions on Network and Service Management, vol. 16, no. 1, pp. 167–175, 2018.

[15] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile Edge computing: The communication perspective," IEEE Communications Surveys & Tutorials, vol. 19, no. 4, pp. 2322–2358, 2017.

[16] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, "Fog computing for the Internet of Things: A survey," ACM Transactions on Internet Technology (TOIT), vol. 19, no. 2, pp. 1–41, 2019.

[17] IoT For All, "The Big Three Make a Play for the Fog," https://www.iotforall.com/big-three-make-play-fog/, 2018, [Online; accessed 20-October-2021].

[18] Yifat Perry, "Azure Hybrid Cloud: Azure in Your Local Data Center," https://cloud.netapp.com/blog/azure-cvo-blg-azure-hybrid-cloud-in-your-data-center, 2018, [Online; accessed 20-October-2021].

[19] Industrial Internet Consortium, "The industrial internet consortium and OpenFog consortium join forces," https://www.iiconsortium.org/press-room/01-31-19.htm, 2019, [Online; accessed 20-October-2021].

[20] Cisco, "IOx and Fog Applications," https://www.cisco.com/c/en_in/solutions/internet-of-things/iot-fog-applications.html, 2016, [Online; accessed 20-October-2021].

[21] SBWire, "Fog Computing Market to Grow at 65% CAGR Till 2024 : Cisco, Dell, Nebbiolo, IBM, Intel, Microsoft and 15 Other Companies Profile," http://www.digitaljournal.com/pr/3997363, 2018, [Online; accessed 20-October-2021].

[22] Nvidia Jetson, "Embedded Systems with Jetson," https://www.nvidia.com/en-au/autonomous-machines/embedded-systems/, 2021, [Online; accessed 20-October-2021].

[23] OpenFog Consortium, "OpenFog reference architecture for Fog computing," OpenFog Reference Architecture for Fog Computing (OPFRA001), vol. 20817, p. 162, 2017.

[24] Ericsson, "Ericsson Mobility Report June 2018," https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-june-2018.pdf, 2018, [Online; accessed 20-October-2021].

[25] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," arXiv preprint arXiv:1606.01540, 2016.

[26] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments," Software: Practice and Experience, vol. 47, no. 9, pp. 1275–1296, 2017.

[27] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in Edge and Fog computing environments," arXiv preprint arXiv:2109.05636, 2021.

[28] Y. L. Lee, T. C. Chuah, J. Loo, and A. Vinel, "Recent advances in radio resource management for heterogeneous LTE/LTE-A networks," IEEE Communications Surveys & Tutorials, vol. 16, no. 4, pp. 2142–2180, 2014.

[29] M. Goudarzi, M. Zamani, and A. T. Haghighat, "A fast hybrid multi-site computation offloading for mobile Cloud computing," Journal of Network and Computer Applications, vol. 80, pp. 219–231, 2017.

[30] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Application management in Fog computing environments: A taxonomy, review and future directions," ACM Computing Surveys (CSUR), vol. 53, no. 4, pp. 1–43, 2020.

[31] C. Fiandrino, N. Allio, D. Kliazovich, P. Giaccone, and P. Bouvry, "Profiling performance of application partitioning for wearable devices in mobile Cloud and Fog computing," IEEE Access, vol. 7, pp. 12 156–12 166, 2019.

[32] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based IoT application placement within heterogeneous and resource constrained Fog computing environments," in Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, 2019, pp. 71–81.

[33] V. Karagiannis and S. Schulte, "Comparison of alternative architectures in Fog computing," in Proceedings of the 4th IEEE International Conference on Fog and Edge Computing (ICFEC). IEEE, 2020, pp. 19–28.

[34] M. Goudarzi, M. Palaniswami, and R. Buyya, "A distributed application placement and migration management techniques for Edge and Fog computing en-

vironments," in Proceedings of the 16th Conference on Computer Science and Intelligence Systems (FedCSIS). IEEE, 2021, pp. 37–56.

[35] H. Lin, S. Zeadally, Z. Chen, H. Labiod, and L. Wang, "A survey on computation offloading modeling for Edge computing," Journal of Network and Computer Applications, p. 102781, 2020.

[36] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Latency-aware application module management for Fog computing environments," ACM Transactions on Internet Technology (TOIT), vol. 19, no. 1, pp. 1–21, 2018.

[37] M. Hu, L. Zhuang, D. Wu, Y. Zhou, X. Chen, and L. Xiao, "Learning driven computation offloading for asymmetrically informed Edge computing," IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 8, pp. 1802–1815, 2019.

[38] M. Abdel-Basset, R. Mohamed, M. Elhoseny, A. K. Bashir, A. Jolfaei, and N. Kumar, "Energy-aware marine predators algorithm for task scheduling in IoT-based Fog computing applications," IEEE Transactions on Industrial Informatics, vol. 17, no. 7, pp. 5068–5076, 2020.

[39] W. Dou, W. Tang, B. Liu, X. Xu, and Q. Ni, "Blockchain-based mobility-aware offloading mechanism for Fog computing services," Computer Communications, vol. 164, pp. 261–273, 2020.

[40] K. Verma, A. Kumar, M. S. U. Islam, T. Kanwar, and M. Bhushan, "Rank based mobility-aware scheduling in Fog computing," Informatics in Medicine Unlocked, p. 100619, 2021.

[41] C. Anglano, M. Canonico, P. Castagno, M. Guazzone, and M. Sereno, "Profit-aware coalition formation in Fog computing providers: A game-theoretic approach," Concurrency and Computation: Practice and Experience, vol. 32, no. 21, p. e5220, 2020.

[42] S. N. Shirazi, A. Gouglidis, A. Farshad, and D. Hutchison, "The extended Cloud: Review and analysis of mobile Edge computing and Fog from a security and re-

silience perspective," IEEE Journal on Selected Areas in Communications, vol. 35, no. 11, pp. 2586–2595, 2017.

[43] R. Roman, J. Lopez, and M. Mambo, "Mobile Edge computing, Fog et al.: A survey and analysis of security threats and challenges," Future Generation Computer Systems, vol. 78, pp. 680–698, 2018.

[44] K. Tange, M. De Donno, X. Fafoutis, and N. Dragoni, "A systematic survey of industrial Internet of Things security: Requirements and Fog computing opportunities," IEEE Communications Surveys & Tutorials, vol. 22, no. 4, pp. 2489–2520, 2020.

[45] P. Zhang, M. Zhou, and G. Fortino, "Security and trust issues in Fog computing: A survey," Future Generation Computer Systems, vol. 88, pp. 16–27, 2018.

[46] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec, and A. V. Vasilakos, "Fog computing for sustainable smart cities: A survey," ACM Computing Surveys (CSUR), vol. 50, no. 3, pp. 1–43, 2017.

[47] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K.-K. R. Choo, and M. Dlodlo, "From Cloud to Fog computing: A review and a conceptual live VM migration framework," IEEE Access, vol. 5, pp. 8284–8300, 2017.

[48] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of Edge computing and deep learning: A comprehensive survey," IEEE Communications Surveys & Tutorials, vol. 22, no. 2, pp. 869–904, 2020.

[49] F. A. Kraemer, A. E. Braten, N. Tamkittikhun, and D. Palma, "Fog computing in healthcare–a review and discussion," IEEE Access, vol. 5, pp. 9206–9222, 2017.

[50] E. Baccarelli, P. G. V. Naranjo, M. Scarpiniti, M. Shojafar, and J. H. Abawajy, "Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study," IEEE Access, vol. 5, pp. 9882–9910, 2017.

[51] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on Fog computing: State-of-the-art and research

challenges," IEEE communications surveys & tutorials, vol. 20, no. 1, pp. 416–464, 2017.

[52] P. Hu, S. Dhelim, H. Ning, and T. Qiu, "Survey on Fog computing: architecture, key technologies, applications and open issues," Journal of Network and Computer Applications, vol. 98, pp. 27–42, 2017.

[53] M. Mukherjee, L. Shu, and D. Wang, "Survey of Fog computing: Fundamental, network applications, and research challenges," IEEE Communications Surveys & Tutorials, vol. 20, no. 3, pp. 1826–1857, 2018.

[54] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," IEEE Access, vol. 6, pp. 47 980–48 009, 2018.

[55] S. B. Nath, H. Gupta, S. Chakraborty, and S. K. Ghosh, "A survey of Fog computing and communication: current researches and future directions," arXiv preprint arXiv:1804.04365, 2018.

[56] P. Habibi, M. Farhoudi, S. Kazemian, S. Khorsandi, and A. Leon-Garcia, "Fog computing: a comprehensive architectural survey," IEEE Access, vol. 8, pp. 69 105–69 133, 2020.

[57] J. Singh, P. Singh, and S. S. Gill, "Fog computing: A taxonomy, systematic review, current trends and research challenges," Journal of Parallel and Distributed Computing, 2021.

[58] M. Aazam, S. Zeadally, and K. A. Harras, "Offloading in Fog computing for IoT: Review, enabling technologies, and research opportunities," Future Generation Computer Systems, vol. 87, pp. 278–289, 2018.

[59] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about Fog computing and related Edge computing paradigms: A complete survey," Journal of Systems Architecture, vol. 98, pp. 289–330, 2019.

[60] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in Fog and Edge computing," ACM Computing Surveys (CSUR), vol. 53, no. 3, pp. 1–35, 2020.

[61] I. Martinez, A. S. Hafid, and A. Jarray, "Design, resource management, and evaluation of Fog computing systems: A survey," IEEE Internet of Things Journal, vol. 8, no. 4, pp. 2494–2516, 2020.

[62] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile Edge computing: A machine learning-based perspective," Computer Networks, p. 107496, 2020.

[63] M. Ghobaei-Arani, A. Souri, and A. A. Rahmanian, "Resource management approaches in Fog computing: a comprehensive review," Journal of Grid Computing, vol. 18, no. 1, pp. 1–42, 2020.

[64] M. S. U. Islam, A. Kumar, and Y.-C. Hu, "Context-aware scheduling in Fog computing: A survey, taxonomy, challenges and future directions," Journal of Network and Computer Applications, p. 103008, 2021.

[65] C.-H. Hong and B. Varghese, "Resource management in Fog/Edge computing: a survey on architectures, infrastructure, and algorithms," ACM Computing Surveys (CSUR), vol. 52, no. 5, pp. 1–37, 2019.

[66] B. Sonkoly, J. Czentye, M. Szalay, B. Németh, and L. Toka, "Survey on placement methods in the Edge and beyond," IEEE Communications Surveys & Tutorials, 2021.

[67] M. Adhikari, S. N. Srirama, and T. Amgoth, "A comprehensive survey on nature-inspired algorithms and their applications in Edge computing: Challenges and future directions," Software: Practice and Experience, 2021.

[68] T. Bahreini, M. Brocanelli, and D. Grosu, "VECMAN: A framework for energy-aware resource management in vehicular Edge computing systems," IEEE Transactions on Mobile Computing, 2021, DOI:10.1109/TMC.2021.3089338, (in press).

[69] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, "Learning-based computation offloading for IoT devices with energy harvesting," IEEE Transactions on Vehicular Technology, vol. 68, no. 2, pp. 1930–1941, 2019.

[70] E. F. Maleki, L. Mashayekhy, and S. M. Nabavinejad, "Mobility-aware computation offloading in Edge computing using machine learning," IEEE Transactions on Mobile Computing, 2021, DOI:10.1109/TMC.2021.3085527, (in press).

[71] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual Edge computing systems via deep reinforcement learning," IEEE Internet of Things Journal, vol. 6, no. 3, pp. 4005–4018, 2019.

[72] A. Asheralieva and D. Niyato, "Learning-based mobile Edge computing resource management to support public blockchain networks," IEEE Transactions on Mobile Computing, vol. 20, no. 3, pp. 1092–1109, 2021.

[73] L. Huang, X. Feng, A. Feng, Y. Huang, and L. P. Qian, "Distributed deep learning-based offloading for mobile Edge computing networks," Mobile networks and applications, pp. 1–8, 2018.

[74] A. Hazra and T. Amgoth, "CeCO: Cost-efficient computation offloading of IoT applications in green industrial Fog networks," IEEE Transactions on Industrial Informatics, 2021, DOI:10.1109/TII.2021.3130255, (in press).

[75] G. Peng, H. Wu, H. Wu, and K. Wolter, "Constrained multi-objective optimization for IoT-enabled computation offloading in collaborative Edge and Cloud computing," IEEE Internet of Things Journal, vol. 8, no. 17, pp. 13 723–13 736, 2021.

[76] M. Goudarzi, M. S. Palaniswami, and R. Buyya, "A distributed deep reinforcement learning technique for application placement in Edge and Fog computing environments," IEEE Transactions on Mobile Computing, 2021, DOI:10.1109/TMC.2021.3123165, (in press).

[77] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, "Efficient dependent task offloading for multiple applications in MEC-Cloud system," IEEE Transactions on Mobile Computing, 2021, DOI:10.1109/TMC.2021.3119200, (in press).

[78] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu, "Optimization of lightweight task offloading strategy for mobile Edge computing based on deep reinforcement learning," Future Generation Computer Systems, vol. 102, pp. 847–861, 2020.

[79] M. Peixoto, T. Genez, and L. F. Bittencourt, "Hierarchical scheduling mechanisms in multi-level Fog computing," IEEE Transactions on Services Computing, 2021, DOI:10.1109/TSC.2021.3079110, (in press).

[80] S. Deng, Z. Xiang, J. Taheri, M. A. Khoshkholghi, J. Yin, A. Y. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed Edges," IEEE Transactions on Mobile Computing, vol. 20, no. 5, pp. 1907–1923, 2020.

[81] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile Edge computing: A reinforcement learning approach," IEEE Transactions on Mobile Computing, vol. 20, no. 3, pp. 939–951, 2021.

[82] T. M. Ho and K.-K. Nguyen, "Joint server selection, cooperative offloading and handover in multi-access Edge computing wireless network: A deep reinforcement learning approach," IEEE Transactions on Mobile Computing, 2020, DOI:10.1109/TMC.2020.3043736, (in press).

[83] C. Zhang and Z. Zheng, "Task migration for mobile Edge computing using deep reinforcement learning," Future Generation Computer Systems, vol. 96, pp. 111–118, 2019.

[84] Z. Cheng, M. Min, M. Liwang, L. Huang, and Z. Gao, "Multi-agent ddpg-based joint task partitioning and power control in Fog computing networks," IEEE Internet of Things Journal, vol. 9, no. 1, pp. 104–116, 2021.

[85] A. Kishor and C. Chakarbarty, "Task offloading in Fog computing for using smart ant colony optimization," Wireless Personal Communications, pp. 1–22, 2021.

[86] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro, "An improved multi-objective genetic algorithm with heuristic initialization for service placement and load distribution in Edge computing," Computer Networks, vol. 194, p. 108146, 2021.

[87] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile Edge computing based on markov decision process," IEEE/ACM Transactions on Networking, vol. 27, no. 3, pp. 1272–1288, 2019.

[88] Z. Han, H. Tan, X.-Y. Li, S. H.-C. Jiang, Y. Li, and F. C. Lau, "Ondisc: Online latency-sensitive job dispatching and scheduling in heterogeneous Edge-Clouds," IEEE/ACM Transactions on Networking, vol. 27, no. 6, pp. 2472–2485, 2019.

[89] W. Chen, D. Wang, and K. Li, "Multi-user multi-task computation offloading in green mobile Edge Cloud computing," IEEE Transactions on Services Computing, vol. 12, no. 5, pp. 726–738, 2019.

[90] C. Wang, S. Zhang, Z. Qian, M. Xiao, J. Wu, B. Ye, and S. Lu, "Joint server assignment and resource management for Edge-based MAR system," IEEE/ACM Transactions on Networking, vol. 28, no. 5, pp. 2378–2391, 2020.

[91] F. Guo, H. Zhang, H. Ji, X. Li, and V. C. Leung, "An efficient computation offloading management scheme in the densely deployed small cell networks with mobile Edge computing," IEEE/ACM Transactions on Networking, vol. 26, no. 6, pp. 2651–2664, 2019.

[92] Z. Hong, W. Chen, H. Huang, S. Guo, and Z. Zheng, "Multi-hop cooperative computation offloading for industrial IoT–Edge–Cloud computing environments," IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 12, pp. 2759–2774, 2019.

[93] I. Sarkar, M. Adhikari, N. Kumar, and S. Kumar, "A collaborative computational offloading strategy for latency-sensitive applications in Fog networks," IEEE Internet of Things Journal, 2021, DOI:10.1109/JIOT.2021.3104324, (in press).

[94] L. Cai, X. Wei, C. Xing, X. Zou, G. Zhang, and X. Wang, "Failure-resilient DAG task scheduling in Edge computing," Computer Networks, vol. 198, p. 108361, 2021.

[95] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading tasks with de-

pendency and service caching in mobile Edge computing," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 11, pp. 2777–2792, 2021.

[96] L. Chen, C. Shen, P. Zhou, and J. Xu, "Collaborative service placement for Edge computing in dense small cell networks," IEEE Transactions on Mobile Computing, vol. 20, no. 2, pp. 377–390, 2021.

[97] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic Edge-Cloud computing environments using A3C learning and residual recurrent neural networks," IEEE Transactions on Mobile Computing, 2020, DOI:10.1109/TMC.2020.3017079, (in press).

[98] R. Zhou, X. Zhang, S. Qin, J. C. Lui, Z. Zhou, H. Huang, and Z. Li, "Online task offloading for 5G small cell networks," IEEE Transactions on Mobile Computing, 2020, DOI:10.1109/TMC.2020.3036390, (in press).

[99] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-Edge computing networks," IEEE Transactions on Mobile Computing, vol. 19, no. 11, pp. 2581–2593, 2020.

[100] B. Yang, X. Cao, J. Bassey, X. Li, and L. Qian, "Computation offloading in multi-access Edge computing: A multi-task learning approach," IEEE transactions on mobile computing, vol. 20, no. 9, pp. 2581–2593, 2021.

[101] X. Wang, Z. Ning, S. Guo, and L. Wang, "Imitation learning enabled task scheduling for online vehicular Edge computing," IEEE Transactions on Mobile Computing, vol. 21, no. 2, pp. 598–611, 2022.

[102] J. L. D. Neto, S.-Y. Yu, D. F. Macedo, J. M. S. Nogueira, R. Langar, and S. Secci, "Uloof: A user level online offloading framework for mobile Edge computing," IEEE Transactions on Mobile Computing, vol. 17, no. 11, pp. 2660–2674, 2018.

[103] M. Tang and V. W. Wong, "Deep reinforcement learning for task offloading in mobile Edge computing systems," IEEE Transactions on Mobile Computing, 2020, DOI:10.1109/TMC.2020.3036871, (in press).

[104] O. Tao, X. Chen, Z. Zhou, L. Li, and X. Tan, "Adaptive user-managed service placement for mobile Edge computing via contextual multi-armed bandit learning," IEEE Transactions on Mobile Computing, 2021, DOI:10.1109/TMC.2021.3106746, (in press).

[105] H. K. Gedawy, K. Habak, K. Harras, and M. Hamdi, "Ramos: A resource-aware multi-objective system for Edge computing," IEEE Transactions on Mobile Computing, vol. 20, no. 8, pp. 2654–2670, 2021.

[106] Z. Yan, P. Cheng, Z. Chen, B. Vucetic, and Y. Li, "Two-dimensional task offloading for mobile networks: An imitation learning framework," IEEE/ACM Transactions on Networking, vol. 29, no. 6, pp. 2494–2507, 2021.

[107] L. Wang, L. Jiao, T. He, J. Li, and H. Bal, "Service placement for collaborative Edge applications," IEEE/ACM Transactions on Networking, vol. 29, no. 1, pp. 34–47, 2021.

[108] V. Farhadi, F. Mehmeti, T. He, T. F. La Porta, H. Khamfroush, S. Wang, K. S. Chan, and K. Poularakis, "Service placement and request scheduling for data-intensive applications in Edge Clouds," IEEE/ACM Transactions on Networking, vol. 29, no. 2, pp. 779–792, 2021.

[109] L. Yang, B. Liu, J. Cao, Y. Sahni, and Z. Wang, "Joint computation partitioning and resource allocation for latency sensitive applications in mobile Edge Clouds," IEEE Transactions on Services Computing, vol. 14, no. 5, pp. 1439–1452, 2021.

[110] Z. Hu, J. Niu, T. Ren, B. Dai, Q. Li, M. Xu, and S. K. Das, "An efficient online computation offloading approach for large-scale mobile Edge computing via deep reinforcement learning," IEEE Transactions on Services Computing, 2021, DOI:10.1109/TSC.2021.3116280, (in press).

[111] D. Kimovski, N. Mehran, C. E. Kerth, and R. Prodan, "Mobility-aware IoT applications placement in the Cloud Edge continuum," IEEE Transactions on Services Computing, 2021, DOI:10.1109/TSC.2021.3094322, (in press).

[112] S. Tian, C. Chi, S. Long, S. Oh, Z. Li, and J. Long, "User preference-based hierarchical offloading for collaborative Cloud-Edge computing," IEEE Transactions on Services Computing, 2021, DOI:10.1109/TSC.2021.3128603, (in press).

[113] S. Ghanavati, J. H. Abawajy, and D. Izadi, "An energy aware task scheduling model using ant-mating optimization in Fog computing environment," IEEE Transactions on Services Computing, 2020, DOI:10.1109/TSC.2020.3028575, (in press).

[114] H. Badri, T. Bahreini, D. Grosu, and K. Yang, "Energy-aware application placement in mobile Edge computing: A stochastic optimization approach," IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 4, pp. 909–922, 2020.

[115] T. Bahreini, H. Badri, and D. Grosu, "Mechanisms for resource allocation and pricing in mobile Edge computing systems," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 3, pp. 667–682, 2022.

[116] Y. Chen, S. Zhang, Y. Jin, Z. Qian, M. Xiao, J. Ge, and S. Lu, "LOCUS: User-perceived delay-aware service placement and user allocation in MEC environment," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 7, pp. 1581–1592, 2022.

[117] Z. Ma, S. Zhang, Z. Chen, T. Han, Z. Qian, M. Xiao, N. Chen, J. Wu, and S. Lu, "Towards revenue-driven multi-user online task offloading in Edge computing," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 5, pp. 1185–1198, 2022.

[118] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in Cloud-Edge continuum," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 8, pp. 1825–1840, 2022.

[119] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in Edge computing based on meta reinforcement learning," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 1, pp. 242–253, 2021.

[120] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 5, pp. 1085–1101, 2021.

[121] J. Meng, H. Tan, X.-Y. Li, Z. Han, and B. Li, "Online deadline-aware task dispatching and scheduling in Edge computing," IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 6, pp. 1270–1286, 2020.

[122] H. Wu, W. J. Knottenbelt, and K. Wolter, "An efficient application partitioning algorithm in mobile environments," IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 7, pp. 1464–1480, 2019.

[123] X. Xu, Q. Liu, Y. Luo, K. Peng, X. Zhang, S. Meng, and L. Qi, "A computation offloading method over big data for IoT-enabled Cloud-Edge computing," Future Generation Computer Systems, vol. 95, pp. 522–533, 2019.

[124] L. Huang, X. Feng, L. Zhang, L. Qian, and Y. Wu, "Multi-server multi-user multi-task computation offloading for mobile Edge computing networks," Sensors, vol. 19, no. 6, p. 1446, 2019.

[125] R. Mahmud, S. N. Srirama, K. Ramamohanarao, and R. Buyya, "Quality of experience QoE-aware placement of applications in Fog computing environments," Journal of Parallel and Distributed Computing, vol. 132, pp. 190–203, 2019.

[126] Y. Nan, W. Li, W. Bao, F. C. Delicato, P. F. Pires, and A. Y. Zomaya, "A dynamic tradeoff data processing framework for delay-sensitive applications in Cloud of things systems," Journal of Parallel and Distributed Computing, vol. 112, pp. 53–66, 2018.

[127] G. L. Stavrinides and H. D. Karatza, "A hybrid approach to scheduling real-time IoT workflows in Fog and Cloud environments," Multimedia Tools and Applications, vol. 78, no. 17, pp. 24 639–24 655, 2019.

[128] R. Mahmud, A. N. Toosi, K. Ramamohanarao, and R. Buyya, "Context-aware

placement of industry 4.0 applications in Fog computing environments," IEEE Transactions on Industrial Informatics, vol. 16, no. 11, pp. 7004–7013, 2020.

[129] E. El Haber, T. M. Nguyen, D. Ebrahimi, and C. Assi, "Computational cost and energy efficient task offloading in hierarchical Edge-Clouds," in 2018 IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC). IEEE, 2018, pp. 1–6.

[130] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile Edge computing," Digital Communications and Networks, vol. 5, no. 1, pp. 10–17, 2019.

[131] H. Lu, X. He, M. Du, X. Ruan, Y. Sun, and K. Wang, "Edge QoE: Computation offloading with deep reinforcement learning for Internet of Things," IEEE Internet of Things Journal, vol. 7, no. 10, pp. 9255–9265, 2020.

[132] X. Xiong, K. Zheng, L. Lei, and L. Hou, "Resource allocation based on deep reinforcement learning in IoT Edge computing," IEEE Journal on Selected Areas in Communications, vol. 38, no. 6, pp. 1133–1146, 2020.

[133] P. Gazori, D. Rahbari, and M. Nickray, "Saving time and cost on the scheduling of Fog-based IoT applications using deep reinforcement learning approach," Future Generation Computer Systems, vol. 110, pp. 1098–1115, 2020.

[134] Z. Wang, Z. Zhao, G. Min, X. Huang, Q. Ni, and R. Wang, "User mobility aware task assignment for mobile Edge computing," Future Generation Computer Systems, vol. 85, pp. 1–8, 2018.

[135] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the Edge: Mobility-aware dynamic service placement for mobile Edge computing," IEEE Journal on Selected Areas in Communications, vol. 36, no. 10, pp. 2333–2345, 2018.

[136] S. Shekhar, A. Chhokra, H. Sun, A. Gokhale, A. Dubey, and X. Koutsoukos, "Urmila: A performance and mobility-aware Fog/Edge resource management middleware," in Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC). IEEE, 2019, pp. 118–125.

[137] D. Wang, Z. Liu, X. Wang, and Y. Lan, "Mobility-aware task offloading and migration schemes in Fog computing networks," IEEE Access, vol. 7, pp. 43 356–43 368, 2019.

[138] Z. Liu, X. Wang, D. Wang, Y. Lan, and J. Hou, "Mobility-aware task offloading and migration schemes in scns with mobile Edge computing," in Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC). IEEE, 2019, pp. 1–6.

[139] C. Yang, Y. Liu, X. Chen, W. Zhong, and S. Xie, "Efficient mobility-aware task offloading for vehicular Edge computing networks," IEEE Access, vol. 7, pp. 26 652–26 664, 2019.

[140] Q. Qi, J. Wang, Z. Ma, H. Sun, Y. Cao, L. Zhang, and J. Liao, "Knowledge-driven service offloading decision for vehicular Edge computing: A deep reinforcement learning approach," IEEE Transactions on Vehicular Technology, vol. 68, no. 5, pp. 4192–4203, 2019.

[141] C. Zhu, G. Pastor, Y. Xiao, Y. Li, and A. Ylae-Jaeaeski, "Fog following me: Latency and quality balanced task allocation in vehicular Fog computing," in Proceedings of the 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON). IEEE, 2018, pp. 1–9.

[142] D. Rahbari and M. Nickray, "Task offloading in mobile Fog computing by classification and regression tree," Peer-to-Peer Networking and Applications, vol. 13, no. 1, pp. 104–122, 2020.

[143] J. Sheng, J. Hu, X. Teng, B. Wang, and X. Pan, "Computation offloading strategy in mobile Edge computing," Information, vol. 10, no. 6, p. 191, 2019.

[144] J. Xie, Y. Jia, Z. Chen, and L. Liang, "Mobility-aware task parallel offloading for vehicle Fog computing," in Proceedings of the International Conference on Artificial Intelligence for Communications and Networks. Springer, 2019, pp. 367–379.

[145] P. Maiti, H. K. Apat, B. Sahoo, and A. K. Turuk, "An effective approach of latency-

aware Fog smart gateways deployment for IoT services," Internet of Things, vol. 8, p. 100091, 2019.

[146] M. Selimi, L. Cerdà Alabern, F. Freitag, L. Veiga, A. Sathiaseelan, and J. Crowcroft, "A lightweight service placement approach for community network micro-Clouds," Journal of Grid Computing, vol. 17, no. 1, pp. 169–189, 2019.

[147] C. Mouradian, S. Kianpisheh, M. Abu-Lebdeh, F. Ebrahimnezhad, N. T. Jahromi, and R. H. Glitho, "Application component placement in NFV-based hybrid Cloud/Fog systems with mobile Fog nodes," IEEE Journal on Selected Areas in Communications, vol. 37, no. 5, pp. 1130–1143, 2019.

[148] H. Bashir, S. Lee, and K. H. Kim, "Resource allocation through logistic regression and multicriteria decision making method in IoT Fog computing," Transactions on Emerging Telecommunications Technologies, p. e3824, 2019.

[149] M. K. Hussein and M. H. Mousa, "Efficient task offloading for IoT-based applications in Fog computing using ant colony optimization," IEEE Access, vol. 8, pp. 37 191–37 201, 2020.

[150] H. Sami, A. Mourad, and W. El-Hajj, "Vehicular-OBUs-as-on-demand-Fogs: Resource and context aware deployment of containerized micro-services," IEEE/ACM Transactions on Networking, vol. 28, no. 2, pp. 778–790, 2020.

[151] R. O. Aburukba, T. Landolsi, and D. Omer, "A heuristic scheduling approach for Fog-Cloud computing environment with stationary IoT devices," Journal of Network and Computer Applications, vol. 180, p. 102994, 2021.

[152] M. Abd Elaziz, L. Abualigah, and I. Attiya, "Advanced optimization technique for scheduling IoT tasks in Cloud-Fog computing environments," Future Generation Computer Systems, 2021.

[153] F. Hoseiny, S. Azizi, M. Shojafar, F. Ahmadiazar, and R. Tafazolli, "PGA: a priority-aware genetic algorithm for task scheduling in heterogeneous Fog-Cloud computing," in Proceedings of the IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2021, pp. 1–6.

[154] S. Ijaz, E. U. Munir, S. G. Ahmad, M. M. Rafique, and O. F. Rana, "Energy-makespan optimization of workflow scheduling in Fog–Cloud computing," Computing, pp. 1–27, 2021.

[155] C. Ju, Y. Ma, Z. Yin, and F. Zhang, "An request offloading and scheduling approach base on particle swarm optimization algorithm in Fog networks," in Proceedings of the 13th International Conference on Communication Software and Networks (ICCSN). IEEE, 2021, pp. 185–188.

[156] B. Liu, X. Xu, L. Qi, Q. Ni, and W. Dou, "Task scheduling with precedence and placement constraints for resource utilization improvement in multi-user MEC environment," Journal of Systems Architecture, vol. 114, p. 101970, 2021.

[157] B. Natesha and R. M. R. Guddeti, "Adopting elitism-based genetic algorithm for minimizing multi-objective problems of IoT service placement in Fog computing environment," Journal of Network and Computer Applications, vol. 178, p. 102972, 2021.

[158] T. Nguyen, T. Nguyen, Q.-H. Vu, T. T. B. Huynh, and B. M. Nguyen, "Multi-objective sparrow search optimization for task scheduling in Fog-Cloud-Blockchain systems," in Proceedings of the IEEE International Conference on Services Computing (SCC). IEEE, 2021, pp. 450–455.

[159] T. Pusztai, F. Rossi, and S. Dustdar, "Pogonip: Scheduling asynchronous applications on the Edge," in Proceedings of the IEEE 14th International Conference on Cloud Computing (CLOUD). IEEE, 2021, pp. 660–670.

[160] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Demand-driven deep reinforcement learning for scalable Fog and service placement," IEEE Transactions on Services Computing, 2021, DOI:10.1109/TSC.2021.3075988, (in press).

[161] J. Wang, J. Hu, G. Min, W. Zhan, A. Zomaya, and N. Georgalas, "Dependent task offloading for Edge computing based on deep reinforcement learning," IEEE Transactions on Computers, 2021, DOI:10.1109/TC.2021.3131040, (in press).

[162] L. Pan, X. Liu, Z. Jia, J. Xu, and X. Li, "A multi-objective clustering evolutionary algorithm for multi-workflow computation offloading in mobile Edge computing," IEEE Transactions on Cloud Computing, 2021.

[163] M. Goudarzi, M. Palaniswami, and R. Buyya, "A Fog-driven dynamic resource allocation technique in ultra dense femtocell networks," Journal of Network and Computer Applications, vol. 145, p. 102407, 2019.

[164] M. Goudarzi, Q. Deng, and R. Buyya, "Resource management in Edge and Fog computing using FogBus2 framework," in Managing Internet of Things Applications across Edge and Cloud Data Centres, R. Ranjan, K. Mitra, P. P. Jayaraman, , and A. Y. Zomaya, Eds. IET, 2022.

[165] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of Cloud computing environments and evaluation of resource provisioning algorithms," Software: Practice and experience, vol. 41, no. 1, pp. 23–50, 2011.

[166] J. Garcia-Morales, G. Femenias, and F. Riera-Palou, "Analytical performance evaluation of OFDMA-based heterogeneous cellular networks using ffr," in Proceedings of the 81th IEEE Vehicular Technology Conference (VTC Spring). IEEE, 2015, pp. 1–7.

[167] F. Fu, Z. Lu, Y. Xie, W. Jing, and X. Wen, "Clustering-based low-complexity resource allocation in two-tier femtocell networks with QoS provisioning," International Journal of Communication Systems, vol. 30, no. 4, 2017.

[168] F. Mhiri, K. Sethom, and R. Bouallegue, "A survey on interference management techniques in femtocell self-organizing networks," Journal of Network and Computer Applications, vol. 36, no. 1, pp. 58–65, 2013.

[169] S. Bu, F. R. Yu, and H. Yanikomeroglu, "Interference-aware energy-efficient resource allocation for OFDMA-based heterogeneous networks with incomplete channel state information," IEEE Transactions on Vehicular Technology, vol. 64, no. 3, pp. 1036–1050, 2015.

[170] K. Rohoden, R. Estrada, H. Otrok, and Z. Dziong, "Stable femtocells cluster formation and resource allocation based on cooperative game theory," Computer Communications, vol. 134, pp. 30–41, 2019.

[171] J. Qiu, G. Ding, Q. Wu, Z. Qian, T. A. Tsiftsis, Z. Du, and Y. Sun, "Hierarchical resource allocation framework for hyper-dense small cell networks," IEEE Access, vol. 4, pp. 8657–8669, 2016.

[172] W. Li, W. Zheng, X. Wen, and T. Su, "Dynamic clustering based sub-band allocation in dense femtocell environments," in Proceedings of the 75th IEEE Vehicular Technology Conference (VTC Spring). IEEE, 2012, pp. 1–5.

[173] A. Abdelnasser, E. Hossain, and D. I. Kim, "Clustering and resource allocation for dense femtocells in a two-tier cellular OFDMA network," IEEE Transactions on Wireless Communications, vol. 13, no. 3, pp. 1628–1641, 2014.

[174] L. Li and Z. Zhou, "Cluster based resource allocation in two-tier hetnets with hierarchical throughput constraints," International Journal of Communication Systems, vol. 30, no. 14, p. e3292, 2017.

[175] W. Li and J. Zhang, "Cluster-based resource allocation scheme with QoS guarantee in ultra-dense networks," IET Communications, vol. 12, no. 7, pp. 861–867, 2018.

[176] Q. Zhang, X. Zhu, L. Wu, and K. Sandrasegaran, "A coloring-based resource allocation for OFDMA femtocell networks," in Proceeding of the Wireless Communications and Networking Conference (WCNC). IEEE, 2013, pp. 673–678.

[177] A. Hatoum, R. Langar, N. Aitsaadi, R. Boutaba, and G. Pujolle, "Cluster-based resource management in OFDMA femtocell networks with QoS guarantees," IEEE Transactions on Vehicular Technology, vol. 63, no. 5, pp. 2378–2391, 2014.

[178] A. Pratap, R. Singhal, R. Misra, and S. K. Das, "Distributed randomized *k*-clustering based pcid assignment for ultra-dense femtocellular networks," IEEE Transactions on Parallel and Distributed Systems, 2018.

[179] J. Kim and D.-H. Cho, "A joint power and subchannel allocation scheme maximizing system capacity in indoor dense mobile communication systems," IEEE Transactions on Vehicular Technology, vol. 59, no. 9, pp. 4340–4353, 2010.

[180] 3GPP, "Further advancements for E-UTRA physical layer aspects," TR 36.814, 2010.

[181] Y. L. Lee, J. Loo, T. C. Chuah, and A. A. El-Saleh, "Multi-objective resource allocation for LTE/LTE-A femtocell/HeNB networks using ant colony optimization," Wireless Personal Communications, vol. 92, no. 2, pp. 565–586, 2017.

[182] F. Capozzi, G. Piro, L. A. Grieco, G. Boggia, and P. Camarda, "Downlink packet scheduling in LTE cellular networks: Key design issues and a survey," IEEE Communications Surveys & Tutorials, vol. 15, no. 2, pp. 678–700, 2013.

[183] N.-T. Le, D. Jayalath, and J. Coetzee, "Spectral-efficient resource allocation for mixed services in OFDMA-based 5G heterogeneous networks," Transactions on Emerging Telecommunications Technologies, vol. 29, no. 1, p. e3267, 2018.

[184] V. N. Ha and L. B. Le, "Fair resource allocation for OFDMA femtocell networks with macrocell protection," IEEE Transactions on Vehicular Technology, vol. 63, no. 3, pp. 1388–1401, 2014.

[185] P. Mach and Z. Becvar, "QoS-guaranteed power control mechanism based on the frame utilization for femtocells," EURASIP Journal on Wireless Communications and Networking, vol. 2011, no. 1, pp. 253–259, 2011.

[186] L. Tan, Z. Feng, W. Li, Z. Jing, and T. A. Gulliver, "Graph coloring based spectrum allocation for femtocell downlink interference mitigation," in Proceedings of the Wireless Communications and Networking Conference (WCNC). IEEE, 2011, pp. 1248–1252.

[187] J. Pitman, "Some probabilistic aspects of set partitions," The American mathematical monthly, vol. 104, no. 3, pp. 201–209, 1997.

[188] K. Sundaresan and S. Rangarajan, "Efficient resource management in OFDMA femto cells," in Proceedings of the 10th ACM international symposium on Mobile ad hoc networking and computing. ACM, 2009, pp. 33–42.

[189] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination," Eastern Research Laboratory, Digital Equipment Corporation: Hudson, MA, USA, pp. 2–7, 1984.

[190] D. G. Roy, D. De, A. Mukherjee, and R. Buyya, "Application-aware cloudlet selection for computation offloading in multi-cloudlet environment," The Journal of Supercomputing, vol. 73, no. 4, pp. 1672–1690, 2017.

[191] M.-H. Chen, B. Liang, and M. Dong, "Joint offloading and resource allocation for computation and communication in mobile Cloud with computing access point," in Proceeding of the IEEE Conference on Computer Communications (INFOCOM). IEEE, 2017, pp. 1–9.

[192] L. Lin, P. Li, X. Liao, H. Jin, and Y. Zhang, "Echo: An Edge-centric code offloading system with quality of service guarantee," IEEE Access, vol. 7, pp. 5905–5917, 2018.

[193] G. L. Stavrinides and H. D. Karatza, "A hybrid approach to scheduling real-time IoT workflows in Fog and Cloud environments," Multimedia Tools and Applications, pp. 1–17, 2018.

[194] S. Bi, L. Huang, and Y.-J. A. Zhang, "Joint optimization of service caching placement and computation offloading in mobile Edge computing systems," IEEE Transactions on Wireless Communications, vol. 19, pp. 4947–4963, 2020.

[195] M. Goudarzi, M. Zamani, and A. Toroghi Haghighat, "A genetic-based decision algorithm for multisite computation offloading in mobile Cloud computing," International Journal of Communication Systems, vol. 30, no. 10, p. e3241, 2017.

[196] X. Chen, Y.-S. Ong, M.-H. Lim, and K. C. Tan, "A multi-facet survey on memetic computation," IEEE Transactions on Evolutionary Computation, vol. 15, no. 5, pp. 591–607, 2011.

[197] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile Cloud computing," ACM SIGMETRICS Performance Evaluation Review, vol. 40, no. 4, pp. 23–32, 2013.

[198] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" Computer, no. 4, pp. 51–56, 2010.

[199] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar, "Mobility-aware application scheduling in Fog computing," IEEE Cloud Computing, vol. 4, no. 2, pp. 26–35, 2017.

[200] M. Taneja and A. Davy, "Resource aware placement of IoT application modules in Fog-Cloud computing paradigm," in Proceedings of the IFIP/IEEE Symposium on Integrated Network and Service Management (IM). IEEE, 2017, pp. 1222–1228.

[201] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live service migration in mobile Edge Clouds," IEEE Wireless Communications, vol. 25, no. 1, pp. 140–147, 2017.

[202] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-Clouds with predicted future costs," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 4, pp. 1002–1016, 2016.

[203] M. Adhikari, S. N. Srirama, and T. Amgoth, "Application offloading strategy for hierarchical Fog environment through swarm optimization," IEEE Internet of Things Journal, vol. 7, no. 5, pp. 4317–4328, 2019.

[204] F. Yu, H. Chen, and J. Xu, "DMPO: Dynamic mobility-aware partial offloading in mobile Edge computing," Future Generation Computer Systems, vol. 89, pp. 722–735, 2018.

[205] Y. Sun, S. Zhou, and J. Xu, "EMM: Energy-aware mobility management for mobile Edge computing in ultra dense networks," IEEE Journal on Selected Areas in Communications, vol. 35, no. 11, pp. 2637–2646, 2017.

[206] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the Fog: a performance evaluation," Sensors, vol. 19, no. 7, p. 1488, 2019.

[207] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, "Towards efficient Edge Cloud augmentation for virtual reality MMOGs," in Proceedings of the Second ACM/IEEE Symposium on Edge Computing, 2017, pp. 1–14.

[208] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in Proceedings of the 9th international conference on Mobile systems, applications, and services, 2011, pp. 43–56.

[209] T. N. Gia, M. Jiang, A.-M. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen, "Fog computing in healthcare Internet of Things: A case study on ECG feature extraction," in Proceedings of the IEEE international conference on computer and information technology; ubiquitous computing and communications; dependable, autonomic and secure computing; pervasive intelligence and computing. IEEE, 2015, pp. 356–363.

[210] A. Al-Shuwaili and O. Simeone, "Energy-efficient resource allocation for mobile Edge computing-based augmented reality applications," IEEE Wireless Communications Letters, vol. 6, no. 3, pp. 398–401, 2017.

[211] A. Brogi and S. Forti, "QoS-aware deployment of IoT applications through the Fog," IEEE Internet of Things Journal, vol. 4, no. 5, pp. 1185–1192, 2017.

[212] D. Jeff, "ML for system, system for ML, keynote talk in workshop on ML for systems, NIPS," 2018.

[213] M. Goudarzi, M. Palaniswami, and R. Buyya, "Scheduling IoT applications in Edge and Fog computing environments: A taxonomy and future directions," arXiv preprint arXiv:2204.12580, 2022.

[214] Q. Mao, F. Hu, and Q. Hao, "Deep learning for intelligent wireless networks: A

comprehensive survey," IEEE Communications Surveys & Tutorials, vol. 20, no. 4, pp. 2595–2621, 2018.

[215] Y. Sun, M. Peng, Y. Zhou, Y. Huang, and S. Mao, "Application of machine learning in wireless networks: Key techniques and open issues," IEEE Communications Surveys & Tutorials, vol. 21, no. 4, pp. 3072–3108, 2019.

[216] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning et al., "IMPALA: Scalable distributed deep-rl with importance weighted actor-learner architectures," in Proceedings of the International Conference on Machine Learning (ICML). PMLR, 2018, pp. 1407–1416.

[217] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu, "iRAF: A deep reinforcement learning approach for collaborative mobile Edge computing IoT networks," IEEE Internet of Things Journal, vol. 6, no. 4, pp. 7011–7024, 2019.

[218] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in Internet of everything. Springer, 2018, pp. 103–130.

[219] S. E. Mahmoodi, R. Uma, and K. Subbalakshmi, "Optimal joint scheduling and Cloud offloading for mobile applications," IEEE Transactions on Cloud Computing, vol. 7, no. 2, pp. 301–313, 2016.

[220] G. Fox, J. A. Glazier, J. Kadupitiya, V. Jadhao, M. Kim, J. Qiu, J. P. Sluka, E. Somogyi, M. Marathe, A. Adiga et al., "Learning everywhere: Pervasive machine learning for effective high-performance computation," in Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2019, pp. 422–429.

[221] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in Proceedings of the International Conference on Learning Representations (ICLR), 2015.

[222] J. Appleyard, T. Kociskỳ, and P. Blunsom, "Optimizing performance of re-

current neural networks on GPUs. corr abs/1604.01946 (2016)," arXiv preprint arXiv:1604.01946, 2016.

[223] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 3, pp. 682–694, 2013.

[224] Q. Li, Z.-y. Wang, W.-h. Li, J. Li, C. Wang, and R.-y. Du, "Applications integration in a hybrid Cloud computing environment: Modelling and platform," Enterprise Information Systems, vol. 7, no. 3, pp. 237–271, 2013.

[225] P. Schulz, M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel et al., "Latency critical IoT applications in 5G: Perspective on the design of radio interface and network architecture," IEEE Communications Magazine, vol. 55, no. 2, pp. 70–78, 2017.

[226] G. Merlino, R. Dautov, S. Distefano, and D. Bruneo, "Enabling workload engineering in Edge, Fog, and Cloud computing through openstack-based middleware," ACM Transactions on Internet Technology (TOIT), vol. 19, no. 2, pp. 1–22, 2019.

[227] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "Fogbus: A blockchain-based lightweight framework for Edge and Fog computing," Journal of Systems and Software, vol. 154, pp. 22–36, 2019.

[228] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H. C. Cankaya, Q. Zhang, W. Xie, and J. P. Jue, "FogPlan: a lightweight QoS-aware dynamic Fog service provisioning framework," IEEE Internet of Things Journal, vol. 6, no. 3, pp. 5080–5096, 2019.

[229] D. T. Nguyen, L. B. Le, and V. K. Bhargava, "A market-based framework for multi-resource allocation in Fog computing," IEEE/ACM Transactions on Networking, vol. 27, no. 3, pp. 1151–1164, 2019.

[230] P. Bellavista and A. Zanni, "Feasibility of Fog computing deployment based on Docker containerization over raspberrypi," in Proceedings of the 18th

international conference on distributed computing and networking, 2017, pp. 1–10.

[231] A. J. Ferrer, J. M. Marques, and J. Jorba, "Ad-hoc Edge Cloud: A framework for dynamic creation of Edge computing infrastructures," in Proceedings of the 28th International Conference on Computer Communication and Networks. IEEE, 2019, pp. 1–7.

[232] S. Noor, B. Koehler, A. Steenson, J. Caballero, D. Ellenberger, and L. Heilman, "IoTDoc: A Docker-container based architecture of IoT-enabled Cloud system," in Proceedings of the 3rd IEEE/ACIS International Conference on Big Data, Cloud Computing, and Data Science Engineering. Springer, 2019, pp. 51–68.