Task Allocation and Scheduling Algorithms for Network Edge Applications

Olamilekan Fadahunsi

School of Computer Science, McGill University
Montreal, Canada

August 2021

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dedication

Dedicated to my parents and grandparents.

Acknowledgements

I want to thank my supervisor Professor Muthucumaru Maheswaran for giving me the opportunity to join his research group and pursue my Ph.D. under his supervision. He motivated and supported me during the course of my Ph.D. studies. His continuous feedback and guidance helped me to improve the quality of my research. I also want to thank my committee members - Profs. Clark Verbrugge and Jorg Kienzle for their invaluable comments and feedback during my Ph.D. program.

I enjoyed the work environment of the School of Computer Science at McGill University. The positive and friendly attitude of my teammates and colleagues at *Cloud, IoT and Edge (CITE)* Lab: Richard Olaniyan, Ben Yu, Richboy David Echomgbe, Jiahui Peng and Yuxiang Ma. I would like to thank them for the valuable and sometimes fun discussions and all the good times.

I would like to acknowledge the organizations that sponsored my research. This research was sponsored in part by the Nigerian Government through the Presidential Special Scholarship for Innovation and Development (PRESSID) program, a Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant, a Grad Excellence Award in Computer Science and a MITACS/Ericsson Accelerate Grant.

Finally, I want to express my high gratitude to my family, especially my mum, whose devotion and sacrifices I can never compensate.

Abstract

Advances in cloud computing enabled the development of exciting applications in the Internet of Things (IoT) domain, however, the higher latency reaching the clouds from the IoT has motivated the need for the distribution of cloud resources towards the network edge. One of the complications of the distribution of cloud resources is resource management, which typically revolves around resource and task allocation, task scheduling, workload balance and quality of service (QoS) to achieve performance improvements. The problem of task allocation and scheduling is more challenging in the distributed configuration when compared to the centralized version of cloud computing.

The geo-distribution of cloud servers complicates the server selection problem for a given task. In this thesis, we develop algorithms for task allocation and scheduling problems when the cloud servers are distributed to the network edge from the core. In the first part, we propose a two-phase approach that focuses on the allocation and scheduling of independent tasks. The first phase decides the allocation of the devices to the fogs (distributed cloud servers close to the network edge). The fogs are allocated in a two-tiered manner, i.e., for each device, a home fog and a pool of backup fogs are allocated. In the second phase, the task requests from the devices are routed to the allocated fogs or the cloud. Using simulation studies, we compared the performance of the proposed allocation algorithms against existing ones. The results indicate that the proposed algorithms outperform existing ones.

Applications in IoT tend to have a mix of real-time and non real-time requirements, which complicates the task allocation and scheduling problems. In the second part of this thesis, we propose a two-stage scheduling framework that can map the tasks of the application across a collection of edge computing servers (distributed cloud servers that are closer to the network edge than fogs). The application model focuses on a mix of real-time and non real-time independent tasks. In the first stage, the framework does a task to resource matching, taking into consideration the requirements associated with the task. In the second stage,

the framework schedules the tasks such that real-time deadlines of the tasks are respected, and non real-time tasks are scheduled to use available free time slots. Both stages of the framework completely operate in the user-level and deadline sensitive scheduling is performed by divvying up the CPU resources efficiently in the user space. We implement the framework over heterogeneous collections of machines and measure its performance under different conditions. Results show that the two-stage architecture is better because the flexibility offered by the architecture can be used by the edge servers to obtain higher performance.

While task allocation and scheduling are important, we also illustrate that edge computing is more susceptible to latency variation in the last hop. To handle this problem, we present an architecture for network-aware scheduling for edge applications and develop algorithms that go into the architecture. The architecture observes the state of the network and schedules the application using the observed knowledge. Our proposed network-aware algorithm deploys the best schedule that minimizes the application execution time given the network conditions. As part of the architecture, we develop a switching algorithm that speculatively deploys better schedules when a network change is detected. The combination of both algorithms in the architecture offers better performance to the application. We evaluate the algorithms by comparing them to a baseline approach that is network-unaware. Our results show the performance benefits of the proposed algorithms over the network-unaware approach.

Résumé

Les progrès du cloud computing ont permis le développement d'applications passionnantes dans le domaine de l'Internet des objets (IoT). Cependant, la latence plus élevée atteignant les clouds à partir de l'IoT a motivé le besoin de distribuer les ressources cloud vers la périphérie du réseau. L'une des complications de la distribution des ressources cloud est la gestion des ressources, qui tourne généralement autour de l'allocation des ressources et des tâches, de la planification des tâches, de l'équilibre de la charge de travail et de la qualité de service (QoS) pour améliorer les performances. Le problème de l'allocation et de la planification des tâches est plus difficile dans la configuration distribuée par rapport à la version centralisée du cloud computing.

La géo-distribution des serveurs cloud complique le problème de sélection de serveur pour une tâche donnée. Dans cette thèse, nous développons des algorithmes pour les problèmes d'allocation de tâches et de planification lorsque les serveurs cloud sont distribués à la périphérie du réseau à partir du cœur. Dans la première partie, nous proposons une approche en deux phases qui se concentre sur l'allocation et l'ordonnancement de tâches indépendantes. La première phase décide de l'affectation des appareils aux fogs (serveurs cloud distribués proches de la périphérie du réseau). Les brouillards sont alloués de manière à deux niveaux, c'est-à-dire que pour chaque appareil, un brouillard domestique et un pool de brouillards de secours sont alloués. Dans la deuxième phase, les demandes de tâches des appareils sont acheminées vers les brouillards alloués ou le cloud. À l'aide d'études de simulation, nous avons comparé les performances des algorithmes d'allocation proposés à ceux existants. Les résultats ont indiqué que les algorithmes proposés surpassent ceux existants.

Les applications dans l'IoT ont tendance à avoir un mélange d'exigences en temps réel et en temps non réel, ce qui complique les problèmes d'allocation des tâches et de planification. Dans la deuxième partie de cette thèse, nous proposons un cadre de planification en deux étapes qui permet de cartographier les tâches de l'application sur un ensemble de serveurs informatiques de périphérie (serveurs

cloud distribués plus proches de la périphérie du réseau que les brouillards). Le modèle d'application se concentre sur un mélange de tâches indépendantes en temps réel et non en temps réel. Dans la première étape, le cadre effectue une correspondance entre la tâche et les ressources en tenant compte des exigences associées à la tâche. Dans la deuxième étape, le cadre planifie les tâches de telle sorte que les échéances en temps réel des tâches soient respectées, et les tâches en temps non réel sont planifiées pour utiliser les plages horaires libres disponibles. Les deux étapes du cadre fonctionnent complètement au niveau de l'utilisateur et la planification sensible aux délais est effectuée en répartissant efficacement les ressources CPU dans l'espace utilisateur. Nous implémentons le framework sur des collections hétérogènes de machines et mesurons ses performances dans différentes conditions. Les résultats montrent que l'architecture en deux étapes est meilleure car la flexibilité offerte par l'architecture peut être utilisée par les serveurs de périphérie pour obtenir des performances plus élevées.

Bien que l'allocation et la planification des tâches soient importantes, nous illustrons également que l'edge computing est plus sensible aux variations de latence dans le dernier saut. Pour gérer ce problème, nous présentons une architecture d'ordonnancement sensible au réseau pour les applications de périphérie et développons des algorithmes qui entrent dans l'architecture. L'architecture observe l'état du réseau et programme l'application en utilisant les connaissances observées. L'algorithme réseau que nous proposons déploie le meilleur calendrier qui minimise le temps d'exécution de l'application compte tenu des conditions du réseau. Dans le cadre de l'architecture, nous développons un algorithme de commutation qui déploie de manière spéculative de meilleurs horaires lorsqu'un changement de réseau est détecté. La combinaison des deux algorithmes dans l'architecture offre de meilleures performances à l'application. Nous évaluons les algorithmes en les comparant à une approche de base qui ignore le réseau. Nos résultats montrent les avantages en termes de performances des algorithmes proposés par rapport à l'approche non consciente du réseau.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ericsson predicts that 45% of the global internet data will be generated by the Internet of Things (IoT) devices in 2026 [3]. Cloud computing, the de facto computing backend for mobiles, is not suitable for IoT, mainly because IoT would generate data at much higher rates and clouds are not best suited for the timely processing of such data [4, 5]. Also, the decentralized nature of IoT does not fit the centralized nature of the cloud as data sourced in a distributed fashion is sent to the centralized cloud for processing, which results in high link delays, low bandwidth between IoT devices as well as IoT devices and potential users [6, 7]. This problem has created the need for distributing the cloud infrastructures closer to the network edge where the IoT would live [8, 9, 10].

Organizations need scalable networked computing systems to meet the increasing demands of new and evolving application use cases, from intelligent traffic management to streaming media, and to deliver the highest-level quality of service. Distributing cloud resources to the network edge, closer to where the data is created and consumed, provides performance gains for latency-sensitive IoT applications. One of the central challenges of distributing cloud resources to the network edge is resource management [11, 12], which typically revolves around task allocation and scheduling at the edge. However, the different application types at the edge require scheduling solutions different from what is used in the centralized version of the cloud [13, 14, 15, 16, 17].

The solutions will need to satisfy requirements such as processing requests in real-time on edge nodes. Current cloud computing frameworks such as the Amazon web service [18], Microsoft Azure [19] and Google App Engine [20] can support data-intensive applications, however, implementing real-time data processing at the network edge is still a challenge [21]. For example, IoTs capture data in real-time from real-world situations and want to glean intelligence about the physical situation from their observations [22, 23]. The intelligence acquisition must be completed within given time constraints so that IoT can react to events in the real world safely [24, 25, 26].

In practice, many network edge applications can be composed of multiple modules performing different tasks, and the tasks can be either independent of each other or mutually dependent [27] as shown in Fig. 1.1. Furthermore, they can be broken down into tasks that have deadlines and tasks that do not have deadlines. Applications such as deep learning algorithms and neural networks demand powerful edge resources, whereas real-time applications such as self-driving cars require real-time responses. The different classes of applications necessitate the need for efficient scheduling algorithms [28].

First, it is important to schedule the tasks such that they are executed in a timely fashion and make optimal use of the available resources [29]. One important question to be considered in the distributed cloud landscape is to determine which tasks should be executed in the edge layer, and which ones in the cloud layer [30]. The proper scheduling of the tasks can help design novel applications such as smart traffic systems, smart homes, along with reducing the processing delays.

Second, some applications can have tasks with real-time and non real-time requirements [31]. While tasks with non real-time requirements can tolerate some quality loss in the computed results, it is important to guarantee deadline compliance for the real-time tasks. Scheduling such applications not only requires minimizing response times, but ensuring the timing constraint on the application

is met. Scheduling such tasks can lead to designing novel future applications in robotics and edge artificial intelligence (AI) etc.

Third, with the introduction of 5G networks, there is high network capacity available for device connections in the distributed cloud landscape. However, there are scenarios where there could be network link variations, so that high network capacity is not always available [32]. Scheduling applications with network link variations becomes a challenge [33, 34]. It is important to schedule the tasks while taking the network conditions into consideration. The proper scheduling of such tasks will be useful for designing applications related to autonomous vehicles, vehicle-to-everything (V2X), and so forth.

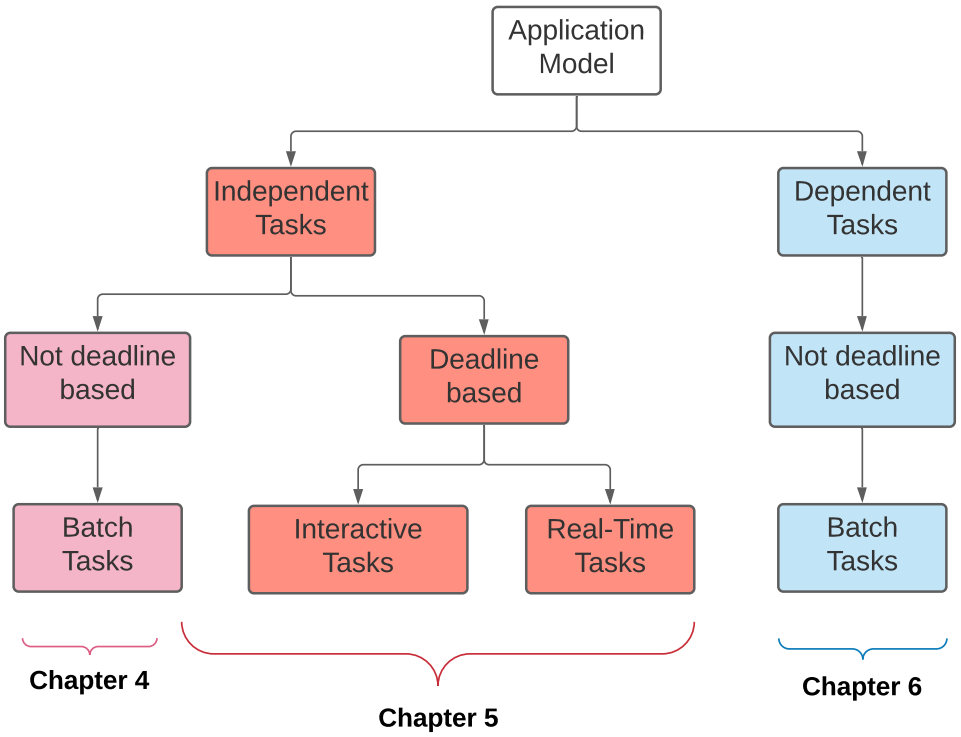

Figure 1.1 Network edge application model and chapters covering the different classes.

In this thesis, we handle the allocation and scheduling problem by proposing

algorithms for the different application classes. In Chapter 4, we use a traffic application as a case study and focus on independent batch tasks as shown in Fig. 1.1. The objective is to minimize the response times of the application tasks. Chapter 5 uses a collaborative robotic application and focuses on a mix of independent real-time and interactive tasks with deadlines and batch tasks without deadlines. The objective is to ensure that the deadline of the real-time tasks is met while maximizing the throughput of the non real-time tasks. In both Chapters 4 and 5, we assume that edge computing resources can always be available at a very short network link away from the origin of data and the network link connecting the origin to the edge has a very high capacity so that data can be transferred at high rates. While this assumption is nominally true, there can be many situations where this assumption is not true. For example, a mobile device or vehicle can roam away from high coverage areas to low coverage areas and suffer bad connection [35, 32]. The network link will not have the expected performance when the device is dwelling in the low to no coverage area. Therefore, variations on network links would significantly impact the performance of the application. In Chapter 6, we introduce network link variation into the scheduling problem and focus on dependent tasks that can be part of a streaming media application. The objective is to minimize the schedule makespan and improve the application performance.

1.1 Thesis Contributions

This thesis makes the following contributions:

1. We propose a two-phase design for resource and task allocation of independent batch tasks. In the first phase called the offline phase, we mathematically formulate the resource allocation problem in the form of integer linear programming (ILP), which could be easily solved with common ILP solvers. In the second phase called the online phase, we develop a probing algorithm to route the task requests to the resources. The performance of the proposed solutions are measured through several metrics, i.e., (i) average response times, (ii) server utilization and (iii) the above metrics considering different failure probabilities.

We evaluate the performance of the proposed algorithms through extensive simulations and compare to existing solutions under different runtime conditions and configurations. The results show the performance benefits of our approach over existing solutions.

2. We further propose a two-stage scheduling solution for a mix of independent real-time and non-real time tasks at the network edge. The first stage periodically recomputes the schedule to address the changing node configurations and application requirements of the real-time tasks while the second stage uses the schedule obtained from the first stage to execute the real-time tasks and the available free-time slots to execute the non real-time tasks.

   We show the feasibility of our approach by developing schedulability tests for the proposed scheduling algorithms. We demonstrate that our proposed scheduler works with high precision. We implemented the scheduling solution over a heterogeneous collection of machines and measured its performance under different conditions. Our results show that the two-stage architecture is better because of the flexibility offered by the architecture can be used to obtain higher overall performance.

3. We develop a network-aware scheduling architecture for dependent tasks of an application. The proposed scheduling algorithms that are part of the architecture ensure that applications deliver efficient performance in the presence of network variations. We evaluate the performance of the proposed algorithms through simulations and compare it to a baseline approach that is network-unaware. The results show the performance benefits of the proposed algorithms over the network-unaware approach.

1.2 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we provide some background on distributed computing at the network edge and resource management approaches in distributed systems. We develop a taxonomy for resource management at the network edge. Use cases and application scenarios as well as challenges in resource management for IoT applications are also given in Chapter 2.

An overview of existing literature and works on task allocation and scheduling in distributed systems is given in Chapter 3. In Chapter 4, we introduce algorithms for resource and task allocation for independent tasks of an IoT application in the cloud-edge continuum. We propose a two-phase approach for task allocation. The evaluation of the proposed algorithm is carried out through extensive simulations and compared with existing schemes. A detailed analysis of the results is given.

Chapter 5 introduces a two-stage approach that considers the scheduling of real-time application tasks. We evaluate the performance of our algorithms by implementing the two-stage scheduler in a fully working middle-ware - JAMScript; an open-source programming language for edge computing and present the experimental results. Chapter 6 presents a network-aware scheduling scheme for stateful applications. We propose algorithms that take the network variation into consideration. The algorithms are evaluated using extensive simulations and the detailed analysis of the results is likewise given. A summary of the thesis and possible future extensions is given in Chapter 7.

1.3 Co-author Contribution

Major portions of this thesis are published in the following two publications [36, 37]. In [37], Olamilekan Fadahunsi played the key role of devising the main conceptual ideas, designing and evaluating the algorithms. Yuxiang Ma provided help with implementing the algorithms.

Chapter 2

Background

Current exuberance about the Internet of Things (IoT) is driven by both the advances and widespread adoption of technologies such as cloud computing [38, 39, 40, 41]. In conventional cloud computing, data processing and storage typically occur within the boundaries of a cloud and its underlying infrastructure. Cloud computing is not designed to cater to the scale of geographically dispersed and low latency required for many IoT use cases. As such, edge and fog computing have been proposed to cater for the scale of data processing and storage needed to support the requirements of the IoT to function in a distributed and coordinated way at minimum latency [42, 43, 10].

2.1 Edge Computing

Edge computing is a computing model that makes use of resources at the edge of the network [44, 45]. It enables the processing of the data at the edge of the network by bringing the computation facilities closer to the source of the data [46, 47]. Edge network consists of edge devices (e.g. mobile phone, smart vehicles, smart objects etc.) and edge servers (e.g., border routers, base stations, wireless access points etc.) and these components can be equipped with the necessary capabilities for supporting edge computation [48, 8]. Edge computing ensures that the localized computations are performed at the edge and it provides faster responses to the computational service requests [49, 50, 51].

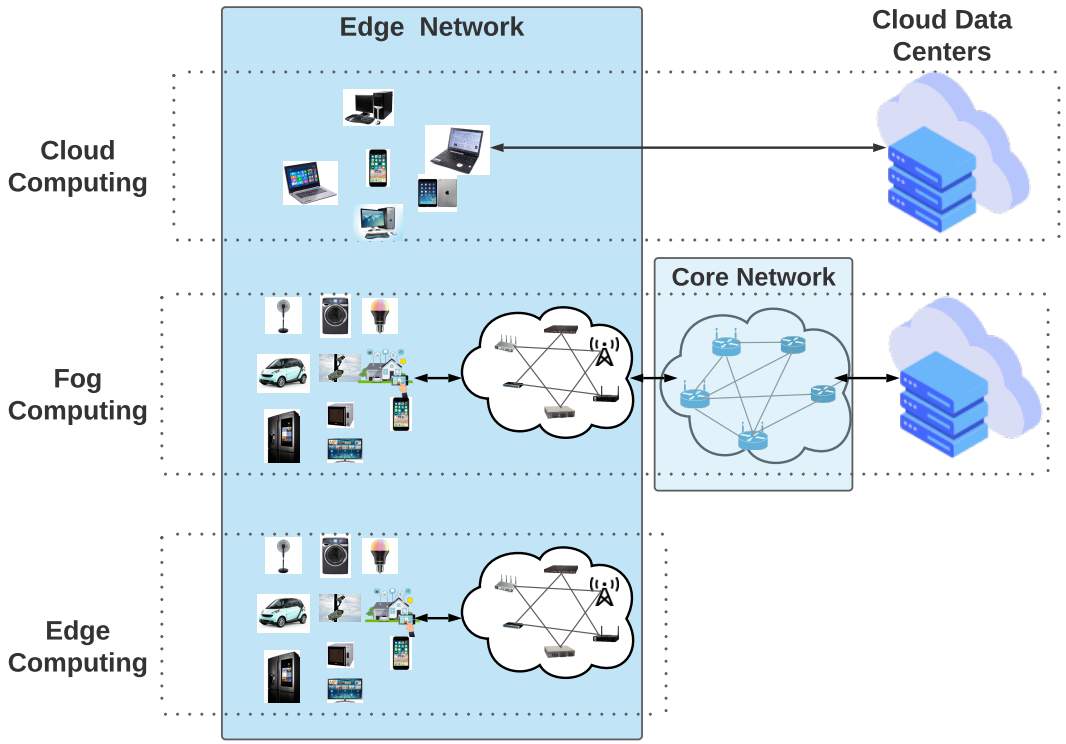

Figure 2.1 Cloud, Fog and Edge computational domains (based on Mahmud et al. [1]

2.2 Fog Computing

Like edge computing, fog computing can also enable edge computation. However, fog computing is a model that makes use of both the resources at the edge and the cloud as seen in Fig. 2.1 i.e., it can be extended to the core of the network. It can be described as a distributed computing paradigm that provides cloud-like services to the network edge [52, 53, 54, 55, 56, 57]. Fog computing according to the National Institute of Standards and Technology (NIST) [42] is defined as *"... a horizontal, physical or virtual resource paradigm that resides between smart end-devices and traditional cloud or data centers. This paradigm supports virtually-isolated, latency-sensitive applications by providing ubiquitous, scalable, layered, federated and distributed computing, storage and network connectivity"*.

Fog computing extends the cloud computing paradigm to the edge of the network, thereby enabling a new breed of services and applications. While cloud computing is an efficient alternative to owning and managing private data centers for customers deploying web applications and batch processing [58, 59], it is inefficient for latency-sensitive applications that are prevalent in IoTs which require resources (or nodes) in the vicinity to meet their delay requirements. IoTs require mobility support and geo-distribution in addition to low latency and location awareness. Fog computing was conceptualized as a complement to cloud computing to meet these needs. It is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional cloud computing data centers. Fog computing can better meet the requirements of IoT applications compared to the sole use of cloud computing.

The defining characteristics of fog computing include but are not limited to the following [52, 60]:

- Low latency: The proximity of fog nodes and resources to end-users makes the support of real-time services (e.g., video streaming, gaming, augmented reality) possible.
- Geographical and large-scale distribution: Fog provides storage resources and distributed computing to large distributed sensor networks and applications. In contrast to the centralized cloud, the fog will play an active role in delivering high-quality streaming to moving smart vehicles through proxies and access points that are deployed as roadside units.
- Mobility and Location awareness: Rich services are delivered to moving users and location-constrained devices.
- Flexibility and heterogeneity: It allows the interaction of different physical environments and infrastructures among multiple services. Fog nodes will be in different form factors and can be deployed in a variety of environments.
- Scalability: The closeness of the fog enables scaling the number of connected devices and services.

These defining characteristics of fog computing make it the perfect solution for a broad range of applications such as Smart Home, Smart Connected Vehicles, Health Care Systems and Big Data Analytics [61]. However, the explosive prevalence of big data, IoT and fog computing in the context of cloud computing makes it very challenging to investigate the efficiency of resource utilization and satisfy the user's quality of service requirement.

2.2.1 Fog-Enabled IoT Applications

Device ubiquity is one of the main factors that brought about fog computing. Currently, there are 25 billion connected devices in the world and the number of IoT devices is expected to grow rapidly in the coming years [62]. These devices are made up of both user mobile devices and sensing devices, which have been termed Internet of Things (IoT) devices. With so many devices, there would be an increase in network traffic, fog will help reduce the network traffic by localizing some processing and reducing the need of engaging the cloud, thereby ameliorating the likely bandwidth problems. As shown in the architecture in Fig. 2.2, the fog will play a role in delivering quality streaming to mobile nodes, like moving vehicles, through access points positioned accordingly, like along highways. Overall, for smart communications, fogs are going to play an important role as it suits applications with low latency requirements, emergency and health-care related services, video streaming, gaming etc. As shown in Fig. 2.3, many IoT applications, such as healthcare, smart home, smart grid, autonomous vehicles can be enabled by fog computing. We present some specific typical applications as follows.

Healthcare

Data management in health related issues is a sensitive topic since the health data contains valuable and important private information [63]. With fog computing, the patient can have custody of their health data locally [64]. These data would be stored in a fog node such as a smart vehicle or a smartphone. The computation will then be outsourced in a privacy-preserving manner when the patient is seeking help from a physician's office or a medical laboratory. The modification of the health data happens directly on the patient-owned fog node [65]. In [66], the authors

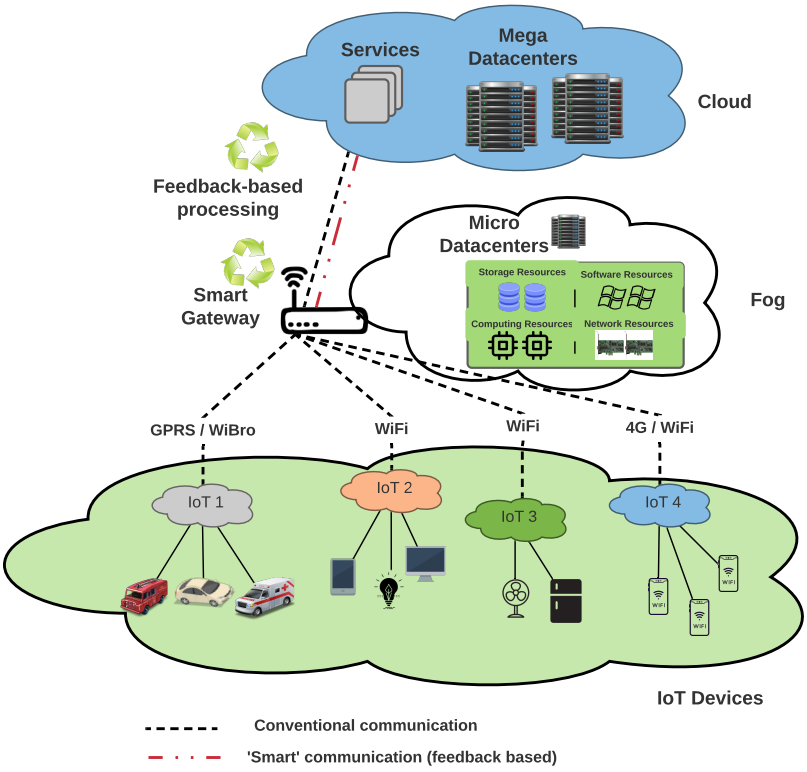

Figure 2.2 Smart gateway with Fog computing [2]

proposed FAST which is a fog computing assisted distributed analytics system to monitor fall for stroke patients. They developed a set of fall detection algorithms that were based on acceleration measurements, time series analysis methods and filtering techniques to enhance the fall detection process. The authors designed a real-time fall detection system that is based on fog computing, which divides the detection task between the edge devices and the cloud. The system achieved a high sensitivity when tested against real world data and the response time and energy consumption were close to the most efficient existing approaches.

Smart Home

With the development of Internet of Things, more and more sensors and smart devices are connected at home [67]. This would require different devices from separate vendors to work together (which is a hard task) and it would also require

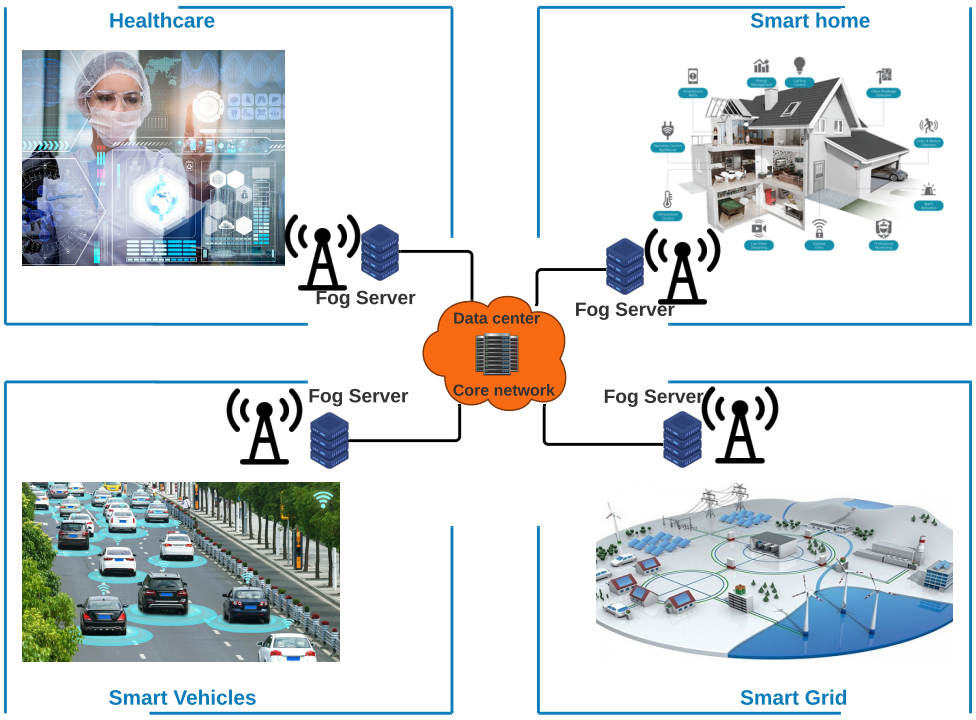

Figure 2.3 Applications of Fog-enabled IoT.

a large amount of computation and storage. For example, real-time video analytics is infeasible because of the limited capability of the hardware. To solve these problems, fog computing is used to integrate all entities into a single platform and empower those smart home applications with elastic resources [65]. For example, given a home security application, widely deployed secure sensors consist of smart locks, various sensor monitors, video/audio recorder etc. If they are not products of the same vendors, those secure devices are hard to combine. Fog computing can provide home security applications with the following:

- a unified interface to integrate all kinds of independent applications
- flexible resources to support computation and storage
- real-time processing and low-latency response

Once the fog platform is set up, each secure sensor can be connected as a client and the corresponding server application can be installed on independent virtual

machines (VMs). Advanced processing logic can also be implemented on VMs, which can process the data shared by those secure monitor applications. Fog computing plays a key role in achieving localized services and low latency for smart homes.

Smart Grid

A smart grid is an electricity distribution network. Smart meters are deployed at various locations to measure real-time information. A Supervisory control and data acquisition (SCADA) server is used to stabilize the power grid in case of fluctuations, emergency, or demand changes. It does this by gathering and analyzing status information and issuing commands in response to any demand change or emergency. Fog computing can be used to benefit SCADA immensely by introducing a decentralized model with micro-grids. This would not only improve the scalability, cost efficiency, security, and rapid response of the power system but also integrate distributed power generators (wind farms, solar panels, etc.) with the main power grid. Using fog computing, the smart grid will be a hierarchical system with the interplay between fog and SCADA [68]. In this scenario, the fog is in charge of a micro-grid and communicates with neighboring fogs and higher tiers. The higher the tier, the wider the geographical coverage and the larger the latency as well. SCADA provides the final global coverage, which is responsible for economic analytics and long-time repository.

Smart Vehicles

The smart vehicle deployment displays a rich scenario of connectivity and interactions: cars to access points (Wi-Fi, roadside units, 3G, smart traffic lights), cars to cars and access points to access points. Fog computing has many attributes that make it a suitable platform for delivering a rich menu of services in infotainment, traffic support, safety and analytics: geo-distribution (throughout cities and along roads), location awareness and mobility, heterogeneity, low latency and support for real-time interactions.

Popular applications of vehicular fog computing include traffic light scheduling, parking facility management, congestion mitigation, precaution sharing, traffic in-

formation sharing. Fog computing can be integrated into vehicular networks, and it can be classified into two types depending on whether extra infrastructure is needed. The types are infrastructure-based [69] and autonomous-based [70]. In infrastructure-based, driving-by vehicles rely on the fog nodes deployed on the roadside to send and retrieve information while in autonomous-based, the vehicles can be used to form a fog resource and support ad-hoc events. New applications such as virtual reality, self-driving etc. deal with complex data processing and storing applications. Thus, they require a higher level of communication, computation, and storage.

2.3 Differences between Edge Computing and Fog Computing

Both edge computing and fog computing share a lot of similarities. They are both enablers of data traffic to the cloud. While edge computing carries out processing where the data is being generated i.e., at the edge of a given application network, the traffic of data being sent to the cloud can be massive and irrelevant data can be sent to the cloud. Fog computing can act as a layer between the cloud and the edge such that instead of the edge sending large streams of data directly to the cloud, fog computing can filter the data from the edge layer and decide what is relevant or not. With fog computing, the relevant data gets stored in the cloud while irrelevant data can be analyzed at the fog layer to inform localized learning models, or it can be deleted. Fig. 2.4 shows the edge, fog and cloud layers of a computing infrastructure. The rationale behind edge computing is that computing should happen at the proximity of data sources. Edge computing focuses on the things side while fog computing focuses on the infrastructure side [71].

2.4 Resource Management

Due to the heterogeneity and limitations of the fog resources, the dynamic nature and unpredictability of the fog environment along with the application deployment in both fogs and physical servers in the clouds, resource management becomes one of the challenging problems to be considered in the fog landscape. We catego-

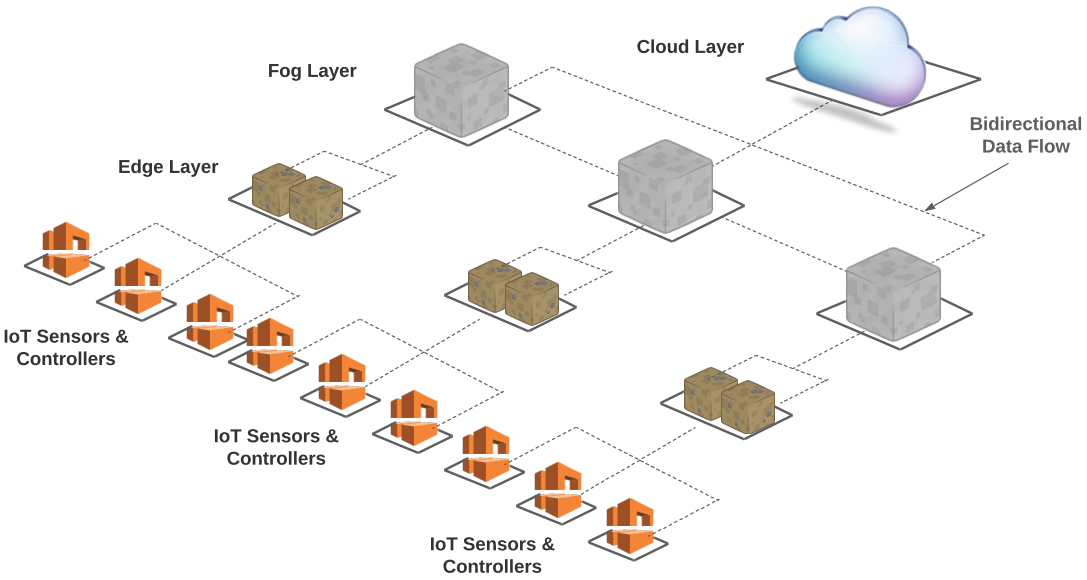

Figure 2.4 Example showing how IoT devices, edge, fog and cloud layers of a computing infrastructure connect.

rize the resource management problem in fog computing using the following approaches: load balancing, resource allocation, service scheduling, task offloading, and resource provisioning.

2.4.1 Load Balancing

Load balancing is the process of redistributing workload among fog nodes to improve both resource utilization and job response time. It avoids situations where some fog nodes are heavily loaded while others are idle or doing little data processing. In such scenarios, load balancing between fog nodes helps minimize user response time and detect events in real-time. Load balancing in distributed environments such as fog computing is divided into two main approaches: static and dynamic load balancing [72]. In static allocation, a set of tasks is provided to specific fog nodes so that the performance function is minimized. The allocation is done using either probabilistic or deterministic means. In a probabilistic allocation technique, a fog node I allocates its overloaded tasks with some probability to some backup fogs. E.g., it allocates the overloaded tasks to fog node K with

probability x and to fog node L with probability y. In a deterministic allocation technique, a fog node I allocates its overloaded tasks to a backup fog node J all the time. The major disadvantage of static allocation is that the destination fog node resource status is not considered when making the allocation decision. In dynamic load balancing, the current status of the fog nodes is taken into consideration when making the allocation decision. As a result, tasks are allocated dynamically from an overloaded fog node to an underloaded fog node. Although it is more challenging, it gives a more efficient solution to achieve a sustainable load balancing/allocation. We consider the dynamic allocation strategy in our proposed solution.

2.4.2 Resource Allocation

Resource allocation deals with allocating a set of geographically distributed heterogeneous fog nodes to competing IoT services/users with different Quality of Service (QoS) requirements while considering fairness and service priority. In the IoT era, the demand for low-latency computing services for time-sensitive applications (e.g., mobile augmented reality applications, real-time navigation using wearables, real-time smart traffic management) has been growing rapidly [73, 74, 75]. Fog computing provides a suitable infrastructure to fill the latency gaps between the IoT devices and the core network infrastructure (i.e., back-end computing infrastructure). Ensuring that resources are allocated fairly while considering the applications' QoS requirements is a challenging task.

2.4.3 Resource and Service Scheduling

Service requests from IoT devices can be served by several fog nodes. An application request can be divided into a set of tasks. Fog computing scheduling problem determines an optimal assignment of the tasks submitted to the fogs to meet the agreed quality of service with the IoT end-user while minimizing the execution time for the submitted task [76, 77]. A scheduling algorithm searches for an optimal solution in an enormous search space that schedules a set of tasks with various QoS requirements (cost, deadline etc.) onto a set of fog nodes with different capabilities for optimizing the scheduling objective function (i.e., minimizing the execution

time). In a fog computing environment, the objective of resource scheduling is to assign appropriate resources for submitted tasks according to scheduling goals.

2.4.4 Task Offloading

IoT devices have limited computational power, storage space, and battery energy and are therefore constrained to run compute-intensive tasks. As a result, some of their tasks need to be outsourced to the cloud or fog to improve their performance and save battery. The process of executing tasks on behalf of IoT devices on the fog or cloud and returning the results to the devices is called task offloading [78, 79, 80]. It is a mechanism by which compute-intensive tasks are transferred from resource-constrained IoT devices to resource-rich computing nodes to improve the performance of delay-sensitive IoT applications. It should be noted that tasks are not bound to computation only but also to other resources such as storage [81].

2.4.5 Resource Provisioning

IoT device application requests (i.e., services workload) change over time thereby leading to workload fluctuations [82, 83]. The fluctuating workload can lead to either under-provisioning or over-provisioning problems. Under-provisioning occurs when the allocated fog resources are less than the actual loads of the IoT user demand, leading to Service Level Agreement (SLA) violations and loss of IoT users. On the other hand, over-provisioning occurs when the allocated resources for a given IoT service are more than the actual loads of the IoT user demand. It is important to dynamically provide the appropriate number of resources to handle the IoT workload for minimizing costs while meeting the QoS constraints.

2.5 Taxonomy and Use cases

2.5.1 Taxonomy

The fog computing platform can provide many services to support IoT-based applications. It usually consists of multiple fog nodes that are distributed across a city to enable IoT applications such as real-time traffic control systems [84, 85]

that require low latency responses to avoid potential collisions, virtual and augmented reality applications [86, 87, 88, 89], real-time smart grid management systems [90, 91, 92, 93] that aggregates data from geo-distributed sensors and control the grid in real-time.

The platform allows services to be executed geographically close to the IoT-based applications, thereby providing low latency, location awareness, mobility, and real-time support. The fog services are categorized as follows:

- *Stateless Services Vs Stateful Services:* The fog services that are provided depend on the IoT application. A stateful fog maintains state data that is updated with each service invocation, and the operations and outcomes of the service invocation can be affected by the current state data [94]. Examples of such stateful fog services include a service that coordinates and synchronizes local IoT devices, a service that provides aggregate and/or average measurements of sensors within a defined period. In both examples, the fog services need to keep information that propagates from one invocation to the next and cannot operate correctly without it. This type of service requires that all current state data be replicated and must be updated continuously with any changes in the state data, making it challenging to accurately maintain at all times.

  In contrast, a stateless fog service performs a specific task invocation without requiring or updating state data. Both its operation and outcomes are not affected by the state data. Examples of such services include a computation service that performs a specific calculation on a provided data, a service to retrieve the current measurement of a sensor etc. Generally, a stateless service can be easily replicated on another fog node as it does not maintain any state data.

- *Non Real-Time Services Vs Real-Time Services:* Real-time fog services are services that need to be completed within a time frame [95, 96]. The responses and actions are useless if they are not completed within the specified time. As a result, the operations of the application using the service may be negatively affected if the actions and responses are not delivered within the specified time. Fog computing is better equipped than cloud computing

to support real-time services as fog nodes are better and closer placed to the requesting applications. Fog computing can also support non real-time services that do not have time constraints to perform their tasks. Fault tolerance handling with non real-time services is easier compared to real-time services. This is mainly due to the time constraints as non real-time services can be offloaded to other fog nodes or the cloud with minimal impact. However, with real-time services relocation, a suitable fog that meets the required time constraints must be identified. Therefore, it is important to identify suitable backup fog nodes that are close enough and have the required capabilities to perform the service.

- *Mobile Vs Non-Mobile services:* Mobile services are services that require mobility. Either the IoT devices are running applications while being mobile or the fog nodes are allowed to be transported in a specified region [97, 98]. We focus on mobile IoT devices such as smart vehicles. Non mobile services encompass services that are fixed, i.e., the devices are stationary when requesting for a fog service. Examples of mobile services include smart cars, automotive technologies, while non-mobile services include smart buildings, shopping malls etc.

The resource management architecture for fog computing is classified using the fog services identified. There are eight types of services for IoT applications as shown in Table 2.1. They exhibit different levels of difficulty in handling fault tolerance, task allocation, and scheduling. For example, stateless, non real-time and non-mobile services are the easiest as there are limited constraints and replication can be done easily and stateful, real-time services and mobile are the most complex as they require multiple constraints and continuous monitoring for replication.

2.5.2 Use Cases

We discuss various application use cases that fall into our stated taxonomy.

A *Stateless, non real-time and non-mobile services*
IoT applications such as smart building controls fall into this category [99,

Table 2.1 Combination of different types of services

Number	Service Type
1	Stateless, non real-time and non-mobile services
2	Stateless, non real-time and mobile services
3	Stateless, real-time and non-mobile services
4	Stateless, real-time and mobile services
5	Stateful, non real-time and non-mobile services
6	Stateful, non real-time and mobile services
7	Stateful, real-time and non-mobile services
8	Stateful, real-time and mobile services

100, 101, 102]. Sensing applications such as temperature and humidity reading of a room do not need stateful data and are not time-constrained.

B *Stateless, non real-time and mobile services*
Applications that fall into this category include smart mall shopping [103, 104, 105]. Fog services are provided on every floor of a mall and as customers move about, they request different information from that section of the mall. General purpose tasks where there is no hard deadline such as image recognition also falls in this category.

C *Stateless, real-time and non-mobile services*
Content retrieval such as file transfer and video streaming are stateless since the client-server is agnostic of the client device session [106]. The extremely low response times enabled by the fog can enhance the user experience.

D *Stateless, real-time and mobile services*
A moving car requesting weather readings or pothole information from a roadside unit fall into this category [107].

E *Stateful, non real-time and non-mobile services*
Data gathering and processing on millions of devices require stateful fog services [108]. The fog can handle and help process data on devices where the data for the device and computation are co-located.

F *Stateful, non real-time and mobile services*
Data analytics and workflow processing applications such as those available

in financial and transitional systems fall under this category [109]. The fog service can support application processing pipeline where results must be reliable and passed on to the next processing stage without any loss.

G *Stateful, real-time and non-mobile services*
Gaming and session-based interactive applications require low latency read and writes [110]. The fog service supports these interactive stateful applications without having to create a separate store or cache at the client device.

H *Stateful, real-time and mobile services*
Applications for autonomous driving fall in this category [111]. Traffic monitoring applications used in smart autonomous vehicles allow the user to avoid potential dangers and traffic scenarios in real-time.

2.6 Summary and Discussion

In this thesis, we explore a variety of algorithms that focuses on the different resource management approaches for the identified fog services. We develop resource and task allocation algorithms that help minimize the average response times of stateless applications. We propose scheduling algorithms for real-time services that help meet the application deadlines, and we consider mobile services by introducing network-aware algorithms to minimize the stateful application execution times.

Chapter 3

Related Work

In this chapter, we discuss the literature on task allocation and scheduling algorithms related to our work. Given that IoT applications are latency-sensitive and have both real-time and non real-time constraints, a proper scheduling framework that employs both edge and cloud resources is needed to achieve the objectives of IoT applications. We classify these objectives into two parts, which are termed *fast computing* and *real-time computing*. In the first part - fast computing, we address algorithms that aim to provide low latency by minimizing the average response time of a set of tasks in the application and then in the second part - real-time computing, we describe solutions that aim to meet the individual timing requirements of each of the tasks in the application [112]. While the works discussed in the first two parts assume that the computing resources are always available at a very short network link away from the end device with high capacity to transfer data, the works presented in the network-aware scheduling section show some scenarios where the network link variation can have an impact on the application performance.

3.1 Fast Computing

In fast computing, we examine task allocation algorithms in the context of fog computing that aim to minimize the average response time of the application.

Hou et al. [113] proposed a combined system with an edge-cloud and a backend

cloud. Tasks served by the edge-cloud incur smaller overall costs compared to the ones that eventually get served by the backend servers because those tasks incur processing costs at the edge and communication costs over the backhaul connections that connect the edge to the backend. If a service is popular, it should be downloaded to the edge-cloud at a cost. This will allow subsequent tasks to get served at lower costs. The objective [113] is to minimize the total costs of the system by intelligently reconfiguring the set of services around the edge-cloud.

Oueis et al. [114] dealt with load balancing in a small cell cloud computing system while simultaneously optimizing the power consumption costs and latency constraints. The user tasks are associated with small cells and small cell clusters. Small cell clusters are composed of small cells and are formed for tasks that cannot be served by a small cell. In forming the cluster, several strategies were employed such as minimizing the cluster latency and power consumption costs. The authors formulated a clustering problem for multiple users as a joint optimization of communication and computational resources with the objective of minimizing the overall power consumption in the cluster. The heuristics proposed by the authors join task scheduling and cluster formation. The goal is to allocate resources to satisfy as many user tasks as possible while keeping both the power consumption and processing complexity low.

Song et al. [115] presented an approach to increase the number of tasks that can be processed at the edge. They formulated a task distribution problem that includes satisfying the quality of service (QoS) requirements of application tasks amid constraints stipulated by the edge resources. They solved the task distribution problem using an algorithm that relies on relaxation, rounding, and validation. They evaluated their work against two baseline approaches, which are a random approach that routes tasks randomly to any edge node and a local approach that executes the task locally without distributing it to the edge computing network. The main contribution of this work focuses on increasing the number of tasks served, with metrics such as response times added as constraints in the problem formulation.

Dang and Hoang [116] developed fog-based regions to provide nearby computing services and proposed a scheduling algorithm that distributes tasks to multiple regions and the cloud when resources are needed. The aim is to minimize the completion time of tasks and improve users' experience. The author's concept of "region" addresses the latency requirements of applications in fog computing and in collaboration with the cloud to provision resources on demand. In their work, tasks are sorted in the order of latency constraints and resources are allocated according to the policy that aims to minimize the computation latency for each task. They evaluated the proposed scheduling policy against two approaches, namely, "Cloud-based" that schedules all tasks on the cloud and "Region-based" that schedules all tasks on the fogs until they are fully loaded.

Yousefpour et al. [117] proposed a delay-minimizing allocation algorithm for IoT applications. When a device sends a task to a fog, the fog will serve the task if its waiting time is less than a specified threshold; otherwise, the fog offloads the task to the 'best' fog in its domain. When offloading the task, fogs use a reliability table to decide the task processing. The reliability table shows, for each fog in the domain, the sum of the propagation and processing delay in ascending order. The fog with the lowest delay is selected as the best fog to offload the task to. If a selected fog's waiting time is greater than a threshold, the next best fog is selected in the domain and the task is offloaded to it. The process is repeated until either a fog's waiting time is less than the threshold and it serves the task, or it is offloaded to the cloud after a limit (e_m) has been reached, or all fogs have been visited (N_{fwd}).

Skarlat et al. [118] modeled a task placement problem for IoT applications over fog resources as an optimization problem. They considered the heterogeneity of the applications and resources in terms of quality of service attributes. They solved the optimization problem using the IBM CPLEX library [1] and also proposed a genetic algorithm (GA) heuristic to show that the task execution achieves a reduction of network communication delays when the GA is employed and a better utilization of fog resources when the exact optimization method is applied.

[1] https://www.ibm.com/analytics/cplex-optimizer.

Table 3.1 Comparison of related task allocation algorithms.

	MinDelay [117]	FRBC [116]	QoS-Approach [115]	Fog Balancing [114]	VFR (Proposed algorithm)
QoS / Metric considered	Minimize response time	Minimize response time	Maximize number of tasks completed	Optimize power consumption	Minimize response time
Load Balancing	Yes	Yes	No	Yes	Yes
Cloud Involved	Yes	Yes	No	No	Yes
Mode of Interaction	Decentralized	Decentralized	Centralized	Centralized/Distributed	Decentralized
Regions/Clusters	Yes, formed dynamically	Yes, formed dynamically	No	Yes	No
Location Sensitive	Yes	Yes	Yes	No	Yes
Service Offered	Task allocation	Task scheduling	Task allocation	Task scheduling	Task allocation & scheduling

Table 3.1 gives an overall comparison of the related studies as it relates to fast computing and our proposed algorithm which will be discussed in Chapter 4. Our algorithm does not use clusters or regions. At least for task offloading among the fogs, our algorithm does not expect any tuning parameters. The tuning parameters are involved in forwarding the tasks to the cloud when fogs are deemed underperforming. The proposed algorithm does not do explicit load balancing among the fog nodes. It spreads the workload when it detects bad performance at the designated fog node.

While the goal of Hou's work [113] is to minimize cost, Oueis [114] aim to maximize resource allocation while keeping power and processing complexity low. Song [115] also focused on maximization on the number of tasks that can be served by the fogs. Dang's [116] objective was to offload the tasks appropriately to fog and cloud resources while minimizing response times. Yousefpour [117] focused on minimizing response times when tasks are injected to the fog and cloud resources. Skarlat [118] aimed at reducing the communication delays incurred by processing tasks and offers better utilization of fog resources. Our goal is to minimize the response time it takes for tasks to get served while using as little fog resources as we can. Our approach, therefore, minimizes the cost of using a fog resource, the response time of serving tasks, and using the fog resources efficiently.

3.2 Real-time Computing

In real-time computing, we examine algorithms that aim to meet the individual timing requirements of the application tasks.

Isovic et al. [119] dealt with a combination of a mixed set of tasks and constraints. These include periodic tasks, firm aperiodic tasks and sporadic tasks. They proposed an offline scheduler to manage the start times and deadlines of periodic tasks using the Earliest Deadline First (EDF) model. The execution of offline scheduled tasks is flexibly shifted at run time to allow for the feasible inclusion of arriving sporadic and aperiodic tasks. The maximum frequency of sporadic tasks is assumed to be known, however, the exact knowledge of arrival is known at runtime. Therefore, using the maximum frequency assumption, sporadic tasks undergo an acceptance test and are guaranteed offline. Since resources are reserved for a pessimistic scenario of sporadic tasks offline, the online arrival of sporadic tasks is assumed to be less pessimistic and the extra freed-up resources are used to serve firm periodic tasks.

Tang et al. [120] proposed a method where slack windows are prescheduled in a non-preemptive system that consists of hard periodic and soft aperiodic tasks. The aim was to improve the acceptance rate and waiting time of aperiodic tasks without violating periodic task deadlines. The proposed method creates an offline schedule of periodic tasks and pre-scheduled slack, which models the estimated characteristics of the aperiodic tasks. The offline schedule is used at runtime by the local schedulers to guide their decisions to adjust start times of periodic task instances while scheduling aperiodic tasks arriving from the global scheduler.

Tang et al. [121] focused on scheduling soft aperiodic tasks alongside periodic tasks with hard deadlines. They proposed a method to improve the aperiodic task responsiveness without breaking the periodic task deadline guarantees. The method schedules periodic tasks offline while using the remaining slack time to dynamically schedule aperiodic tasks. The authors noted that the quality of aperiodic scheduling depends on two factors: a) the slack distribution across and

within resources after scheduling periodic tasks, and b) the flexibility of the scheduler to rearrange slacks to accommodate incoming aperiodic tasks.

Jeon et al. [122] proposed a method to reduce the response time of aperiodic tasks while meeting the deadline of periodic tasks. They developed a slack stealing algorithm that attempts to make time for servicing aperiodic tasks by stealing the processing time from the periodic tasks without causing the periodic tasks to miss their deadline. The authors claim that their method results in a significant reduction in the response time of aperiodic tasks.

A soft-real system was modeled for multimedia applications in a UNIX system by Chu et al. [123]. The system supports different service classes of tasks such as periodic and aperiodic constant processing, variable and one-time processing classes. The classification was based on the processor usage pattern of the real-time processes. The EDF algorithm was used for the scheduling of multiple real-time processes. The authors implemented the system in the user space without any modifications to the underlying kernel.

Lin et al. [124] developed a system for mixing batch and interactive virtual machines on physical machines subject to the latency and execution constraints for each workload. The system uses a periodic real-time scheduling model to schedule the virtual machines.

A system that supports a mix of interactive and batch application tasks in commodity operating systems was presented by Yang et al. [125]. The authors aim to maximize the responsiveness of interactive applications even in the presence of extreme workloads. An EDF-based algorithm was used to schedule interactive applications.

Tasks with hard real-time constraints were considered by Zhao et al. [126]. The authors developed a system that does not consider the prior and complete knowledge of the task set. Tasks arrive in a batch mode and are scheduled non-preemptively.

Chen et al. [127] addressed challenges related to communicating processes from different nodes sending data to one another. To realize end-to-end predictability, a challenge that was resolved by the authors is integrating real-time and non real-time tasks on the same platform.

Our system design given in Chapter 5 works with non-preemptive tasks. We also consider a system where a mix of hard real-time, soft real-time (i.e., interactive tasks), and batch tasks are scheduled. Some works [119, 126, 122, 124] deal with preemptive systems and those that deal with non-preemptive [120] do not take the three classes of tasks into consideration as we have done. Furthermore, some works considered scheduling tasks on a single node [126]. Those that considered distributed systems schedule the task exclusively on the distributed nodes [119, 124]. In this work, tasks can be scheduled in distributed systems in both exclusive and non-exclusive ways, i.e., tasks are allowed to run on several resources. Finally, we designed a two-stage system architecture where tasks are matched to resources in the first stage and then scheduled onto the matched resources in the second stage. Isovic et al. [119] employ a 2-phase approach but task matching does not occur. Instead, tasks are assumed to be assigned to the resources and a schedule is created offline in the offline phase for periodic tasks, while the aperiodic tasks are scheduled on top of the offline schedule at runtime in the online phase.

3.3 Network-Aware Scheduling

A handful of research efforts have been performed for network-aware scheduling in cloud environments, while very few have been undertaken in fog and edge computing environments.

LaCurts et al. [128] propose Choreo - a system that tenants (cloud service provider clients) can use to place a mix of applications on the cloud infrastructure by understanding the underlying cloud network as well as the application demands. The proposed system has three components a) a measurement component to obtain the network rates between the virtual machines (VMs) b) a component to

profile the data transfer characteristics of the application, and c) an algorithm to map the application tasks to VMs such that tasks that communicate often are placed on VMs with higher rates between them. Choreo's placement method aims to minimize the application completion time. The authors formulated the placement method as an integer linear program problem, however, they noticed it took a very long time to solve, thereby hampering the ability to place application tasks quickly. Alternatively, they proposed a greedy network-aware approach that places pairs of tasks that transfer the most data on the fastest paths.

Arslan et al. [129] proposed an algorithm for the reduce task scheduling component of a map-reduce application by taking both data locality and network traffic into consideration. The network-aware component of their work aims to distribute network traffic over the whole network to reduce the effects of congestion on data transfer. The authors combined data locality (by moving reduce tasks to where the map tasks are located) with network-traffic awareness to decrease the shuttle phase duration as well as lower the network traffic caused by data transfers. The authors consider the network bandwidth capacity and congestion when comparing a pair of potential paths for reduce tasks input data movement. The scheduling decisions are made by considering the impact of network congestion on the end-end performance of the application.

Conversely, He et al. [130] focused on map tasks of a map-reduce application and proposed Firebird - a derivative of spark as a network-aware scheduling method in Spark that runs on top of Software Defined Networks (SDN). The authors argue that data processing and network are separate entities that do not communicate with one another and therefore make it hard to create a relationship between the network and Spark. The introduction of SDN removes the communication barriers between the underlying network and the compute clusters, and it provides direct application programmable interfaces (APIs) for the applications to control and monitor the state of the network. This ensures that tasks can be scheduled based on the network conditions in Spark clusters. Their method aims to avoid network congestion and achieve efficient scheduling to increase the throughput of the whole system. Essentially, when a compute node requires a task, the scheduler selects

tasks that will not experience congestion during data shipping, which thereby accelerates the execution process.

Fiore et al. [131] proposed a data-centric meta-scheduling scheme for distributed big data processing architectures based on clustering techniques. The objective is to aggregate tasks around storage repositories, driven by a "gravitational" attraction between tasks and their data of interest. The authors stated that their proposed scheme benefits from a heuristic criterium based on network awareness and advance resource reservation to suppress long delays in data transfer operations and result in optimal use of data storage and runtime resources.

While most of the works discussed above are suited to cloud computing environments, the work done by Santos et al. [132] is suited to fog computing environments. The authors proposed a network-aware scheduling approach for container-based applications in Smart city deployments. They validated their approach on the Kubernetes platform, and it is implemented as an extension to the default scheduling mechanism available in Kubernetes. The proposed network-aware approach is presented to provide up-to-date information about the current network status to Kubernetes to enable it to make informed resource allocation decisions. They compared the performance of the approach with the available standard scheduling features in Kubernetes, and they state that their approach achieves a network latency reduction of 80% when compared to the Kubernetes default scheduling mechanism.

Although existing and ongoing research cited address network-aware scheduling in cloud and fog computing environments, they have not considered mobility as well as variations in wireless network conditions such as 5G. Our proposed network-aware scheduling approach discussed in Chapter 6 focuses on applications that run in a mobile environment with varying network conditions while the edge device is on the move. We consider the implications of dynamically changing network conditions on the execution of the application.

Chapter 4

Task Allocation Algorithms for Fog and Cloud Servers

In this chapter, we introduce solutions for resource and task allocation for independent batch tasks of an IoT application in a distributed cloud computing system. The distributed cloud configuration complicates the server selection problem for a given task. Therefore, we deal with the server selection problem by developing a two-phase fog (distributed cloud server at the edge) selector.

In the first phase, the selector allocates a set of fogs to a device in a tiered configuration: a primary and pool. In the second phase, the tasks are allocated to the nodes, including the primary and pool. One of the unique aspects of the algorithm is its probing design that adapts according to the network latencies that connect the device to the fogs and cloud. As the network or network plus server latencies changes with time, the fog selector changes the fog that is chosen to serve a task. The two-phase procedure allows us to control the dispersion of the tasks because the task routing function is confined to the fogs that are selected by the allocation algorithm.

The described framework allows us to provide a suitable fog resource management approach, i.e., a solution for the distribution of IoT tasks among fog nodes which aims at minimizing the delays arising from the transfer times between the

fog and cloud and efficiency of resource usages of the fog nodes. To evaluate our approach, we compare different scenarios and task allocation policies. We aim to identify the best allocation policy by comparing suitable metrics, e.g., response time and resource usage.

4.1 System Architecture

The overall system architecture is shown in Fig. 4.1. The fog resource management system has components running in the devices, fogs, and cloud. We have the Fog Allocator running in the cloud with a global perspective. It is responsible for running the fog allocation optimizer given in Sect. 4.2. Another important component of the fog resource management system is the Fog Resolver. This component runs in each device. It is responsible for forwarding the task to different fogs. The Fog Allocator sets up the Fog Resolver periodically or at triggers generated by the Fog Resolver, so that the Fog Resolver can make decisions according to the state of the system.

The Fog Resolver maintains a routing table as shown in Fig. 4.1 via device a. In this table, homeFog (the default fog to use) and the pool (alternate or backup fogs) are set up by the Fog Allocator. Collectively, the homeFog and the pool are known as the virtualFog. It is the job of the Fog Resolver to resolve the virtualFog to a physical fog (i.e., the homeFog or one of the fogs in the pool). The pool concept is introduced to replicate services in it for fault tolerance purposes. It is similar to what is obtained when distributing databases in a cloud framework such as Openstack and Rackspace. The use of backups is generally proposed to meet the high availability constraint that is mandatory in production infrastructures [133]. To provide consistency, when a task is processed by an instance of a fog, changes to the service is propagated to the associated pool where the data are replicated. This ensures that the data which the devices interact with at the fogs are updated.

One of the unique aspects of our allocation system is the ability to consider data linking among the devices. The data linking (also referred to as the linked data) comes from one device generating data that another device wants to use (i.e.,

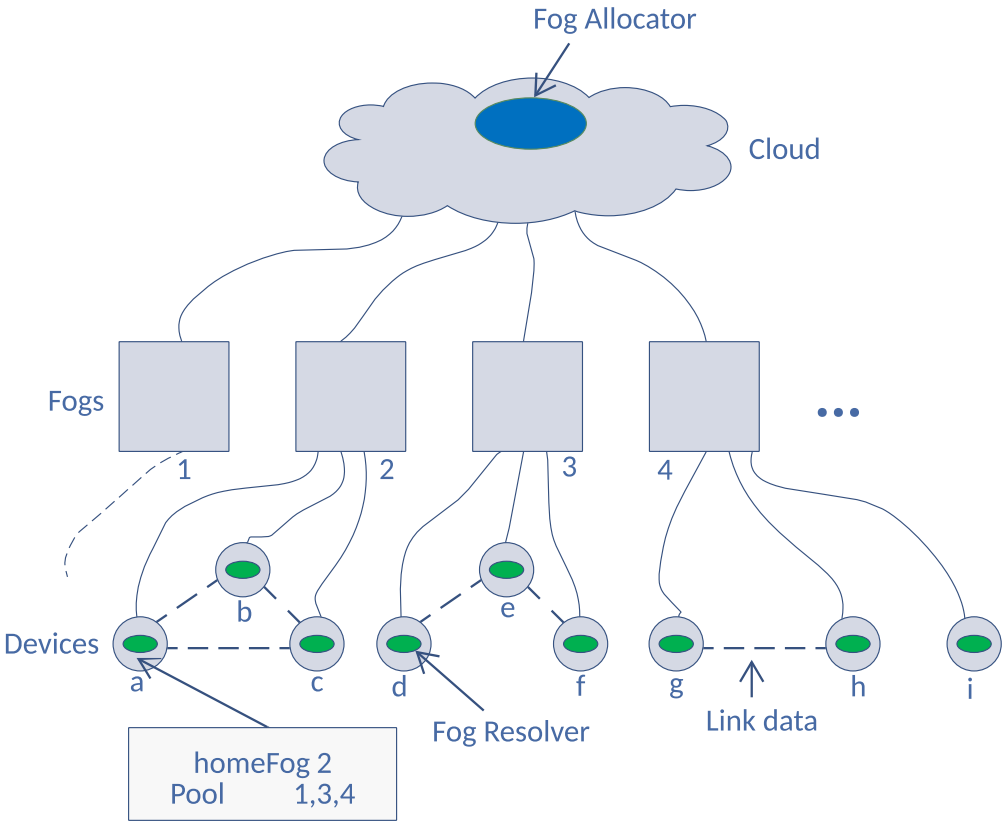

Figure 4.1 Overall Fog Resource Management Architecture.

data dependency) or the fog making a collective decision as a result of the linked data processing. More than two devices could be involved in a data dependency. By placing all such devices in a fog (assuming the fog capacity and proximity constraints are satisfied), we could get the data transmitted faster from one device to another or have data aggregation occur at the fog for faster processing. For example, a sprinkler actuator device and a fire sensor need to be connected to the same fogs so that they can interchange the data very quickly. We could have multiple redundant interchanging fogs to provide fault tolerance.

We describe a traffic management application use case that shows the feasibility of our architectural design. Traffic congestion is a severe problem that has the potential to paralyze major cities, choking off growth and development. Some cities resolve to drastic and expensive solutions to mitigate the problem, such as

expanding toll roads or restricting the number of license plates issued. Our architecture can provide a source of solution to help combat traffic-related problems. By leveraging traffic related data, congestion can be alleviated by connecting and analyzing previously unconnected infrastructure devices, on-board vehicles, and roadside sensors, to redirect traffic based on real-time data. We consider two aspects of the described use for which our architecture applies.

1. A device can ask a fog to perform some computational processing while sending the data for the function to be performed.

2. Device(s) can receive updates or tasks to be carried out from the fog.

In case 1, cars can ask a fog for the best route to take while supplying their source and destination information. Similarly, they can ask for infotainment services or a recommendation of a nearby service based on their location input.

In case 2, fogs situated on a roadside can collect data from all kinds of sensors and devices installed along a roadway. The fog can perform local analysis and trigger an automated emergency response such as "Flooded roads ahead: take the next exit" or "Water on road: slow down". Sensors and local devices can send their data to the same fog for faster analysis and enable city officials to respond to the emergency needs on time.

For the second case, devices are grouped such that their data is being fed into the same fog or nearby fogs for processing. Data from vehicles in the same area can be aggregated to quickly clear the traffic congestion or road flooding. Using vehicles' data, dynamic routing scenarios can be planned in real-time or planned in advance using historical data. Congestion can be anticipated and alleviated before it happens. Similarly, vehicles can connect to roadside sensors, aggregating their data to provide a larger view of traffic and roadside conditions. This is done to avoid traffic-causing accidents and makes the vehicle safer.

In Fig. 4.1, we have linked data between some devices and not others. For example, devices a, b, and c are interconnected by linked data and want to be in

the same fog, while device i does not have any linked data. The Fog Allocator considers all factors while allocating the virtualFog. In the example shown, for device *a*, we have homeFog *fog 2* and pool *fogs 1, 3, 4*. Because the network latencies and fog state can change with time, the Fog Allocator needs to recompute the allocation after some time to account for the changes, including device mobility. This recomputation is carried out when the Fog Resolver in the device notices that the assigned homeFog is not suitable for the device because the pool or the cloud is performing better.

In our architecture, the Fog Resolver is responsible for allocating the tasks to the fogs and monitoring the response times of the fogs. If the fogs are not yielding acceptable response times, the Fog Resolver can request the Fog Allocator for a remapping of the virtualFog.

4.2 Allocation Optimization Problem

The fog allocation problem is seeking to allocate multiple fogs for each device. One of the fogs (possibly the best candidate) is denoted as the *primary* fog (termed homeFog in the system architecture section) and remaining fogs that cover (or are assigned) a device are called the *backup* fogs (termed pool in the system architecture). This allocation problem is quite different from the widely solved facility-location problem where a single server is allocated to satisfy the demands originating from a given device [134]. The problem we have here can be solved using a related idea called the *Q-coverage* problem [135]. In Q-coverage, we assume that a device is going to be covered by a homeFog and Q-1 backup fogs. The objective is to find fog locations such that the total latency between all devices and their nearest fogs is minimized. We now formulate the model with the Q-coverage requirement.

- *Sets of Indices:*
  $i \in I$: set of device locations
  $j \in J$: set of fog locations

- *Parameters:*
  m = number of fogs ($m = |J|$)
  n = number of devices ($n = |I|$)
  r_i = number of tasks issued by device i
  cap_j = capacity of fog j
  c_{ij} = latency between device i and fog j
  d_{ik} = link relation cost between devices i and k
  f_j = setup cost of fog j
  Q = minimum number of coverage (assignment) required

- *Decision Variables:*

$$y_j = \begin{cases} 1 & \text{if fog } j \text{ is available} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{if device } i \text{ is assigned to fog } j \\ 0 & \text{otherwise} \end{cases}$$

- *Objective function:*

$$min \sum_{i \in I} \sum_{j \in J} \sum_{k \in I} r_i c_{ij} d_{ik} x_{ij} + \sum_{j \in J} f_j y_j \tag{4.1}$$

- *Constraints:*

$$\sum_{j \in J} x_{ij} = Q, \qquad \forall i \in I \tag{4.2}$$

$$\sum_{i \in I} r_i x_{ij} \leq cap_j, \qquad \forall j \in J \tag{4.3}$$

$$x_{ij} \leq y_j, \qquad \forall i \in I, j \in J \tag{4.4}$$

$$y_i, x_{ij} \in \{0, 1\}, \qquad \forall i \in I, j \in J \tag{4.5}$$

The first term in the objective function (4.1) seeks to minimize the total weighted latency between devices and their nearest fogs while taking the link relation cost between devices into consideration. The link relation cost is introduced

to enable us to place devices that share similar data in the same fog or nearby fogs. The second term in the objective function aims to minimize the setup cost of the fogs. Constraint (4.2) ensures that all devices are covered by Q fogs. Constraint (4.3) ensures that the capacity of the fog is respected when processing the demands at the devices. Constraint (4.4) ensures that the fogs that are not activated cannot cover any device. Constraint (4.5) declares the variable types. The model is solved using IBM CPLEX[1] optimization solver and the solution is given as input to the Virtual Fog Resolver (VFR) algorithm that is described in the next section.

We describe a small example scenario to illustrate the benefits of linked data. Given 4 fogs and 6 devices and a fog setup cost of $10, with the fogs having a capacity of 3 tasks at any given time and each device issuing a single task. The device to fog latency is given in Table 4.1. Using CPLEX to solve this scenario without linked data cost, the output is given in Table 4.3. For this scenario, each device is mapped to each fog such that device 1 is mapped to fog 1, device 2 to fog 2 and so on. With the linked cost introduced as shown in Table 4.2, device 1 shares data with device 6 and device 2 shares data with device 3. The solution shows that devices 1 and 6 should be mapped to fog 2 while devices 2 and 3 should be mapped to fog 3 as shown in Table 4.4.

This solution shows us the benefit of having devices sharing similar data being grouped in the same fog. The mapping obtained from this algorithm is passed into the VFR algorithm as an input.

Table 4.1 Device to Fog Latency.

	Fog 1	Fog 2	Fog 3	Fog 4
Dev 1	2	4	6	8
Dev 2	8	2	4	6
Dev 3	6	8	2	4
Dev 4	4	6	8	2
Dev 5	2	4	6	8
Dev 6	8	2	4	6

[1]https://www.ibm.com/analytics/cplex-optimizer.

Table 4.2 Linked Data Input.

	Dev 1	Dev 2	Dev 3	Dev 4	Dev 5	Dev 6
Dev 1	1	0	0	0	0	2
Dev 2	0	1	2	0	0	0
Dev 3	0	0	1	0	0	0
Dev 4	0	0	0	1	0	0
Dev 5	0	0	0	0	1	0
Dev 6	0	0	0	0	0	1

Table 4.3 Solution without Linked Data.

	Primary Fog	Backup Fog
Dev 1	Fog 1	Fog 2
Dev 2	Fog 2	Fog 3
Dev 3	Fog 3	Fog 4
Dev 4	Fog 4	Fog 1
Dev 5	Fog 1	Fog 2
Dev 6	Fog 2	Fog 3

Table 4.4 Solution with Linked Data.

	Primary Fog	Backup Fog
Dev 1	Fog 2	Fog 3
Dev 2	Fog 3	Fog 4
Dev 3	Fog 3	Fog 4
Dev 4	Fog 4	Fog 1
Dev 5	Fog 1	Fog 2
Dev 6	Fog 2	Fog 3

4.3 Task Allocation Algorithms

In this section, we describe the task allocation algorithms. As mentioned in Sect. 4.1, each device runs a Fog Resolver that is responsible for forwarding a task from the application in the device to the appropriate fog. The task from the application could explicitly specify its destination or leave it to the Fog Resolver. The Fog Resolver needs to take various factors such as the type of task, cloud and fog latencies into consideration while determining the best destination for a task.

We develop a probing-based task allocation algorithm. In this algorithm, tasks are sent to a set of potential destinations and based on the responses collected from the probed destinations, the allocation algorithm adapts. The design of the probing algorithm is quite simple and is inspired by the process adopted by the *Transmission Control Protocol* (TCP) [136] to detect the available bandwidth in a transmission link. In the design of the probing algorithm for task allocation, we consider the following objectives:

1. To use the least amount of resources (fogs) as possible, that is, to reduce superfluous task submissions as much as possible.

2. To recover from fog failures or slow fog responses as quickly as possible. The fog failure like condition can be triggered due to device mobility because a fog being used by a device could easily get distant from the device as the device moves or the network path gets congested.

3. To determine the characteristics and reconfigure the fog assignments in the best possible way as quickly as possible. The response characteristics of a fog depend on the application characteristics as well as the fog loading.

4.3.1 Using Fogs Only

Here, we introduce the algorithm that allocates the tasks. The algorithm is given a *virtualFog* as the input. The virtualFog consists of a homeFog (i.e., the main fog the device wants to reach) and a pool (a set of backup fogs the device could

reach). The virtualFog is obtained from the output of the optimizer. The algorithm is responsible for probing and determining the best fog among them to serve a particular task. The algorithm can either forward the task to (a) the homeFog or (b) the pool.

In option (a), the algorithm sends the task to the homeFog for service as it is determined to be a better choice based on relative latency to serve the tasks. On the other hand, in option (b), the algorithm determines the pool to be a better choice than the designated homeFog. This could be as a result of the homeFog being overloaded (i.e., it has many tasks that needs to be processed) or the homeFog failure.

In the algorithm, we run the simulation until a number of tasks have been served, alternatively, we could run it until a time duration has elapsed. The variables given are *h_resp, p_resp, decay, probes, numTask, order* and are all initialized to zero unless otherwise stated. Variable *h_resp* is the response time of the homeFog when a task is sent to it by the device. It encompasses the time it takes to serve the task, along with the waiting times and transmission times along the network. Variable *p_resp* is the response time for the pool when a task is forwarded to a fog in the pool, *probes* is introduced to measure the number of unsuccessful attempts at using the homeFog while *numTask* keeps count of the number of tasks that have been served. The *order* variable is used to increment the total delay exponentially. This is introduced to reduce the total number of tasks being submitted to the pool. Assuming that the homeFog is performing well, but the delay is of small value, superfluous tasks would be submitted to the pool. However, it can be avoided if the delay is large enough. Experiments were run to determine an optimal *order* value, and the results show that a value of 4 is sufficient. Similarly, the *decay* variable is introduced to reduce the delay when the homeFog is performing well and after a series of experiments, our results show that 0.3 is sufficient.

The ProbeFog algorithm generally works as follows: A device wants to send a batch of tasks for computation purposes. The first task in the batch is sent to both the homeFog and fog pool. This is because the delay is initially set to zero. Assuming that the response from the homeFog returns first (since it is the closest

Algorithm 1: Fog Probing Algorithm

```
1  Input: a virtual fog
2  function ProbeFog(virtualFog)
3    variables: h_resp, p_resp, decay, probes, numTask, order
4    while(numTask < TotalNumTasks)
5      if (first task)
6        send task to both homeFog and poolFog
7        note response times and calculate delay as in line 14
8        go to line 15
9      else
10       h_resp = send(task, homeFog)
11       if h_resp > delay
12         p_resp = send(task, poolFog)
13         probes++
14         delay = | p_resp − h_resp | ∗probes^order
15         if (h_resp < p_resp)
16           homeCount++ ;                    ▷ task is served by homeFog
17           responseTime = h_resp
18           ArrayResponseTimes.add(responseTime)
19         else
20           poolCount++ ;                      ▷ task is served by pool
21           responseTime = p_resp
22           ArrayResponseTimes.add(responseTime)
23       else
24         delay *= decay  ;                  ▷ decay is less than 1.0
25         homeCount++ ;                    ▷ task is served by homeFog
26         responseTime = h_resp
27         ArrayResponseTimes.add(responseTime)
28     numTask++
```

fog), the response time is measured and the result is returned to the device. At some later point, the response from the pool returns and the response time is also measured (we have 3 fogs in the pool, but we only take the fastest response time among them). We now set the delay to the absolute difference between the two response times. The next task is then sent to the homeFog and if the response does not return before the delay, we send the task to the pool, otherwise we reduce the delay. The reduction in delay is done to ensure that while the homeFog is performing well, the task will not be sent to the pool. Also, if the homeFog were to be overloaded or failed, there will not be a long wait before sending the task to the pool.

Assuming that the homeFog response does not return before the delay, two things can happen, the homeFog response can either arrive before the pool response or the pool response arrives before the homeFog response. In either case, we send the results of the task to the device and update the delay accordingly. It is quite possible that the cloud can be best suited to serving the application task especially when the virtualFog is not performing well. This could be due to overloaded tasks at the fog or network congestion. In Algorithm 1, the task allocation only deals with the homeFog and pool. Therefore, a complete solution to the task allocation problem needs to handle the cloud as well.

4.3.2 Using the Cloud

In this section, we introduce the *Virtual Fog Resolver* (VFR) algorithm. A pseudocode for VFR is shown in Algorithm 2. The VFR uses the cloud when it detects bad performance from the virtualFog.

The following new variables are introduced: *c_resp, cap_level, p_inc, T_min, T_max, probe_cloud and probe_prob.* All variables are initialized to zero unless otherwise stated. Variable *c_resp* is the response time of the cloud when a task is served by the cloud, *probe_prob* is the probability of probing the cloud, and it is initialized to 0.5 while *cap_level* is the maximum capacity level that the cloud probing probability (i.e., *probe_prob*) can reach. Variable *p_inc* is an over-subscription rate

that determines the probability with which a task should be sent to the cloud in the event of a malfunctioning virtualFog. Variable *probe_cloud* determines whether a task should be sent to the cloud, and it is initialized to false.

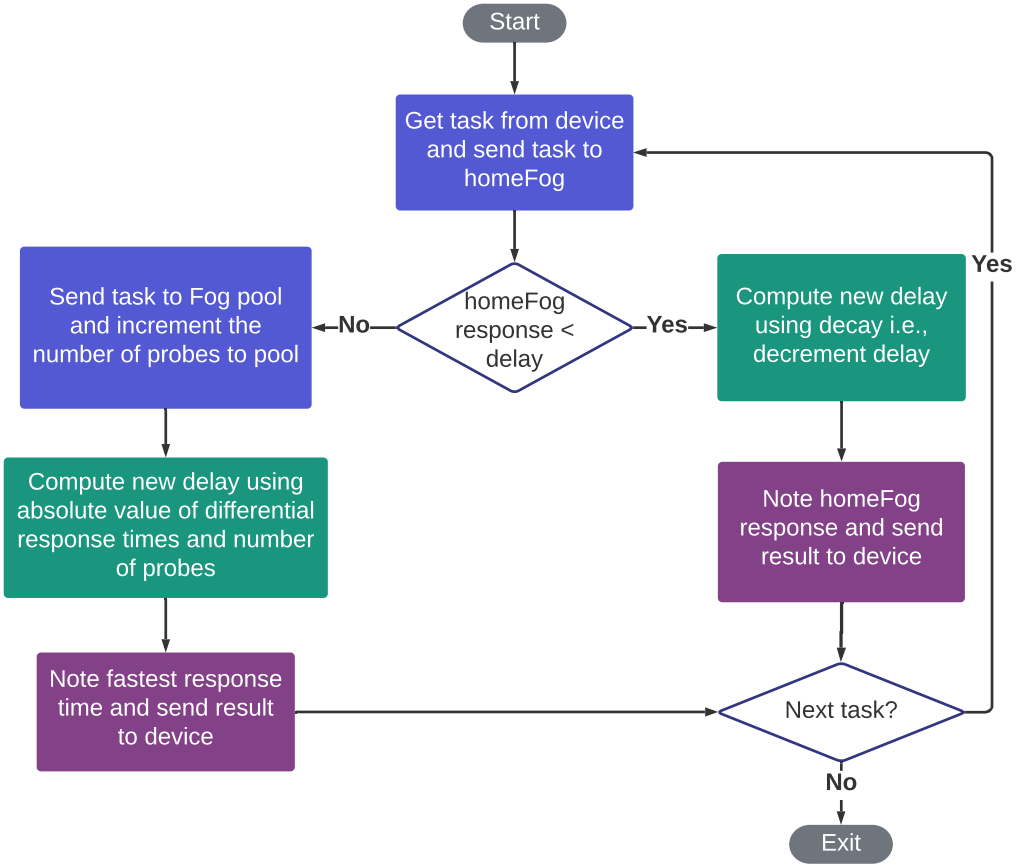

Figure 4.2 Flowchart of the Probing Algorithm

Variables T_min and T_max in lines 10 and 7, respectively, are defined as the lower and upper threshold of the expected response time of the virtualFog. For the simulations reported here, T_min is the average value of the responses received from the fogs in virtualFog and T_max is the maximum value of the responses received from the fogs in virtualFog.

The VFR algorithm is given a virtualFog by the Fog Allocator. The VFR

Algorithm 2: Virtual Fog Resolver (VFR) Algorithm.

```
1  Input: a virtual fog
2  function ProbeCloud(virtualFog)
3    variables: h_resp, p_resp, c_resp, decay, probes, numTask, order, p_inc
     T_min, T_max, probe_cloud, probe_prob
4    while(numTask < TotalNumTasks)
5      if ArrayResponseTimes is not empty
6        T_min = mean(ArrayResponseTimes)
7        T_max = max(ArrayResponseTimes)
8      if (first task)
9        send task to both homeFog and poolFog
10       note response times and calculate delay as in line 16
11       go to line 19
12     else h_resp = send(task, homeFog)
13     if home_resp > delay
14       p_resp = send(task, poolFog)
15       probes++
16       delay = | p_resp − h_resp | ∗probes^order
17     else
18       delay *= decay  ;                                  ▷ decrement delay
19     if probe_cloud is true
20       c_resp = send(task, cloud)
21       if h_resp or p_resp < c_resp
22         probe_prob *= decay ;                            ▷ decrement probability
23         with (1 - probe_prob) make probe_cloud false
24         if (h_resp < p_resp)
25           homeCount++; ▷ task is served by homeFog
26           responseTime = h_resp
27           ArrayResponseTimes.add(responseTime)
28         else
29           poolCount++ ;                                  ▷ task is served by pool
30           responseTime = p_resp
31           ArrayResponseTimes.add(responseTime)
32       else
33         cloudCount++ ;                                   ▷ task is served by cloud
34         responseTime = c_resp
35         ArrayResponseTimes.add(responseTime)
36     else
37       if h_resp or p_resp < T_min
38         if probe_prob < cap_level
39           probe_prob += p_inc
40         else
41           probe_prob = cap_level
42       else if h_resp and p_resp > T_min
43         with (probe_prob) make probe_cloud = true
44           c_resp = send(task, cloud)
45           responseTime = c_resp
46           ArrayResponseTimes.add(responseTime)
47       else if h_resp and p_resp > T_max
48         probe_cloud = true
49         responseTime = c_resp
50         ArrayResponseTimes.add(responseTime)
51     numTask++
```

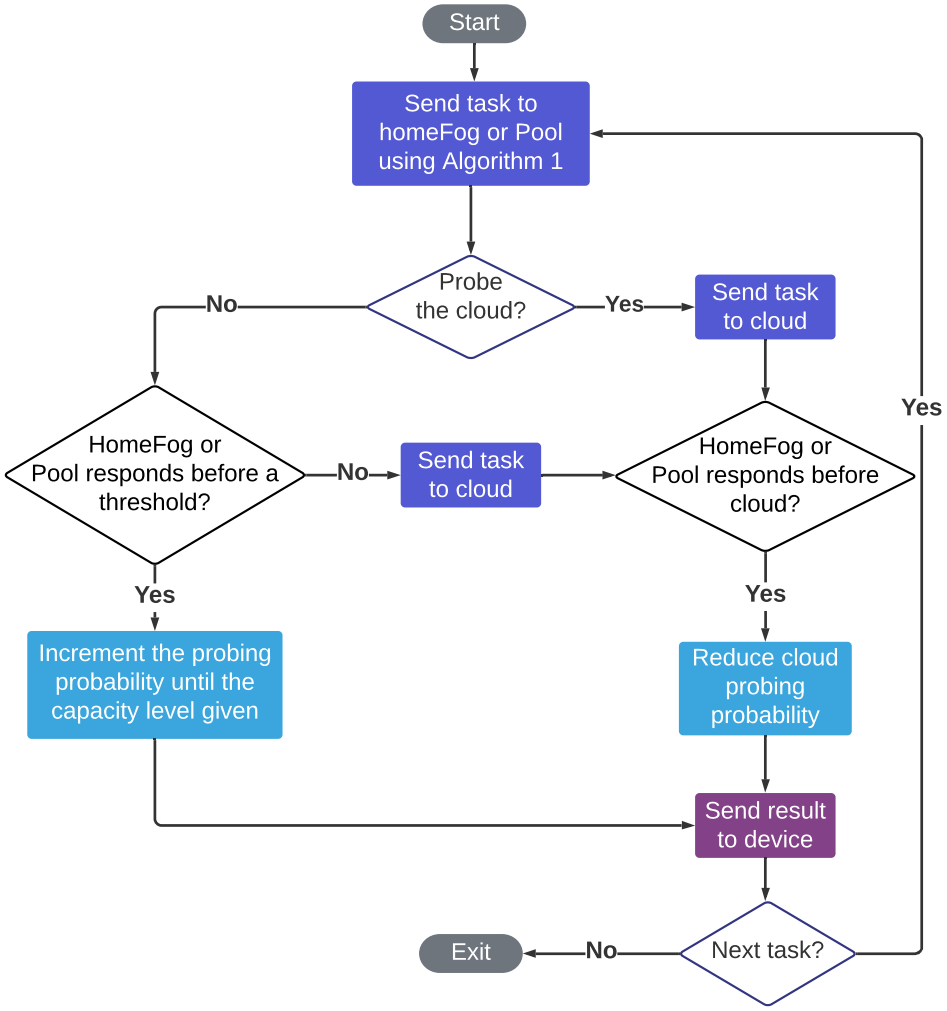

Figure 4.3 Flowchart of the VFR Algorithm

uses the same probing algorithm as in Algorithm 1 to send the tasks to homeFog and pool. While the tasks are routed to the virtualFog (i.e., not to the cloud), the cloud probing probability keeps increasing, but it is capped at the maximum value of *cap_level*. Once it reaches the *cap_level*, there is a very high probability of sending the task to the cloud should the virtualFog fail. However, when the cloud is used to serve a task (i.e, *probe_cloud* is true), if a virtualFog happens to perform better than the cloud (lines 19-31), then we decrement the cloud probing probability by multiplying it with a *decay* in line 22. This ensures that we do not send superfluous tasks to the cloud.

Lines 37-41 show that if the virtualFog responds before T_min (i.e., virtualFog is performing very well), we increment the cloud probing probability by the over-subscription rate p_inc. Line 42 ensures we send the task to the cloud with a computed probability if virtualFog does not respond after T_min. If virtualFog does not respond after T_max, we can consider that the virtualFog has malfunctioned and send the task to the cloud if it has not been sent yet.

The virtualFog is explicitly described as follows: Given that the cloud is now introduced, we only probe the cloud upon the failure of the fogs. However, the cloud is probed with some probability. This probability is increased or decreased based on the functioning of the virtualFog. A flowchart to describe both algorithms is given in Figs. 4.2 and 4.3.

4.3.3 Overload and Fault Tolerance

We consider a scenario whereby a task sent to a fog by the Fog Resolver fails to respond. This could be due to fog failure. In cloud computing, it is common to implement fault tolerance by replicating the services hosted in a virtual machine across several virtual machines with independent failure modes. This way, when one machine fails, another machine is bound to be available to take over the computation. More precisely, for a given service, we should be able to find a quorum of functioning server replicas running that service. In fog computing, at any given location, the amount of resources will be limited. As a result, a pure replication based fault tolerance strategy as obtained in the cloud is not suitable. When a fog node fails or has so many tasks to attend to, a pool is created from the partitions of nearby fog nodes and that partition (pool) is used as a backup to serve the task. The virtualFog algorithm detects when the homeFog is taking longer than usual to respond and forwards the application task to the fog nodes in the fog pool.

In our work, we had established a quorum to be a collection of 3 backup fogs, earlier termed as the fog pool. Upon partial failure of the homeFog, the results from the fog pool are updated to the homeFog when it is back up and running

properly. In the MinDelay algorithm [117], when the manager (i.e., fog resolver - used for comparison purposes) detects that the waiting time at the fog is greater than a specified threshold which might be due to an overload of tasks at the fog or fog failure, the resolver forwards the incoming application task to the neighboring fogs.

4.4 Evaluation and Simulation Results

In this section, we compare the performance of the proposed VFR algorithm with the existing ones. To deploy different task allocation approaches to a fog computing system, we developed a simulator in Java using the *Stochastic Simulation in Java* (SSJ) discrete event library [137]. The simulator can support configurations with cloud, many fogs connecting to the cloud, and devices connecting to the fogs. The experiments are carried out on a powerful machine cluster with the following hardware specification: it has 125 GB of system memory and an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz. We describe the other related algorithms in the next section.

4.4.1 Related Task Allocation Algorithms

In addition to our algorithm which is discussed in Sect. 4.3, we study four more algorithms in this chapter. They are the HomeFog algorithm, Power of Two (PO-2) algorithm, Modified Power of Two (MODPO-2) algorithm, and Minimum Delay (MinDelay) algorithm.

HomeFog Algorithm

The HomeFog algorithm is proposed and implemented as a baseline to evaluate the other task allocation algorithms. In this algorithm, the fog resolver routes the application task to the closest fog, which is termed the *homeFog*. It does not matter if the homeFog is overloaded or failed, all tasks are to be routed to the designated homeFog.

Power of Two(PO-2) Algorithm

The power of two is adapted from the algorithm presented by Mitzenmacher [138]. In PO-2, any two random fogs are selected from the available fog nodes. Among the two randomly selected fogs, the fog nodes with the fewer number of tasks on its queue are selected to serve the incoming application task.

The power of two choices states that having two choices leads to a significant reduction in the maximum load over having one choice [138]. With this observation, it is easy to notice that routing tasks to a single fog (as in the case of the homeFog) can lead to significant overloading. Although PO-2 is very good in load balancing, in fog computing, fog selection matters as much as the load. That is, the overall distance from the originating device and the fog is an important measure in selecting the fog.

Modified Power of Two (ModPO-2) Algorithm

We propose a modified Power of Two that addresses the major issue with PO-2, where tasks could be sent to far away fogs. Instead of selecting two random fogs, modified PO-2 selects two fog nodes in a predefined zone. The zones can be created by clustering the fog nodes such that all fog nodes in a cluster are close to each other (i.e., one fog in the cluster could be substituted for another fog without losing much in terms of response time). The application tasks are allocated to fogs within a particular zone. Therefore, random fog selections occur within the zone to which the device belongs to.

Minimum Delay (MinDelay) Algorithm

The MinDelay algorithm is an algorithm from the related literature that is reimplemented for comparison purposes [117] and to serve as a provisioning policy by which we can measure the performance of our proposed approach. In MinDelay, tasks are routed to the homeFog first. If the waiting time of tasks in the homeFog is greater than a threshold, the task is routed to its neighbors. The neighbors are the nearby fogs closest to the homeFog. However, when routing the task to the neighbors, the neighboring fogs are sorted in an ascending latency and waiting

times at the fog. The fog with the lowest latency and waiting time is probed first, and the rest are probed in that order. At a neighboring fog, the task is served if the waiting time at the fog is less than a threshold, otherwise, it is routed to the next fog in the neighborhood and so on. If all neighboring fogs of the homeFog have been probed and none can serve the task, it is then forwarded to the cloud.

4.4.2 Evaluation Scenarios

The allocation optimization model is solved using the IBM CPLEX optimization solver and it is implemented in Java. The solution to the model is used as input to the VFR algorithm. We simulated the different task allocation algorithms along with the proposed VFR and compared their performance and resource (fog) utilization. We evaluated them against the stated objectives in Sect. 4.3 while considering two scenarios; when there are no fog failures and when failures are present. The results indicate that the VFR approach provides a better overall performance while maintaining good utilization of fog resources.

4.4.3 Experimental Setup

A network of IoT devices, fogs, and cloud nodes are simulated using the process-based simulation facility offered in SSJ. The network topology is obtained from the Cogent network topology data available at the Internet Topology Zoo [139]. We use a subset of the graph provided by the dataset. The topology used in the simulations contained 1 cloud node, 20 fog nodes, and 290 devices. Each device hosts a number of applications, and this number is varied during the simulation. Each application issues 1000 tasks for service and the first 100 tasks are ignored to warm up the simulations. Tasks are processed in a FIFO order and the tasks are homogeneous for simplicity's sake (i.e., there is no variation in task type).

The link rate between nodes is 1 Mbps and the processing capacities of the fog nodes vary from 10 to 12 CPU cores while the cloud node is 100 CPU cores. The number of CPU cores is obtained from an aggregated Google trace data used to set up the fogs' configuration [140]. We ran each of the simulations 10 times and obtained an average of the runs.

For the optimization model, we used the location data obtained from the cogent data and computed a latency matrix, which was given as input into the optimization solver. We also used a fixed setup cost of $100. The number of tasks is varied from 1000 to 4000 tasks. The capacity of the fog is obtained from the Google trace data and normalized to serve a number of tasks at a time. The linked data input is a matrix that was generated to show the relationships between devices. In the experimental setup described, the allocation problem is solved very quickly (0.05 s) by the optimization solver. It is feasible to rerun the optimizer when the Fog Resolver determines the need for a change in virtualFog allocation.

In our simulator, there is a relationship between the number of tasks and the number of applications on a device. Each application issues a number of tasks for service - 1000 tasks each. By increasing the number of applications on a device, we are increasing the number of tasks that can be served by the fog resources. Therefore, 1 application issues 1000 tasks while 2 applications issue 2000 tasks and so on.

4.4.4 Discussion of Simulation Results

Comparing VFR versus Other Algorithms

In Figs. 4.4 and 4.5, we vary the number of applications per device which changes the workload intensity on the fog. We measure the response times and fog resource utilization in these experiments. By varying the number of applications per device, we effectively examine how the different algorithms are performing as the workload intensity changes.

Response time is the time elapsed since launching a task for service until it gets served by the selected server. The time includes the processing delay at the nodes (fogs or cloud), the propagation delay (i.e., sending the task on the link and getting a response back after it has been served) and the waiting time at the nodes.

We discuss the results in terms of the objectives stated in Sect. 4.3. However,

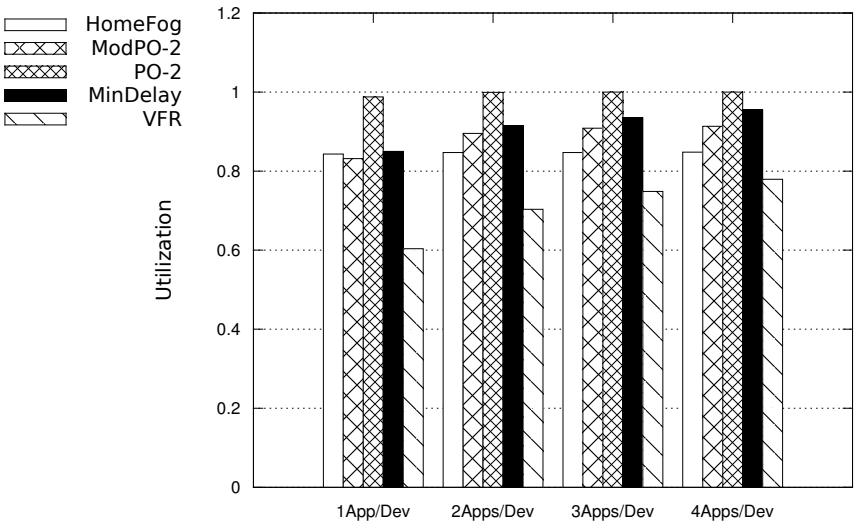

Figure 4.4 Resource utilization - variation of fog performance with different number of tasks/applications

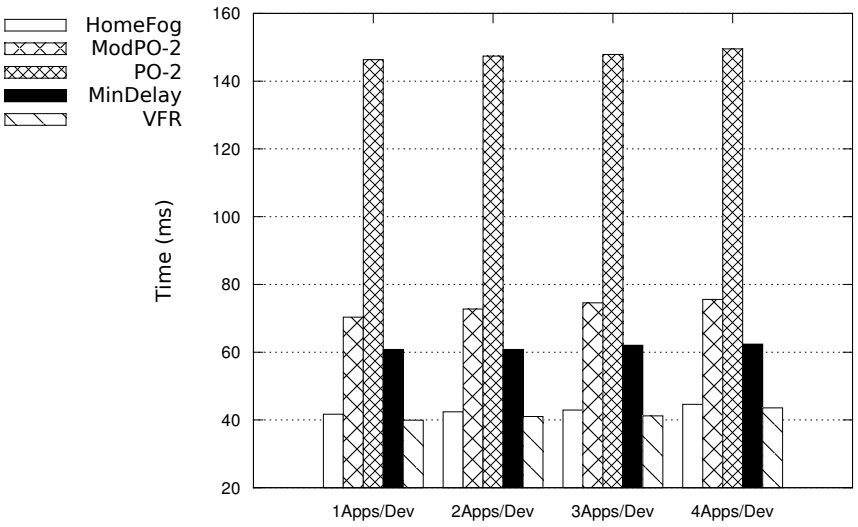

Figure 4.5 Response times - variation of fog performance with different number of tasks/applications

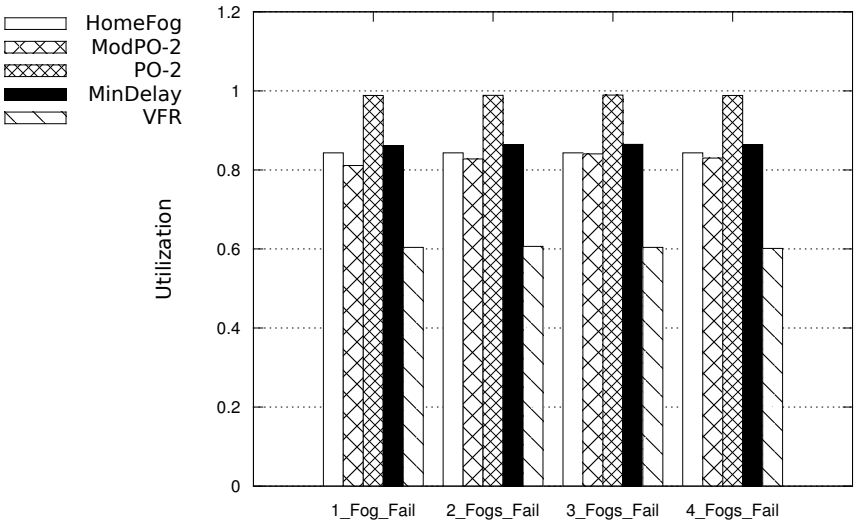

Figure 4.6 Resource utilization - variation of fog performance with different number of fog failures

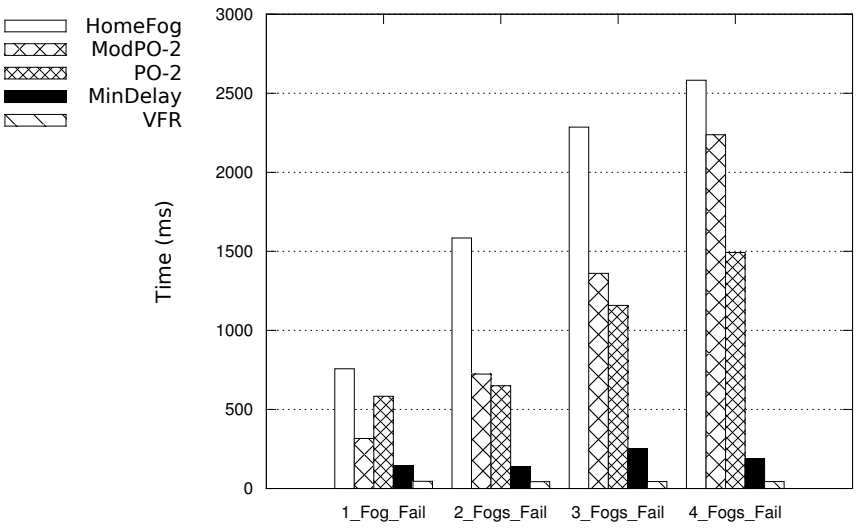

Figure 4.7 Response times - variation of fog performance with different number of fog failures

we look closely at two scenarios; when there is no fog failure and when there are fog failures.

1. As shown in Figs. 4.4 and 4.6, the resource usage for VFR algorithm is the lowest with PO-2 being the highest. In terms of resource efficiency, the proposed VFR algorithm uses the least amount of fog resources due to the optimal mapping of devices to fogs and the resources being well managed by the routing algorithm. PO-2 algorithm uses the most amount of fog resources due to the random selection of fog resources. Fig. 4.4 shows when the number of applications is increased per device and as the number of tasks being served increase, the resource usage increases as well. However, in Fig. 4.6, the number of fog failures is varied while keeping the number of tasks constant at 1000 or 1 application per device. The utilization across the algorithms is fairly constant with VFR using the least amount of resources and PO-2 using the most amount of resources.

2. To recover from fog failures, we measured the response times of the algorithms as fogs fail. Fig. 4.7 shows that the response time is minimal for VFR when compared to the rest of the algorithms. When the fogs fail, the failure is well masked to the requesting device as the task is processed by other available allocated fogs. As more fogs fail, the response time increases. Note that the baseline algorithm (HomeFog) has the highest response time because even if the designated homeFog fails, the task is still sent to the failed fog and it times out after a long time has elapsed.

3. To determine the characteristics and reconfigure the fog assignment, we measured the response times when the load intensity on the fogs increase. As observed from the results in Fig. 4.5, as the load intensity (the number of applications per device) increase, the response time increases as well. VFR has the lowest response times among the algorithms, while PO-2 has the highest response times. The response times of PO-2 algorithm can be attributed to the random selection of fog nodes. Fog node serving a device might be farther from the device, on the other hand, MODPO-2 performs better than PO-2 due to its selection criteria being biased toward nearby fogs, however, it is still worse off than the rest of the algorithms.

Comparing VFR against MinDelay

We examine how VFR and MinDelay are using the fog and cloud resources, respectively. To measure that, we look at the ratio of the tasks placed by the algorithm at fog or cloud divided by the total number of tasks. In this examination, we selected MinDelay and VFR because they use both cloud and fog components.

In Fig. 4.8, we obtained the percentages of jobs served by the homeFog, pool, and cloud for both VFR and MinDelay algorithms. While VFR uses more of its homeFog and less of the cloud, MinDelay algorithm is heavily reliant on the cloud. This also explains why VFR response time is better than MinDelay due to the distribution of the tasks to nearby fogs rather than relying on a distant cloud.

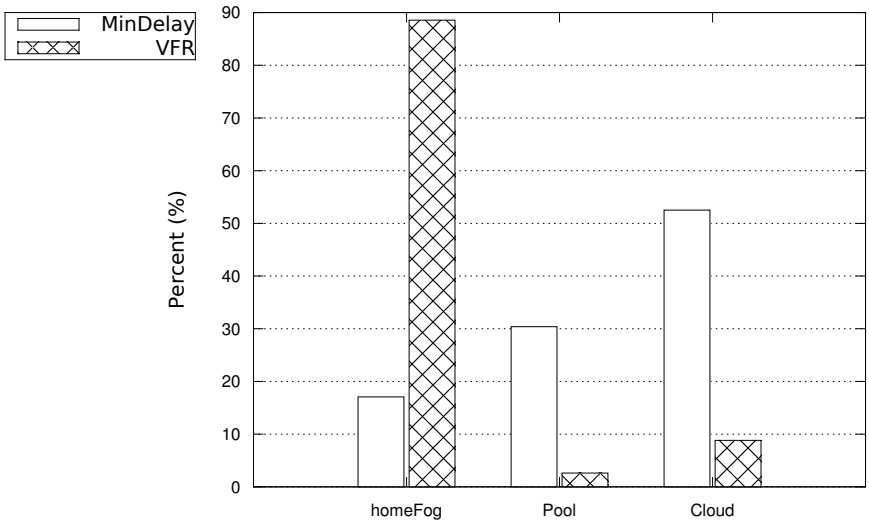

Figure 4.8 Percentage of jobs served by homeFog, pool and cloud.

As seen from the results, the VFR algorithm is robust in allocating tasks to appropriate servers. In terms of overload and failures, it can minimize the response times for service requests (i.e., requests to run application tasks), which is one of the primary goals of fog computing.

Chapter 5

Scheduling Framework for Real-Time & Non Real-Time Tasks

In the previous chapter, we focused on resource and task allocation algorithms for non real-time independent tasks. In this chapter, we focus on scheduling algorithms for applications with a mix of real-time and non real-time tasks. In particular, we focus on Edge Intelligence (EI) applications. Edge Intelligence [141, 142, 143, 71, 144] is one of the most exciting edge computing scenarios envisioned, where Artificial Intelligence (AI) is deployed at the edge of the network to process the data there and respond to requests from devices and users very fast. EI is expected to enable several new application use cases including EI for autonomous driving [145, 146], EI for collaborative robots [147], EI for smart spaces [148, 149, 150], and EI for environmental sensing [151]. Although EI got much of its impetus from 5G, it is considered as a key driver of wireless technologies 6G and beyond [152, 153].

Some key use cases of EI need coordinated scheduling of tasks across different resources. For instance, with platoons of autonomous cars or collaborative robots [11, 154], it is necessary to perform real-time task scheduling with coordination requirements. The task scheduling model we consider in this chapter supports

such coordinated scheduling. In our scheduling model, tasks are categorized into real-time, interactive, and batch classes. The real-time tasks are scheduled by a global scheduler. The allocation of a time slot in the global schedule puts the tasks for execution on the same time slot across different resources, thus, providing the task coordinated execution across the resources.

In this chapter, we introduce a two-stage scheduling framework to support EI. An edge zone would run a global scheduler (also referred to as the L1-Stage). The L1-Stage only handles tasks with real-time constraints. The L1-Stage of the scheduler is responsible for creating partial schedules and matching them with L0-Stage schedulers that run in the eventual resource. The L0-Stage scheduler takes the partial schedule from the L1-Stage and activates it subject to the constraints in the schedule. The L0-Stage scheduler is non-preemptive and is implemented at the user level. It can influence the underlying kernel-level scheduler to get a share of the CPU. Because the L0-Stage scheduler is non-preemptive, we insert pseudo-preemption (yield) points into the tasks to keep the interactive and batch tasks from hogging the CPU. We implemented the two-stage scheduler as a middleware and ran it on Linux and macOS. The results indicate that the flexibility offered by the two-level architecture improves performance. The L0-Stage can decide to switch between the normal and greedy modes depending on the local task mix to maximize the number of tasks successfully completed by the scheduler.

5.1 Motivating Scenario

One of the key use cases of EI is collaborative robots (Cobots) [11, 155]. Cobots is a well established idea in robots that deals with machines that heavily collaborate with humans like in factory floors and are expected to play an increasingly prominent role in many other domains (e.g., healthcare facilities). Cobots need to do autonomous navigation within a factory floor, automatic monitoring of machine health properties, collaborate with other cobots, and collaborate interactively with humans. For instance, autonomous navigation can be solved by a collaborative activity among the cobots and the navigating computers on the factory floor that are in the same zone as the cobots. The cobots can be connected using a 5G

network. To complete the collaborative computations for autonomous navigation and other collaborative activities on time (for real-time processing), we need to schedule the tasks across all different computers (including the cobots) at the same time. Further, tasks associated with autonomous navigation have real-time deadlines because without completing the navigation processing, the cobots would not be able to proceed with the next move. This brings us to the first problem.

Problem P-1 – Real-time and coordinated task scheduling: The distributed resource manager we seek for EI must handle this problem efficiently. EI would run on virtualized platforms (e.g., virtual machines) that would be hosting multiple tenants. As a result, the infrastructure (i.e., underlying OSes) itself may not support real-time computing. We need solutions that implement real-time support at the user level.

As one of the key purposes of cobots is to create robots that collaborate with humans, cobots can receive a lot of user interaction tasks. Unlike the real-time tasks in **P-1**, these tasks happen at human speeds and can also tolerate low rates of error. For the users to have responsive interactions with the cobots, we need to complete as many interaction tasks as possible before their soft deadlines elapse. This brings us to the second problem.

Problem P-2 – Responsive scheduling of interactive tasks: Cobots can interact with the humans at different rates based on where they are on the factory floor and the particular application they are running. In certain times, the cobots need to give higher preference for interactive tasks and reduce their engagement with other cobots while at other times they need to run more of the real-time tasks with coordination constraints. The resource manager needs to shift the task scheduling preferences to cater to the changing requirements.

Cobots need to run periodic health checks, analytics, and other data-intensive tasks to detect impending failures and keep the overall reliability levels high. Before the cobots undertake safety critical tasks, we need to schedule all necessary health checks to minimize cobot failures. Similarly, analytics tasks are necessary

to improve the overall management procedures on the factory floor and they must be scheduled when cobots and the factory computing infrastructure have sufficient free processing capacity. This brings us to the third problem.

Problem P-3 – High-throughput batch task processing: The resource manager needs to give a higher preference for the real-time and interactive tasks. However, if the batch tasks keep piling up, the cobots might have to be taken away from routine activity into a service mode so that pending tasks can be cleared.

In the sections below, we show how the scheduling framework we propose in this chapter addresses problems P-1 to P-3.

5.2 System Model and Scheduling Algorithms

In our system model, applications can have three types of tasks: real-time tasks, interactive tasks and batch tasks [156]. Real-time tasks of the application are periodic in nature and have a hard deadline, whereas interactive and batch tasks are aperiodic. Interactive tasks have soft deadlines and batch tasks do not have deadlines.

Let τ_i^R denote a real-time task in the application. It has a corresponding arrival time a_i^R (time the task is launched for service), computation time c_i^R, deadline D_i^R, and period T_i^R. Let τ_i^I denote an interactive task in the application. It has a corresponding arrival time a_i^I, computation time c_i^I and deadline D_i^I. Similarly, a batch task of the application is denoted by τ_i^B and it has a arrival time a_i^B, and computation time c_i^B. Note that "R", "I" and "B" refer to Real-time, Interactive and Batch tasks, respectively. The full set of parameters for the tasks are given in Table 5.1. The following assumptions are made about the task properties:

- Parameters of periodic real-time tasks are known a *priori* given the deterministic nature of the tasks.
- The worst-case computation time is assumed.
- Real-time tasks deadline is equal to its period, i.e., $D_i^R = T_i^R$.

- Preemption of tasks is not allowed.

Table 5.1 Real-time, Interactive and Batch Task Parameters

Parameters for a real-time task τ_i^R	
a_i^R	the arrival time of the first instance of τ_i^R
c_i^R	the computation time of τ_i^R
D_i^R	the relative deadline of τ_i^R (the deadline of each τ_i^R instance is D_i^R time units after its arrival);
d_i^R	the absolute deadline of τ_i^R; $d_i^R = a_i^R + D_i^R$
T_i^R	the period of τ_i^R, which is the length of time between the arrivals of two consecutive τ_i^R instances
Parameters for an interactive task τ_i^I	
λ_i^I	the average arrival rate of instances of τ_i^I
c_i^I	the computation time of τ_i^I
D_i^I	the relative deadline of τ_i^I
Parameters for a batch task τ_i^B	
λ_i^B	the average arrival rate of instances of τ_i^B
c_i^B	the computation time of τ_i^B

A two-stage approach for scheduling the application tasks is proposed. The first stage termed 'L1-Stage' deals with scheduling the real-time tasks of the application while the second stage, termed 'L0-Stage' deals with executing the real-time tasks according to the schedule from the L1-Stage and executing interactive and batch tasks as well. In the first stage, real-time tasks are mapped to appropriate resources based on the task requirements using a task matching algorithm. For example, the task requirements could be the amount of memory or CPU cycles the edge resource should have for processing the task. A scheduling algorithm is then proposed to generate schedules for the tasks which will be executed on the matched resources. The schedules generated are termed 'Normal' and 'Greedy' schedules according to the scheduling algorithm.

In the second stage, the L0 scheduler running at the resource executes the tasks using the generated schedules. The scheduler selects the best schedule among the two schedules at each scheduling cycle. The selection is based on the arrival pattern of the real-time tasks and how the slack server is utilized by the interactive

Table 5.2 Important concepts used in the algorithms

Term	Definition
Hyper-period	the Least Common Multiple (lcm) of all periods of the real-time tasks in the system. It is also referred to as scheduling cycles in the chapterr.
Matching algorithm	the task matching algorithm is developed as a matrix where the real-time tasks are specified on the rows and their resource attribute requirements are specified on the columns. Resources are matched to the columns of the matrix.
Payoff(s)	are associated with tasks in each column in the specified matrix. It allows tasks with higher priority to be selected. It is also adjusted at every matching iteration to limit task starvation. In the chapter, we use constants, however, the payoff is a tuning knob the tasks can use to influence the decision of the matching algorithm.
Priority values	these are values obtained as a result of the total payoffs in the columns of the matrix.
Slack servers	are the idle processing time available after real-time tasks have been scheduled.
Scheduling algorithm	is an algorithm proposed to create static schedules for the real-time tasks assigned to a resource. Two static schedules are developed. They are 'Normal' and 'Greedy' schedules.
Normal schedule	is an extension of the non-preemptive Earliest Deadline First (EDF) algorithm. Real-time tasks are scheduled using the EDF algorithm.
Greedy schedule	developed as a means of enlarging the slack servers available in the Normal Schedule. We define this concept as "borrow". This is the second schedule developed in the scheduling algorithm.
Fitness values	are associated with the utilization of the slack servers based on the arrival of the tasks. This value is used to control the selection among the two schedules at every scheduling cycle.
Virtual clock	is used for selecting between batch and interactive tasks when a slack server becomes available. Penalties associated with the virtual clock is to give preference to interactive tasks due to missed deadline at the head of the interactive queue.

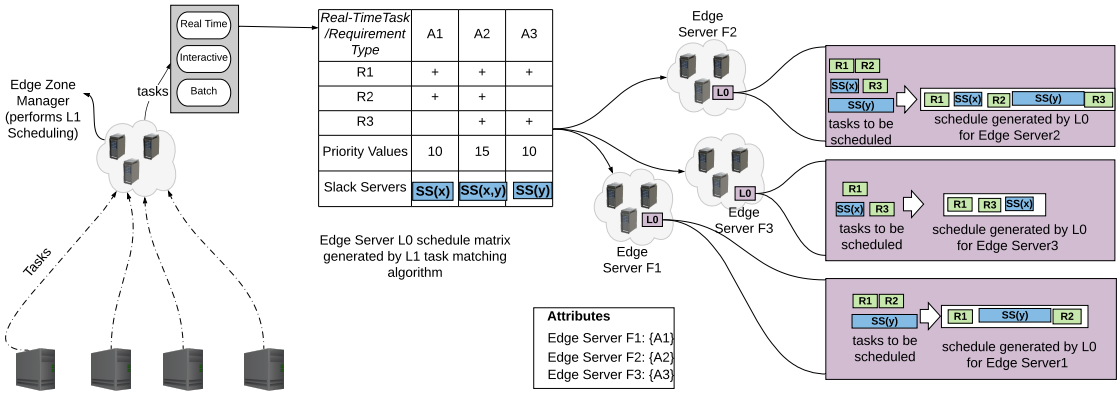

Figure 5.1 Scheduling Architecture

and batch tasks. A slack server is the idle processing time available for executing interactive and batch tasks without violating the deadline requirements of the real-time tasks. A fair selection algorithm is proposed to select between batch and interactive tasks when a slack server is available. This is to ensure the non-starvation of either batch or interactive tasks.

Problem P-1 identified in Section 5.1 is addressed using the task matching and scheduling algorithms while Problems P-2 and P-3 are addressed using the fair selection algorithm.

The two-stage scheduling architecture is shown in Fig. 5.1. We define the important concepts used in the scheduling architecture and algorithms in Table 5.2. Tasks arrive from devices to the L1-Stage edge zone manager in a particular zone. Batch and interactive tasks are queued up at the edge zone manager in the zone. The figure shows three real-time tasks R_1, R_2 and R_3 and three edge servers denoted as F_1, F_2 and F_3. The edge servers satisfy different location requirements, such as their association with different areas of a zone. For instance, edge server F_1 is placed in Area A_1, F_2 in Area A_2 and F_3 in Area A_3 assuming that the zone is divided into three distinct areas A_1, A_2 and A_3. Task R_1 requires that the task is served in all areas of the zone, task R_2 requires that the task should be served in areas A_1 and A_2 while task R_3 requires that the task should be served in areas A_2 and A_3. Furthermore, note that for simplicity, the payoff values are equal for

each task (5) since the edge server selection is exclusive, but it need not be.

The task matching algorithm generates a set of columns as shown in the figure based on the requirement specification. In the first column, A_1, edge server F_1 satisfies the location criteria since it is located in A_1, therefore it selects the column. Similarly, edge servers F_2 and F_3 select columns A_2 and A_3 respectively. In this example, a location requirement was used for simplicity, therefore, edge servers can only be placed at one location at a time. However, in a general case, the requirement need not be exclusive. Edge servers can satisfy one or more requirements. In such a scenario, edge servers match columns with the highest priority value. After the resources have selected the columns, a scheduling algorithm is used to generate real-time task schedules at the respective resources. The schedule contains the ordering of the tasks and idle processing times, known as slack servers. In the second stage, the resource executes the tasks using the generated task schedules. The two stages are described fully in the next sections.

5.2.1 L1-Stage Scheduling

In this stage, two algorithms are proposed, and they are run in the edge zone manager. The algorithms are the task matching algorithm and the scheduling algorithm. The task matching algorithm is shown in Algorithm 3 on page 65. It takes as input a set of real-time tasks, their resource attribute requirements, i.e., the attributes the resources should have when executing the task. It uses the task requirements to create a schedule that consists of a subset of tasks collectively satisfying the resource requirement.

Lines 5 - 6 assign the real-time task set (i.e., rows) and edge server attributes (i.e., columns). Line 7 initializes the set of tasks assigned to a resource attribute. Lines 8 - 12 relate to step 1, it checks the task requirements and assigns the task to a column matching the resource attribute. Lines 13 - 18 relate to step 2, it carries out a schedulability test [1] check on the tasks assigned to a resource type. Lines 19 - 21 relate to step 3, it computes the priority value(s) which is the sum of the

[1] The tests is given as necessary conditions in Section 5.3.

individual task payoffs matched to the resource type. Edge servers select columns, i.e., a group of tasks, based on the priority values if they match more than one resource attribute. Lines 22 - 24 relate to step 4, it calls the scheduling algorithm (which is described next) on the selected columns to generate the schedules of the tasks.

The scheduling algorithm is given in Algorithm 4 on page 66. It takes as input a set of real-time tasks. The output is two schedules containing reservation slots for real-time tasks and the slack server(s), which is the idle processing time left after reserving slots for real-time tasks on the edge server resource. Lines 9 - 13 describe the normal schedule generation. The real-time tasks are sorted according to their deadlines in a non-decreasing manner. The hyper-period of the task set is computed and a counter for each task which shows the number of task instances expected in the hyper-period, is derived. The task instances are then scheduled using their earliest starting times. Lines 14 - 25 describe the greedy schedule. The greedy task scheduling is divided into two components. The tasks are sorted according to their laxity, i.e., 2 * (Deadline - Computation time). Ties are broken in favour of the task with a larger period. The first two periods are the first component of the greedy schedule, and the remaining periods until the hyper-period is the second component. In the first component, the first two instances of tasks whose deadline falls within the range of the first task in the sorted list are scheduled using the earliest and latest starting times respectively. By scheduling the second instance using its latest starting time rather than the earliest starting time, we introduce the concept of 'borrow' by flexibly shifting the task to accommodate a larger idle processing time, i.e., a slack server. In the second component, task instances are scheduled using their earliest start times. The purpose of the greedy schedule is to compact the real-time tasks as much as possible to fit in large slack servers as possible. The scheduling algorithm thus produces two schedules: a normal schedule where the 'borrow' concept is not applied and a greedy schedule where the 'borrow' concept is applied. An example illustrating the scheduling algorithm producing normal and greedy schedules on a set of real-time tasks is shown in Fig. 5.2 on page 67. The tasks shown are $R1, R2, R3$ and $R4$. The values for the tasks e.g., R1(0,1,5,5) represent the arrival time of 0, a computation

time of 1, a deadline and period of 5. The normal schedule produces slack servers with execution lengths {1,1,1,0.5,0.5,3} and the greedy schedule produces slack servers with execution lengths {4,3}.

The normal schedule is an implementation of the non-preemptive Earliest Deadline First (EDF) algorithm and the schedulability test is given in [157]. In the normal schedule, the scheduler considers the *deadline* D_i^R as the task property and schedules the tasks as soon as possible using the earliest start time denoted as s_i^{min}, whereas, in the greedy schedule, the scheduler considers the following task properties: the *deadline* d_i^R, *computation time* c_R^i, *earliest possible start time* s_i^{min} and *latest possible start time* s_i^{max}. Initially $s_i^{min} = a_i^R$ and $s_i^{max} = D_i^R - c_i^R$, but, in contrast to D_i^R and c_i^R, these properties are not constant. Instead, they vary depending on the properties of the other tasks and the scheduling decisions. The interval $[s_i^{min}, s_i^{max}]$ represents the period where the *ith* task (τ_i^R) must begin its execution. This interval cannot be increased, but only decreased as a result of the other tasks. In particular, if we know that a task (τ_i^R) must be postponed, i.e., delayed for a certain amount of time, its s_i^{min} parameter must be increased. Similarly, if this increase means that another task (τ_j^R) cannot start as late, s_j^{max} would be decreased.

The schedulability test for the scheduling functions - normal and greedy schedules is given in Section 5.3. The scheduling algorithm generates global schedules for the real-time tasks that effectively create reservations for the tasks. This solves Problem P-1 that needs coordinated execution over multiple resources.

5.2.2 L0-Stage Scheduling

In this stage, the L0-Scheduler runs at each resource. The schedules are assigned to the resource by the L1-Scheduler running in the zone manager in the L1-Stage. The tasks of the application are executed in this stage. Real-time tasks are executed on arrival according to the schedule to ensure end-to-end performance guarantees. Interactive and batch tasks are queued up and executed at the resource and are offered best-effort service.

Algorithm 3: Task Matching Algorithm.

Input: Real-time task set T with resource type requirements, Resource types A
Output: Task schedules
function TaskMatch(A, T):
  step 1: Do initialization
    A = resource type i, i = 1,2,...,m;
    T = real-time task j, j = 1,2,...,n;
    $S_i = \emptyset$ *Set of tasks assigned to a resource type i*
    foreach *task j*
      foreach *resource type i*
        Check the requirements of task j
          if a resource type requirement is specified as compulsory, tag with '+', if optional, tag with '' otherwise leave as blank*
          Append required symbol S_{ij}
  step 2: Do feasibility check
    foreach *resource type i*
      Carry out schedulability tests
      if *feasible*
        S_i *is the schedule for each type resource type*
      else *drop some optional tasks OR Append column as not feasible*
  step 3: Compute priority values
    foreach *resource type i*
      priority value $= \sum_{j=1}^{m} P_j$
  step 4: call Scheduling Algorithm
    foreach *resource type i*
      call *scheduling algorithm on feasible column (or schedule)*

Algorithm 4: Scheduling Algorithm.

Input: Real-time task set T, mode m
Output: Schedule containing real-time tasks and Slack servers
function GenerateSchedule(T, m)
  variables: *P_i : period of task i, C_i : computation requirement of task i, D_i : deadline of task i, H : Hyper-period of task set, Cnt_i : Counter - (H/P_i) of task i, s_i^{min}: earliest start time of task i, s_i^{max}: latest start time of task i, n : number of tasks in the task set.*
  if *(m is equal to 1)*
    call *NormalSchedule(T)*
  else
    call *GreedySchedule(T)*
function NormalSchedule(T)
  Sort the tasks according to their non-decreasing deadlines
  for *i = 1 to n*
    for *j = 1 to Cnt_i*
      Schedule task i using its s_i^{min} at each period
function GreedySchedule(T)
  Sort the tasks according to their idle times over 2 periods using $2(D_i - C_i)$
  If there is any tie, break in favour of the task with the largest period
  Select the first task and over the initial 2 periods, do the following
  Schedule first two instances of the selected task over 2 periods using s_i^{min} and s_i^{max}.
  for *i = 2 to n*
    if *$task_i$ $2D_i$ < $task_1$ $2D_1$*
      Schedule first 2 instances of task i over 2 periods using s_i^{min} and s_i^{max} respectively
    else
      Schedule the first instance of task i using its s_i^{min}
    For the remaining periods after the initial 2 periods of the first scheduled task
      Sort the tasks using their deadlines and schedule them using their s_i^{min}

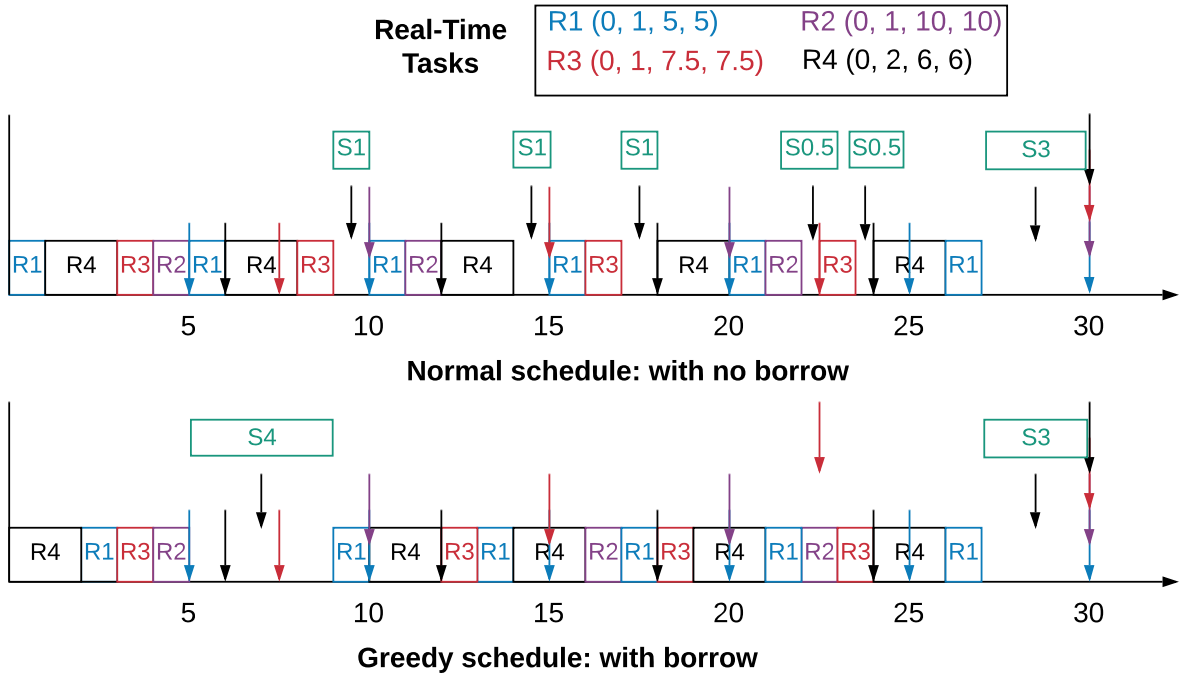

Figure 5.2 Illustration of the scheduling algorithm - Normal and Greedy schedules

To avoid starvation between both task types when a slack server is available to execute them, we propose a fair selection algorithm that uses a virtual clock to ensure fairness. Each task queue has an associated virtual clock such that whenever a slack server is available, the slack with the lower virtual time is selected. The fair selection algorithm is shown in Algorithm 5 on page 69. Line 4 checks the virtual clock time of both queues. Lower clock time at the batch task queue ensures tasks are selected from the batch queue, otherwise tasks from the interactive queue (Line 13) are selected. The tasks in the batch queue are sorted using a First In First Out (FIFO) policy. Line 6 checks if the selected batch task can be executed on the available slack server. Line 7 executes it, and the available slack server time is reduced by the task execution time in line 8.

Line 13 selects a task from the interactive queue. Tasks in the queue are sorted according to their deadlines. Lines 15 and 16 check whether the task can be exe-

cuted on the slack server before the server's deadline. If the task is executed, the virtual clock is incremented by the execution time while the slack server time is appropriately reduced.

However, if the task execution length, i.e., the time required is more than the available slack server time length, then the task is dropped because it cannot be executed without real-time tasks missing their deadlines. Line 23 checks if the deadline is already past the slack server deadline and if true, tasks within the slack server deadline are selected and executed.

The fair selection algorithm through the use of a virtual clock ensures that interactive tasks that will meet their deadlines are scheduled while also ensuring that batch tasks are not starved. In doing so, Problems P-2 and P-3 identified in Section 5.1 are addressed.

The selection between the normal and greedy schedules is carried out by the L0-Scheduler at this stage. The selection is based on the utilization of the slack servers, which is determined by the arrival of the tasks. The selection is handled by the fitness values of the slack server. At each scheduling cycle, the slack servers from both schedules are probed, and the best schedule is selected. Whenever a task is scheduled on the slack server, the slack server is rewarded, whereas dropped tasks lead to a smaller reward or a 'penalty' being assigned to the slack server. The cumulative reward, i.e., the sum of rewards and penalties, determine which schedule is chosen to execute the tasks. The flowchart for the fair selection algorithm is given in Fig. 5.3.

5.3 Schedulability Test for Scheduling Algorithm

In this section, we show the necessary conditions for our algorithm that ensure that the hard deadline of the periodic tasks is guaranteed. An algorithm's schedulability test checks if a task set is schedulable without building the entire execution sequence over the scheduling period [158].

Algorithm 5: Fair Selection Algorithm.

```
1  Input: Slack server S with length L_s and deadline D_s
2  function SelectTask(S)
3    variables: t_l^B : batch task execution length, t_l^I : interactive task execution
     length, t_d^I : interactive task deadline, V_B: batch queue virtual time, V_I :
     interactive queue virtual time
4    if V_B < V_I
5      Select tasks from batch queue (sorted by FIFO)
6      if t_l^B <= L_s
7        Execute the task and advance the virtual time V_B by execution length
8        L_s = L_s − t_l^B
9        reward fitness_value
10       Go to line 10
11     else if t_l^B > L_s
12       skip task and penalize fitness_value
13   else if V_B >= V_I
14     Select task from interactive queue (sorted by EDF)
15     if  t_d^I < D_s
16       if  t_l^I <= L_s
17         Execute the task and increment the virtual time V_I by task execution
     length
18         L_s = L_s − t_l^I
19         reward fitness_value
20         Go to line 10
21       else if  t_l^I > L_s
22         skip task and penalize fitness_value
23     else if  t_d^I > D_s
24       decrement the virtual time V_I and drop the task
25       Go to line 14
```

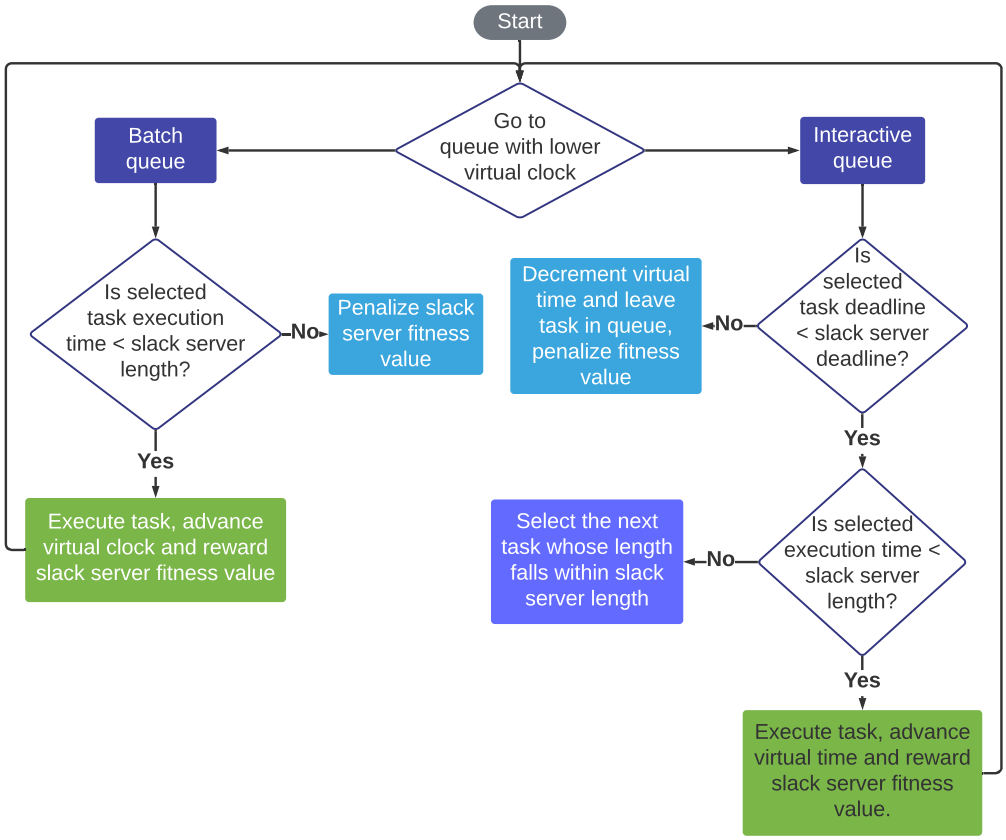

Figure 5.3 Selection Algorithm Flowchart

The scheduling algorithm consists of two functions: a normal schedule function which is a description of the non-preemptive Earliest Deadline First (EDF) algorithm and a greedy schedule function which combines Least Laxity First and Earliest Deadline First algorithms. The necessary conditions for the normal schedule are given in [157] and is repeated here in Theorem 1. In addition to these conditions, an additional condition which is adapted from [159] is necessary for the schedulability of the greedy schedule, and it is given in Theorem 2.

The greedy schedule tasks are sorted according to their laxity in a non-decreasing order. The laxity is given as $D_i^R - c_i^R$. In our work, ties are broken in favour of larger periods. We observe that since the laxity is computed at the initial arrival of the tasks and all tasks are released at time 0, the Least Laxity First algorithm

is equivalent to the Earliest Deadline First algorithm. Therefore, conditions that are necessary for EDF also apply to the greedy schedule function. We derive the feasibility conditions for the greedy schedule function.
The following theorems establish the necessary conditions for ensuring the correctness of the scheduling functions.

Theorem 1. *Let τ^R be a set of periodic real-time tasks $\{\tau_1^R, \tau_2^R, ..., \tau_n^R\}$ sorted in a non decreasing order by period, the following conditions are necessary for τ^R to be scheduled non-preemptively by a uni-processor.*

$$U \leq 1 \; where \; U = \sum_{i=1}^{n} \frac{c_i^R}{T_i^R} \tag{5.1}$$

$$\forall i, 1 < i \leq n; \forall L, T_1^R < L < T_i^R; L \geq c_i^R + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{T_j^R} \right\rfloor c_j^R \tag{5.2}$$

Proof: Condition (5.1) ensures that the resource cannot be overloaded, and condition (5.2) ensures that the worst case processor demand in an interval cannot exceed the length of the interval. The necessity of both conditions are proven in [157] but are repeated here. To validate condition (5.1), we show the following:

For a set of tasks τ^R, the *processor demand* in the interval [a,b], written $d_{a,b}$ is defined as the minimal processing time required by τ^R in the interval [a,b]. That is, $d_{a,b}$ is the minimum amount of processor time required in the interval [a,b] to ensure that no deadline is missed in the interval [a,b]. If a set of tasks τ^R is feasible, then for all a and b, $a < b$, it follows that $d_{a,b} \leq b - a$.

For all i, $1 \leq i \leq n$, let $a_i^R = 0$ and let $t = T_1^R * T_2^R * ... * T_n^R$. In the interval [0,t], $\frac{t}{T_i^R} C_i^R$ is the total processor time that must be allocated to task τ_i^R to ensure that τ_i^R does not miss a deadline in the interval [0,t]. If τ is feasible then

$$d_{0,t} = \sum_{i=1}^{n} \frac{t}{T_i^R} c_i^R \leq t,$$

or simply

$$. \sum_{i=1}^{n} \frac{c_i^R}{T_i^R} \leq 1.$$

For condition (5.2), choose a task τ_i^R, $1 < i \leq n$, and let $a_i^R = 0, a_j^R = 1$ for $1 \leq j \leq n, j \neq i$. This gives rise to a pattern of task execution requests shown in the Figure 5.4

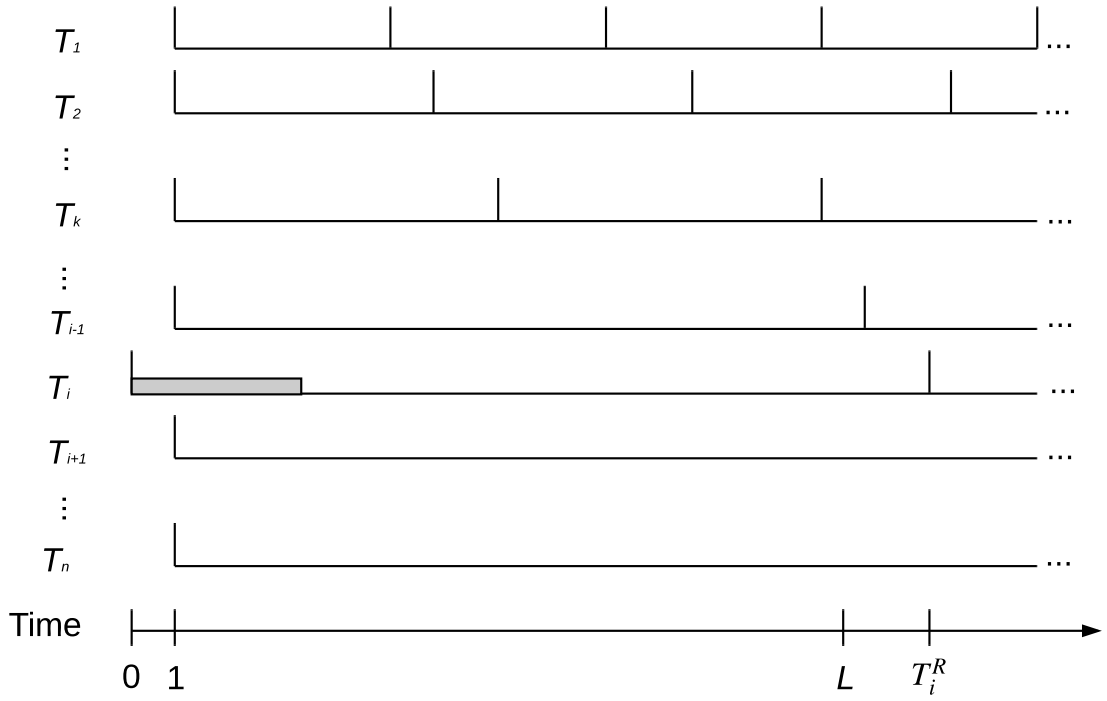

Figure 5.4 Construction for the necessity of condition (5.2)

For all L, $T_1^R < L < T_i^R$, in the interval [0,L], the processor demand $d_{0,L}$, is given by

$$d_{0,L} = c_i^R + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{T_j^R} \right\rfloor c_j^R.$$

Hence for τ to be feasible, we must have

$$L \geq c_i^R + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{T_j^R} \right\rfloor c_j^R.$$

Theorem 2. *The following condition in addition to the condition (5.1) are necessary for the schedulability of tasks scheduled by the greedy schedule.*

$$c_i^R \leq 2(L_1^R - c_1^R) \tag{5.3}$$

Proof: Condition (5.3) limits the execution time of tasks to the slack of the least laxity task. The tasks are sorted in non-decreasing order by laxity with arbitrary period ratio, $k_i > 0, 1 < i \leq n$. To validate Condition (5.3), the tasks are indexed as follows: $L_1^R, L_2^R, \ldots, L_n^R$ where L_1^R is the period of the first task sorted by its laxity. The period ratio is denoted by $k_i \in \mathbb{R}^+$ where $k_i = L_i^R / L_{i-1}^R$ for $1 < i \leq n$.

We select k_2 and consider the following three cases regarding the value of k_2:

Case 1: If $k_2 < 1$, then $L_2^R < L_1^R$ i.e., $L_1^R > L_2^R$. According to Condition (5.1) the utilization $U \leq 1$, therefore, $c_1^R + c_2^R \leq L_1^R$.

Moreover, for the same reason, $c_1^R \leq L_1^R$. If both inequalities are added, we obtain $c_1^R + c_1^R + c_2^R \leq L_1^R + L_1^R$, and hence $c_2^R \leq 2(L_1^R - c_1^R)$.

Case 2: If $k_2 = 1$, then $L_1^R = L_2^R$. The same conditions in Case 1 applies.

Case 3: If $k_2 > 1$. If Condition (5.3) does not hold, it means there is a task τ_j^R, $1 < j \leq n$, with execution time $c_j^R = 2(L_1^R - c_1^R) + \epsilon$ and $\epsilon > 0$. Given that the tasks are all released at time 0, then the task is released synchronously with task τ_1^R. First, assume that this task enters the processor before task τ_1^R. Consequently, it will be finished at time $t + 2(L_1^R - c_1^R) + \epsilon. The latest feasible start - time for \tau_1^R$ is $t + L_1^R - c_1^R$, however, because of the blocking caused by τ_j^R, task τ_1^R cannot enter the processor and therefore misses its deadline. On the other hand, in the best case where τ_j^R enters the processor right after τ_1^R, it will leave the processor at $t + c_1^R + 2(L_1^R - c_1^R) + \epsilon$ which is greater than the latest start-time of the next instance of τ_1^R i.e., $t + 2L_1^R - c_1^R$. This situation is shown in Fig. 5.5. As a result, no task may have an execution time greater than $2(L_1^R - c_1^R)$.

5.4 Experimental Results

We have implemented the two-level scheduler in a fully working middleware that will be the core of an open-source programming language for edge computing. This section presents the results from experiments that executed synthetic tasks

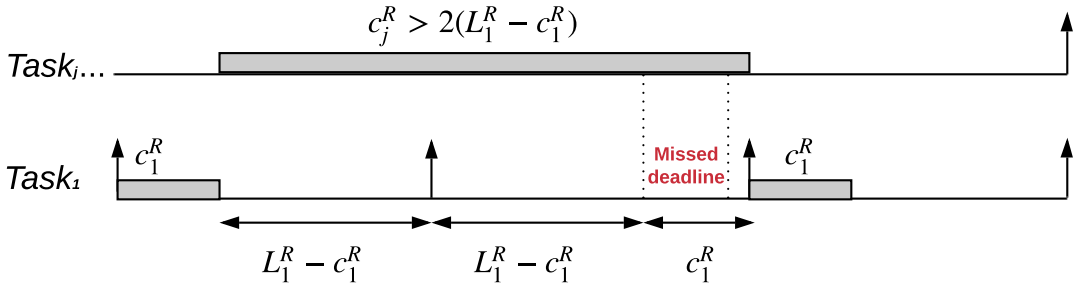

Figure 5.5 A counter example showing why in the task sets, maximum execution time of low priority tasks is bounded to $2(L_1^R - c_1^R)$

using the middle-ware. The experiments faithfully emulate edge scenarios with real-time tasks. We did not find a work that is doing real-time task scheduling at the middleware level for edge computing that we can use as a baseline, as this is an unexplored area to the best of our knowledge. Therefore, we performed many experiments to establish the feasibility of our approach using the prototype implementation.

Our results show that we addressed the problems identified in Section 5.1. The results from the experiments evaluating the L1-Stage are shown in Fig. 5.6 to Fig. 5.9. The L1-Stage is responsible for matching the real-time tasks to the edge servers. It uses two matching schemes: sole matching and grouped matching. In sole matching, an edge server arriving at the L1-Stage is matched to a workload (i.e., a set of real-time tasks to execute and the associated Slack servers). In the grouped matching, the edge servers are grouped, and the matching algorithm tries to find the best way of satisfying the requirements of the tasks given the capabilities of the edge servers.

We inject a workload of seven real-time tasks to the L1-Stage that has four edge servers. The L1-Stage matches the real-time tasks to edge servers and then the L0-Stage running in the edge servers gets the actual task instances for real-time, interactive (I), and batch tasks (B). The B/I tasks arrive in a Poisson process with the specified arrival rates. Fig. 5.6 shows the B/I Goodput (number of interactive tasks completed + throughput obtained for batch tasks) variation with

arrival rates of the B/I tasks for the L0 schedules derived using sole and grouped matching schemes. The grouped matches have higher locality of mapping the tasks to edge servers than the sole matching scheme, which explains the better performance obtained by the group matching. Fig. 5.8 shows the B/I Goodput variation with the number of edge servers. There we don't see a major difference between the two schemes because neither scheme can provide higher locality. This observation shows that the proposed fair selection algorithm provides high throughput for batch tasks, as required in Problem P-3. Using a fixed arrival rate, we can also observe as shown in Fig. 5.8, that the fair selection algorithm ensures a fair throughput for both interactive and batch tasks without any task type being starved.

Another interesting observation we can make is that the Goodput of the interactive (number of tasks completed before the soft deadline) drops as the arrival rate increases. This is because the higher arrival rates introduce execution delays and make the interactive tasks miss their deadlines. This problem does not arise with the batch tasks.

Once a particular L0 schedule is matched to an edge server, the task instances (real-time) start arriving at the local scheduler. The maximum throughput obtained (i.e., the number of real-time task instances completed) are shown in Figs. 5.7 and 5.9. We can observe that sole matching outperforms the grouped matching in terms of real-time throughput. This is because the sole matches an edge server with the columns in the task matching matrix that has high priority. The columns with more real-time tasks have high priority values.

Additional experiments performed at the L0-Stage are shown in Figs. 5.10 to 5.14. These experiments focus on the performance delivered by the L0-Stage to the interactive tasks. We use a measure called guarantee ratio (GR) which is the number of interactive tasks completed over the total number of interactive tasks that arrived to investigate the performance. The L1-Stage provides a *Normal schedule* and a *Greedy schedule* to the L0-Stage. These experiments examine the performance of these schedules and a hybrid scheme that L0-Stage implements,

where it selects the best schedule between the two based on the task mix present at the edge server. This hybrid scheme is shown as the "Combined" scheme in the figures.

In the hybrid scheme, the L0-Stage selects the best between the two schedules by measuring the sum of work that can be completed using the two different schedules. For interactive tasks, the contribution to the measure of work depends on whether the task was executed before the deadline or not. The hybrid scheduler assumes that the task arrivals have some locality (i.e., the arrival pattern in a scheduling cycle is related to the pattern in the previous cycles). Therefore, the hybrid scheduler evaluates the schedule in a cycle or a set of cycles and selects the winning schedule to deploy for the next few cycles.

The results show that the Combined scheme is better than Normal or Greedy in terms of the GR obtained by the interactive tasks. In these experiments, the arrival rates of the interactive tasks are varied while the processor utilization due to real-time tasks is varied from 0.2 to 0.75. It is important to observe that the processor utilization due to real-time tasks are set by the workload when it is matched at the L1-Stage. In addition, this processor utilization is an upper bound of CPU fraction that can be consumed by real-time tasks. If the real-time task instances do not arrive to occupy the scheduled slots, the actual CPU utilization can be lower than this value.

From the experimental results, we can observe that Combined is outperforming Normal and Greedy. Normal performs a real-time task execution as early as possible and greedy pushes it as late as possible (for the first period) along the time dimension. With Combined, we are enabling the L0-Stage to make the local decision as to when the flexibility of selecting between Normal and Greedy should be exercised. The results confirm that offering the L0-Stage that flexibility is better than L1-Stage selecting Normal or Greedy based on its observations. The Combined approach performs very well across the board when the utilization is both 0.3 and 0.75. In the figures, the deadlines of the interactive tasks are set to their worst-case execution times (WCET), c_i^I, and three times WCET. Figs. 5.12

and 5.13 show the effectiveness of our approach when the utilization is varied and arrival rates are at 0.4 and 0.6. Given that the utilization changes from 0.2 when the system is lightly loaded and 0.75 when it is highly loaded, the guarantee ratios are high and the Combined approach outperforms the two other scheduling approaches. In Fig. 5.14, we show the guarantee ratio variation when the deadlines of interactive tasks vary from small to large. The Combined approach gives a high guarantee ratio even with tight deadlines and outperforms the other approaches. The high guarantee ratio shows that a large fraction of interactive tasks are successfully scheduled by the algorithms that we proposed, addressing Problem P-2.

The L0-Stage is implemented as a user-level CPU scheduler. That is, the tasks are coroutines and a coroutine-to-coroutine switch can be performed entirely in the user space. While the context switch is very fast (mostly less than 35 nanoseconds in an Intel i7 running at 3.0 GHz), the fact that it runs entirely in the user-space calls for a deeper examination of the performance. In Figs. 5.15 to 5.17, we examine the scheduler jitter and study its variation with various parameters. The *scheduler jitter* is defined as the difference between the actual and scheduled starting times of an event. As it becomes difficult to control the precision of the scheduled starting times, the jitter increases and eventually leads to dropped starting events (i.e., the start events get dropped by the scheduler because it is way past the scheduled start time). From the figures, we can see that the jitter remains less than 50 us in these experiments. We ran the experiments in macOS (x86, 64bits), Linux (in Raspberry Pi4 (AArch64)), and Linux (x86, 64bits)) and the jitter remains less than 100 us in many experiments. This shows a user-level L0-Stage that is capable of running on a variety of platforms and scheduling coroutines with high precision.

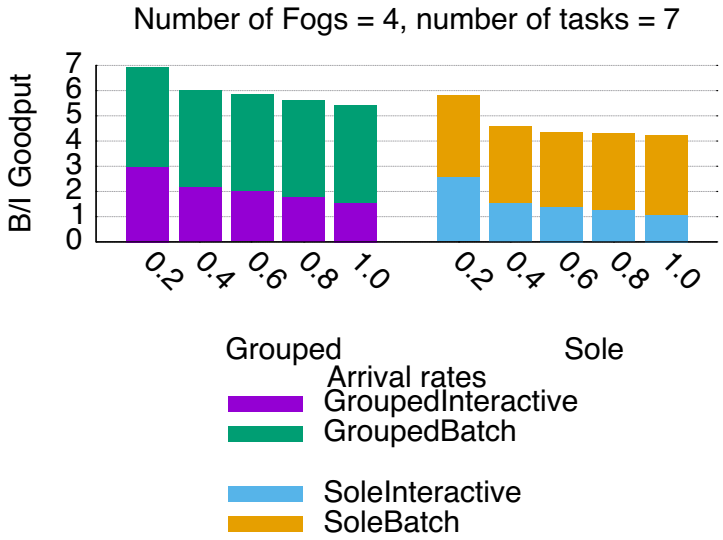

Figure 5.6 Varying arrival rates: work done with batch and interactive tasks

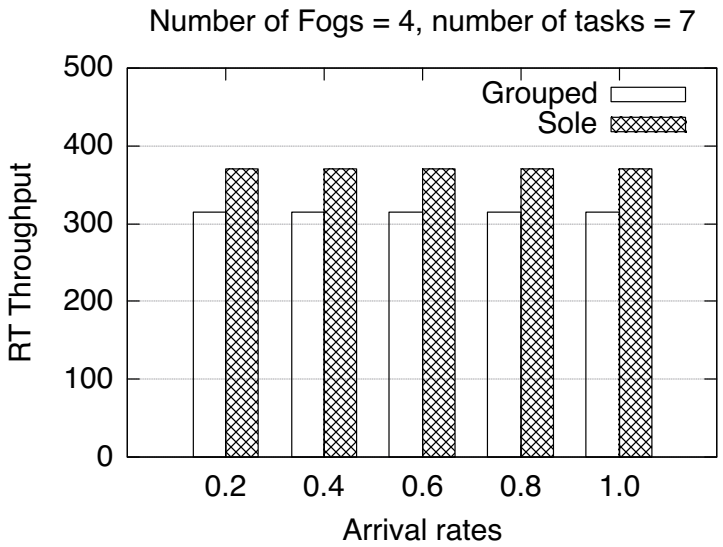

Figure 5.7 Varying arrival rates: work done with real-time tasks

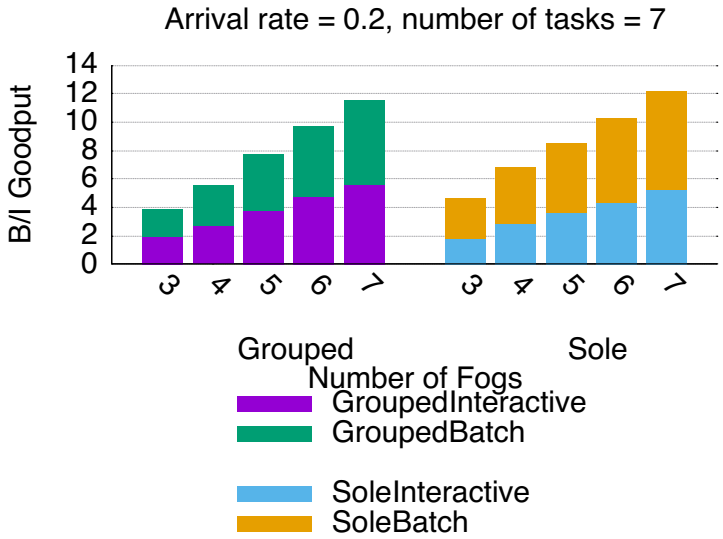

Figure 5.8 Varying number of edge servers: work done with batch and interactive tasks

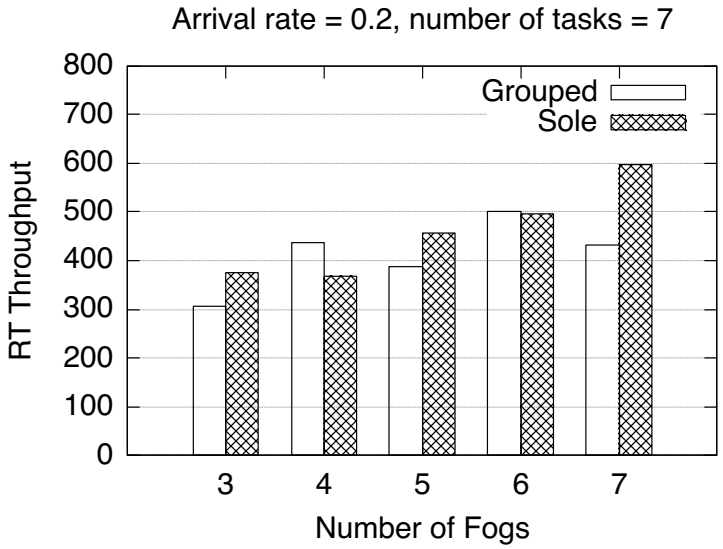

Figure 5.9 Varying number of edge servers: work done with real-time tasks

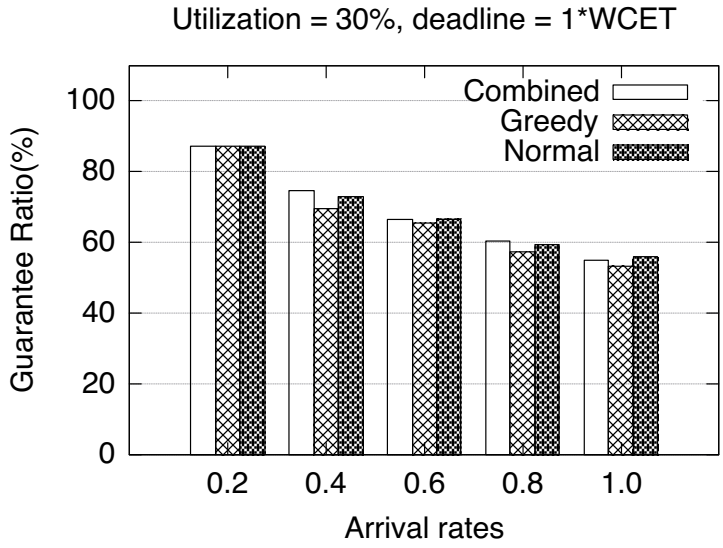

Figure 5.10 Varying task arrival rates. Utilization = 0.3, Deadline = 1 * WCET

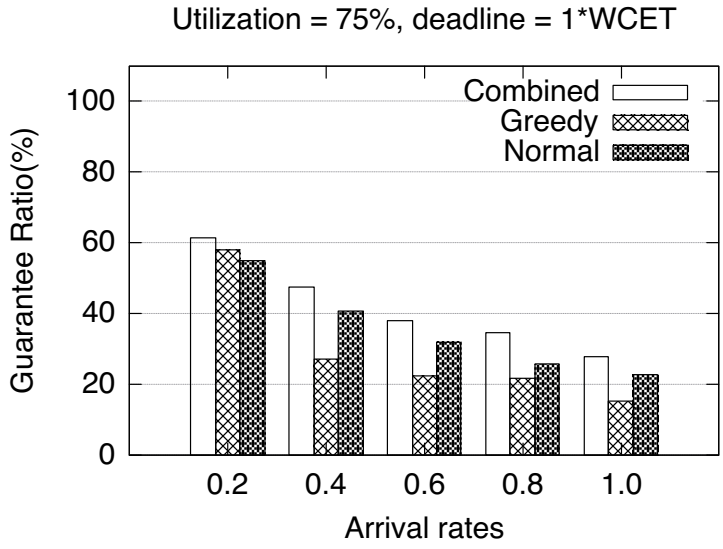

Figure 5.11 Varying task arrival rates. Utilization = 0.75, Deadline = 1 * WCET

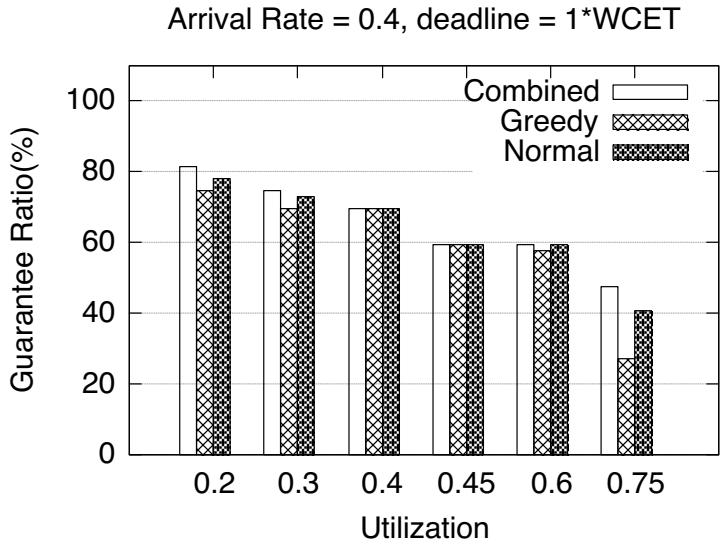

Figure 5.12 Varying utilization. Arrival rate = 0.4, Deadline = 3*WCET

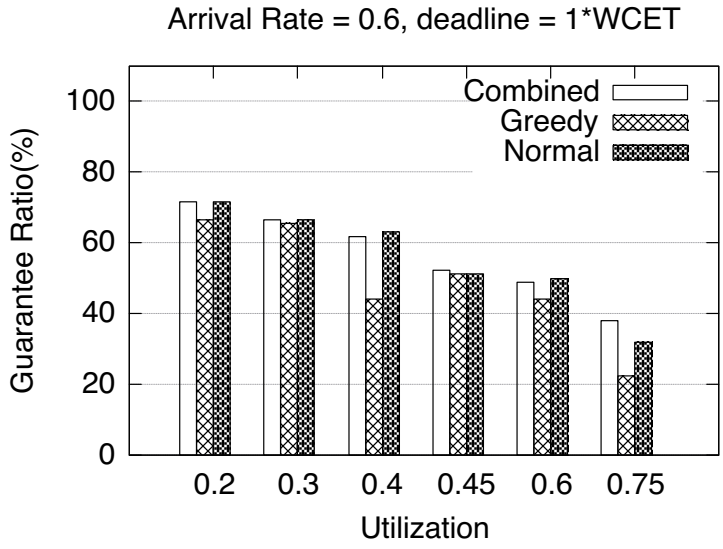

Figure 5.13 Varying utilization. Arrival rate = 0.6, Deadline = 3*WCET

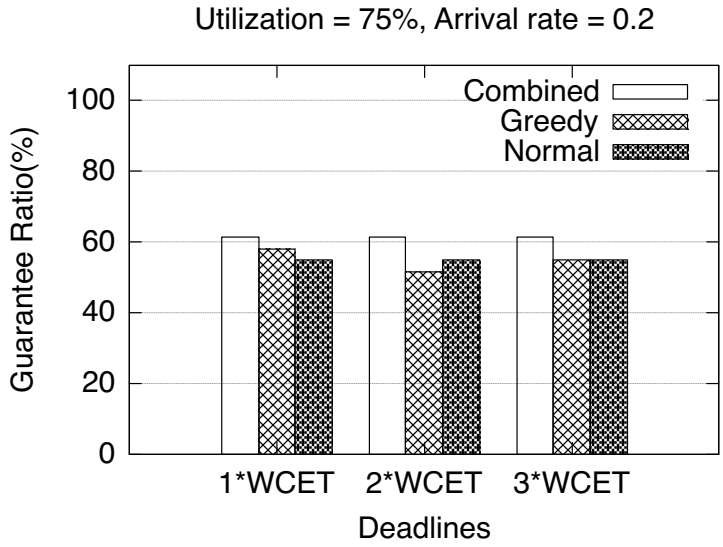

Figure 5.14 Varying deadlines with utilization = 0.75 and arrival rate = 0.2

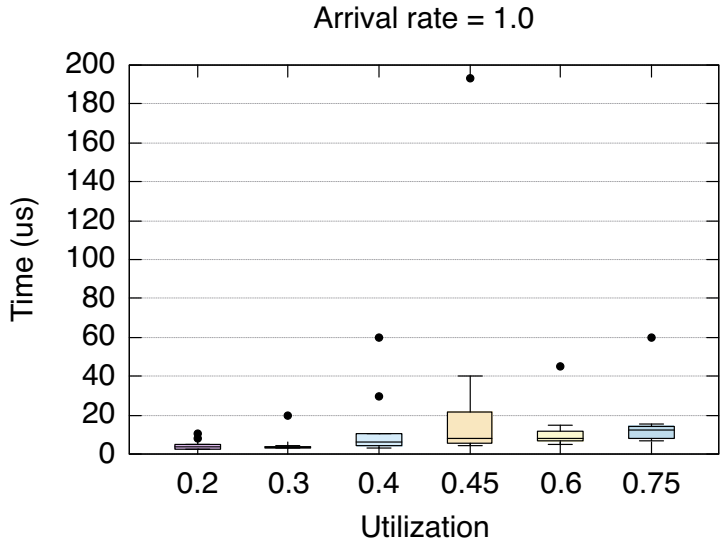

Figure 5.15 Varying utilization, arrival rate = 1.0

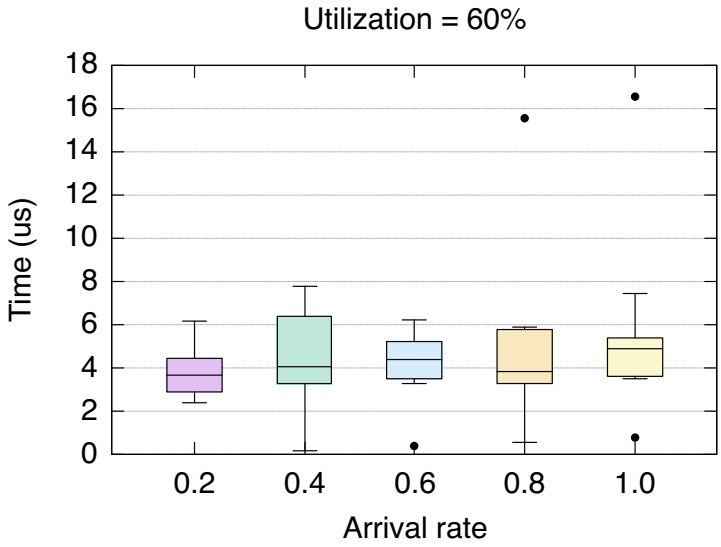

Figure 5.16 Varying arrival rate, utilization = 0.6

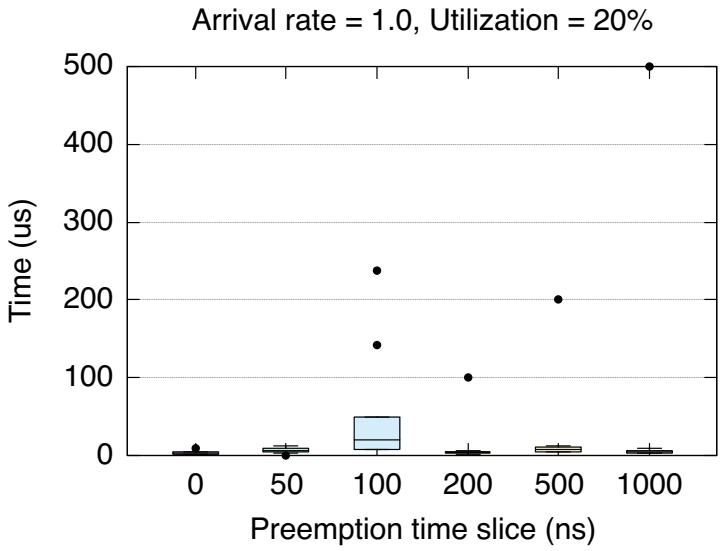

Figure 5.17 Varying preemption time slice, arrival rate = 1.0, utilization = 0.2

Chapter 6

Network-Aware Scheduling for Edge Computing Tasks

In the previous chapter, we considered applications having independent tasks and assumed that the edge resources are always available at a very short network link away from the origin of data (e.g., end device like a vehicle) and the network link connecting the origin to the edge has a very high capacity, so data can be transferred at high rates. While this assumption is nominally true, there can be many situations where this assumption is not true. We consider such scenarios in this chapter.

In this chapter, we consider distributed applications with dependent tasks that are across edge devices and edge servers. That is, some tasks run in edge devices (i.e., mobiles or vehicles) and others in the edge servers at or near the 5G base stations. The application is modelled as a directed acyclic graph (DAG) [160, 161, 162] with nodes denoting computations and edges denoting data communication and precedence constraints. The key objective of the edge task mapping problem is to optimally distribute the DAG across the devices and edge servers such that the makespan for executing the DAG is minimized. In this problem, the DAG is mapped onto a resource graph that has a time-varying configuration. That is, as the devices roam, the device-to-edge server association can change or the network link connecting the devices to the edge server can change in characteristics (i.e.,

from being fast to slow or vice-versa). We are focused on tackling the problems arising from variations in network link characteristics.

Using expected resource graph configurations [163, 164], we create a list of possible schedules for executing the tasks in the DAG at the edge. As the network conditions change, the different schedules among the precomputed schedules become the best for minimum makespan. The scheduling framework we develop here provides a runtime schedule switcher that deploys the best schedule among the precomputed ones as the execution proceeds.

The primary idea of schedule switching is deferring task executions. That is, if the expected network link conditions are not favourable, a particular task execution is deferred to a later time when the network link is favourable. A DAG with many tasks will have many schedules, so it is a matter of selecting the schedules that would defer the task pair, that is having a heavy network link usage and run task combinations without much network usage. Because the network conditions can change in the runtime, we need a way of switching the schedules on-the-fly [165].

The benefit of network-aware scheduling is twofold. First, network-aware scheduling improves the makespan of a single application by lessening the situations where a task would wait on a data transfer over a weak network link. Second, network-aware scheduling helps the network operator by enabling the operator to serve the network transfer requests in an application-specific manner. That is, using the network-aware scheduler, the network operator can provide a service to re-prioritize application tasks that would benefit both the applications and the provider.

We provide an overall framework for network-aware scheduling. As part of this framework, we developed a schedule switching algorithm and network-aware scheduling algorithm that minimizes the makespan of an application DAG. We simulated the algorithm using datasets[1] from 4G/LTE and representative appli-

[1]https://www.kaggle.com/anuragk240/mobile-data-speeds-of-all-india-during-march-2018

cation DAGs. The evaluation of the algorithms is carried out in two stages. In the first stage, we compare the performance of our network-aware algorithm with a network-unaware scheduling algorithm. In the second stage, we compare the performance of our speculatively chosen schedules (i.e., changing schedules with predicted network changes) with non-speculative schedules. The results indicate the benefits offered by network-aware scheduling for edge computing.

6.1 Problem Motivation

In this section, we motivate the need for network-aware scheduling with edge computing in two different ways. First, we illustrate why the shift from cloud to edge makes network-aware task scheduling a necessity. Second, we illustrate the performance benefit network-aware task scheduling could provide. Both of these illustrations make a case for investigating network-aware scheduling in the context of edge computing.

6.1.1 Need for Network-Awareness in Edge Computing

Consider a scenario where a device (i.e., a vehicle) is accessing a backend service. Let the backend service be located at the edge or cloud levels (two possibilities). If the backend service (e.g., a high fidelity mapping service to aid autonomous driving) is at a cloud server, the same backend server will be suitable for serving the device no matter what its physical location is. On the other hand, if the backend is hosted at an edge server it has to be different depending on the location of the vehicle. An example network configuration for this scenario is shown in Fig. 6.1. The first hop of the network (from the device) is always the access link. In this scenario, the access link is the 5G network. The edge server is accessible at the base station that terminates the access link or connected to that base station by a very fast link. The cloud servers are connected to the core network such that they are equally accessible from anywhere on the Internet.

In the example configuration shown in Fig. 6.1, we consider two scenarios: the vehicle in a high coverage zone and the vehicle in a low coverage zone. When the vehicle is in the high coverage zone, the latency of transmitting over the 5G link

is 5 ms (let us say). Conversely, when the vehicle is in the low coverage zone, the latency of the same link is 8ms (a 60% increase). The 5G link takes the packets from the device to the base station. From the base station, the packets are transported through multiple links to the core of the network where the cloud servers are located. Let us say the latency of crossing those links is another 25 ms and the total end-to-end latency of accessing the cloud server from the device is 30 ms when the vehicle is at the high zone. When the vehicle is at the low zone, this value changes to 33 ms. That is, the end-to-end latency increases by 10% due to the vehicle moving from high to low zone for the cloud scenario.

Because the percentage increase is high for the edge computing scenario, the effective performance of the application would degrade more from 5G link variation in the edge scenario and less in the cloud scenario. That is, with edge computing, it is more important to consider the network state.

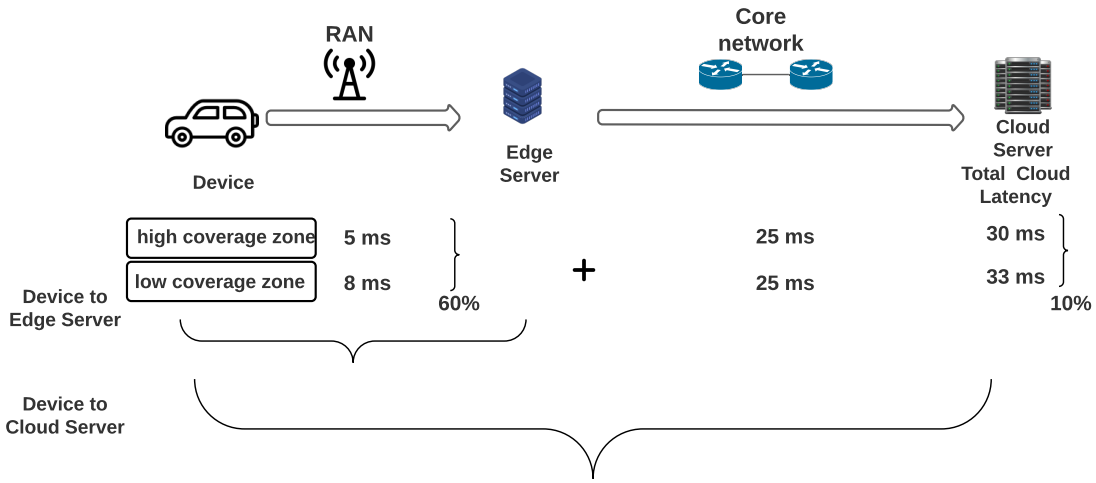

Figure 6.1 Network-Awareness in Edge computing vs Cloud computing

6.1.2 Need for Network-Aware Scheduling for Maximum Performance

Consider a smart vehicle running a Machine Learning (ML) application. Let us say that the ML application is capturing data from the vehicle and uploading it to

the edge server for training a neural network model. The neural network model is then downloaded into the vehicle to provide it intelligence that is context-specific (i.e., sensitive to the data being captured). The vehicle has two options to exercise: use an older neural network model or a newer model. Using the newer model needs uploading the captured data and downloading the trained model, which can incur a significant transfer overhead.

The ML application is represented by the directed acyclic graph (DAG) shown in Fig. 6.2. The tasks are split into vehicle tasks and edge server tasks. In the figure, vehicle tasks are on the left while the edge server tasks are on the right. The double arrows show transfers between the device (vehicle) and edge server sides. The single arrows show transfers in either the vehicle side or edge server side (i.e., transfers that would not use the 5G network link). As illustrated in the DAG, the vehicle can run a variety of tasks such as tasks for capturing data through cameras, preprocess the data, execute a local task, decode the model received from the edge, and deploy the model. Similarly, the edge server can also run a variety of different tasks such as data cleaning (i.e., another level of pre-processing on the data exported by the vehicle), model training, and packaging the model for pushing to the vehicle.

Now, consider a particular situation with the smart vehicles as shown in Fig. 6.3. Here, three vehicles A, B, C are running the ML application described above. The vehicles are all utilizing the network to send data to the edge servers (data push) and download data from the edge servers (model push). Depending on their location, the vehicles can be in a network with high, neutral or low signal strengths. Imagine that the vehicles are currently uploading the camera feeds captured to the edge servers and the data size is 100MB (megabytes). The bandwidth of the network at the high, neutral and low regions are respectively, 10MB/s, 5MB/s and 2MB/s. If all the vehicles are uploading the data in the low zone, it will take them 50s each to transfer the data. The overall throughput in the low network zone will be low as it takes a longer time for all the vehicles to complete their data transfer. Conversely, if all the vehicles are uploading the data in the high zone, it will take them 10s each to transfer the data.

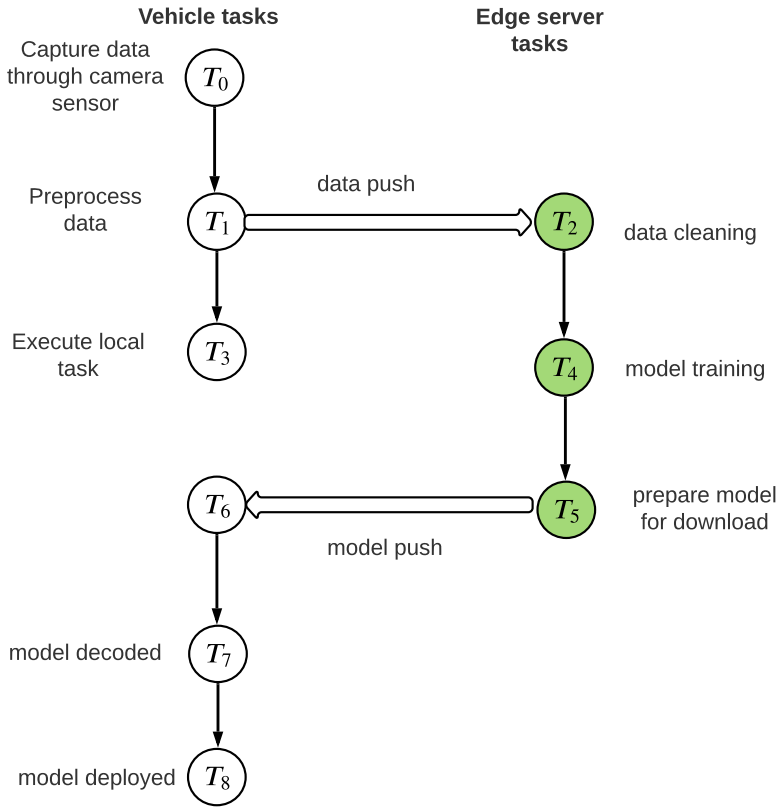

Figure 6.2 Example of a DAG model of ML application running in a smart vehicle

In Fig 6.3, Car A can delay its data transfer until it gets to a high zone if will take a longer time to get the results required, however, it can carry out local tasks that do not require data transfers when it is in the low network zone. Conversely, Car B can carry out its data transfers since it is in the high zone and schedule its local tasks when it gets to the neutral or low zones. By scheduling the application to defer data transfers when possible, the overall application performance can be improved.

In this particular ML application, the vehicle has the option of either using the old model or requesting a new model (we assume that there is some degradation in decision making if an older model is used, thus needing to pull a newer model).

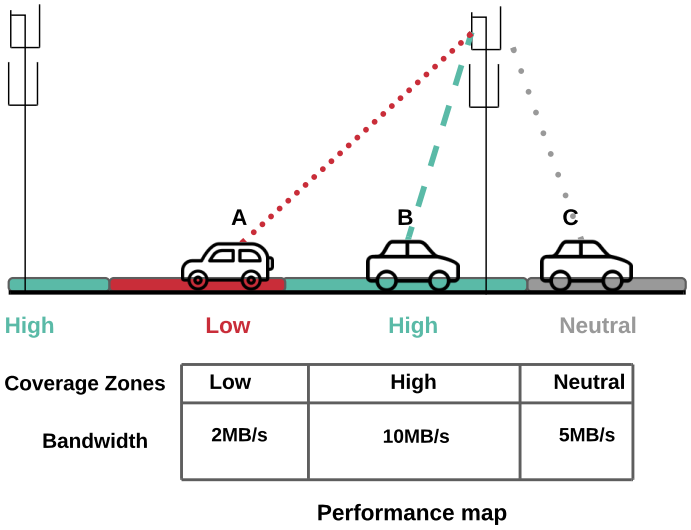

Figure 6.3 An illustration of an application DAG running in different network regions

Therefore, we can defer the data and model transfers until the vehicle has reached a high zone. This way, network throughput can be maximized. As a lot of vehicles will be on the road at any given time, we can make an overall schedule, so that high zone vehicles have priority to use the network.

6.2 Models

This section introduces the different models we use in this thesis. The architecture model shows how the components of the network-aware scheduling framework fit together. The application model describes the way applications are represented and distributed among the devices and edge servers in the network-aware scheduling framework. The transfer model explains how the network is captured in our work.

6.2.1 Architecture Model

The overall architecture of the network-aware scheduling for edge is shown in Fig. 6.4. It contains six components: (1) a physical layer (2) a sensing layer (3) a

neural network model (4) a performance map (5) an ahead-of-time scheduler and (6) a schedule fitness evaluator (SFE). Below, we describe the functions performed by each component.

- Physical layer: this is made up of the physical world and includes end devices (e.g., vehicles), edge servers, 5G links, applications running in the end devices, applications or services running on the edge servers, and other sensors. The applications are represented according to the application model.

- Sensing layer: it is composed of sensors that are placed in the physical layer. The sensors measure network performance such as the latency of accessing an application service from the end device while the service is hosted at an edge server. This measurement would be repeated for many device locations (as observed using a GPS sensor). The measurements obtained from these sensors provide a dataset that quantifies the variation of network performance with vehicular positions. That is, for selected coordinate positions a vehicle has visited, we would have measurements about the network performance at those points.

- Neural network model: it is a machine learning model that continuously observes the states of the network in the physical layer through the sensing layer. The model creates a performance map as the output. The performance map gives a predicted network performance for a physical location for a given time duration. The predicted performance value is an expected range (for example, a high range for bandwidth would indicate that the network is very likely to yield very high transfer rates). In this work, we use three ranges: High, Neutral, Low for the modelling of network performance. The neural network model itself will continuously learn using the measurements that are made by the vehicles in their sensing layer.

- Performance map: it is a lookup table created by the neural network model to provide a prediction of the expected network performance. The map is a time-dependent data structure because the predictions change as time increases. Also, the performance map should cover all the regions of interest to the vehicles that subscribe to the framework (i.e., the performance map

should provide predictions for all points a vehicle could go assuming we have a vehicular application). Using the performance map we should be able to get predictions over different time scales. For example, we should be able to obtain the predicted network state for the next 100 milliseconds or the next 10 seconds. The Ahead-of-Time Scheduler (described below) uses predictions over long time windows. The runtime schedule switcher would use predictions over short time windows.

- Ahead-of-Time (AoT) Scheduler: it is a component that runs in the cloud and takes the application DAG and performance map as the input. For an application DAG, it generates many valid schedules for executing the application on the networked system. AoT can use values from the performance over future time intervals to select a few of the many possible schedules or output all possible schedules. A backtracking algorithm similar to that given in [166] can be used to generate all possible valid schedules for the application DAG. In addition to the list of schedules, AoT also provides a 'Switch matrix' that specifies the switch points where the application execution can change from a schedule to another schedule. The idea is that as the network conditions change, the optimal schedule to use can change. The Switch matrix provided by AoT allows the runtime to change the schedule instead of sticking to a single schedule for the duration of a possibly long-running application. The Switch matrix will be explained in detail in Section 6.3.

- Schedule Fitness Evaluator (SFE): runs in the edge server and device at runtime. It evaluates the schedules given by the AoT scheduler along with the network states given by the performance map. It runs the proposed network-aware scheduling algorithm given in Section 6.3 to determine the best schedule to deploy in the edge device and server given the network configurations by the performance map. It is also responsible for triggering the schedule switching if necessary.

As shown in Fig. 6.4, the neural network model gets data from the sensor layer and learns a performance map for the network. The performance map is used with the device trajectory (e.g., estimated path of the vehicle) and the output is

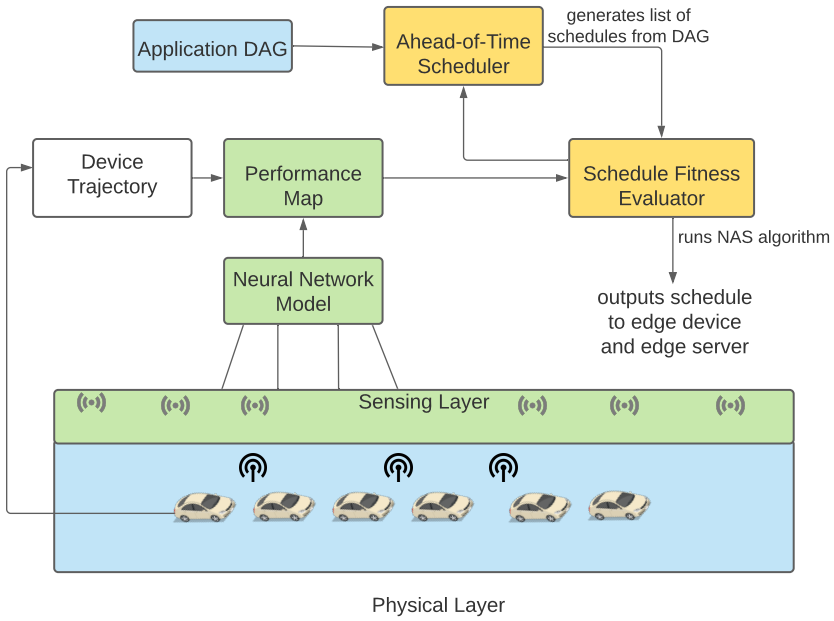

Figure 6.4 Architecture model

fed into the SFE evaluator. The AoT scheduler creates a list of schedules from the application model and it feeds the schedule into the SFE evaluator. The SFE evaluator uses the schedules from the AoT scheduler and the network models from the performance map to obtain the schedule the edge device and server should deploy. There is a feedback loop from the SFE evaluator to the AoT scheduler. The feedback loop gives the AoT scheduler a subset of the best performing schedules after evaluation. The schedules can then be used to obtain the switch matrix. The switch matrix is employed to enable the runtime scheduler switch to the better performing schedules at runtime if there is an anticipated network change.

6.2.2 Application Model

An application is modelled as a directed acyclic graph (DAG). DAG is a directed graph with no cycles and it consists of a set of vertices and directed edges, each of which connects one vertex to the other. DAG model is described by a two tuple $G = (T, E)$, where T is the task set $T = (T_1, ..., T_N)$ consisting of different associated tasks of the application and E is the set of edges between tasks denoting the

execution order and communication between two adjacent tasks, which specifies their dependencies. An edge (i, i') between task nodes i and node i' represents that task T_i is the predecessor of task T_i' and therefore task T_i should complete its execution before task T_i'.

An example DAG is shown in Fig. 6.2. The tasks are executed on computational resources, which can either be an edge device or an edge server. The inter-dependency among the different tasks of the application is captured using the DAG model. The weight attached to each task T_i represents the computation requirement, described by c_i. The weight attached to each edge represents the communication cost in terms of data size between two tasks T_i and T_i', denoted as $C_{i,i'} = data(T_i, T_i')$. The data transfer between two tasks is only required when the two tasks are assigned to different computational resources, i.e., the communication cost is negligible when two tasks are executed on the same computational resource. A DAG model can produce multiple ordering of the tasks. Our goal is to determine the different ordering of task instances to be deployed depending on the network conditions to minimize the execution time.

In our work, we focused on batch mode for data transfers, i.e., once a task has finished processing its input data, it sends the results to the successor task for processing. However, our application model can be extended to consider stream processing where there is a continuous flow of data between tasks. That is, a task will continuously send data to its successor task as it is being processed piece-by-piece. We can handle data streaming with our current model by splitting the data into chunks similar to the idea presented in the HTTP Live Streaming (HLS) protocol [167]. High quality chunks can be delivered to the successor task when the predecessor task is transferring data in a high network zone, while low quality chunks can be delivered when the transfer is done in a low network zone.

6.2.3 Transfer Model

We model our network as having different states as shown in Fig. 6.3. The performance map in the figure shows the High, Neutral and Low network regions given

the locations of the vehicle. The performance map gives different communication bandwidth between the vehicles and the edge servers in the different regions. The average communication bandwidth among computational resources executing task T_i and T_i' is denoted as $B_{\varphi_i,\varphi_i'}$. We denote φ_i as the resource for executing task T_i. Then, the transfer time - $\Gamma_{i,i'}$ of task i on edge (i, i') is given as

$$\Gamma_{i,i'} = \begin{cases} \frac{C_{i,i'}}{B_{\varphi_i,\varphi_i'}}, & \text{if } \varphi_i \neq \varphi_i' \\ 0, & \text{if } \varphi_i = \varphi_i' \end{cases} \tag{6.1}$$

The above equation is used to compute the transfer time if the tasks between an edge are located on different resources. As seen, if the tasks are located on the same resource, then the communication cost is 0. We classify the communication links between tasks into three types as follows:

- inter-transfer communication links: These are data transfers happening between two tasks such that they are executing on different resources and there is an execution dependency between the two tasks. That is, a task T_i is executed on an edge server and task T_i' is executed on an edge device. However, task T_i must complete its execution before task T_i' can start executing. In this work, we are focused on inter-transfers since they incur data transfer costs on the network and therefore have to be taken into consideration when scheduling the tasks.

- intra-transfer communication links: These are data transfers happening between two tasks on the same resource type. That is, both tasks T_i and T_i' are executed on an edge server (or an edge device). However, task T_i' is executed after the completion of task T_i. The communication cost is assumed to be 0 since the data transfer is happening on the same resource and does not require the network.

- non-transfer links: These are tasks that do not have any data transfer between them and therefore no edges exist between the tasks.

In Fig. 6.2, inter-transfer communication links are denoted with the double arrows and intra-transfer communication links are denoted with single arrows.

6.3 Scheduling Algorithms

We propose three scheduling algorithms in this work. The first two are part of our architecture model and the last one is implemented as a baseline to evaluate our proposed algorithm. The algorithms are the network-aware algorithm, the switching algorithm and network-unaware algorithm, respectively. They are described in detail in the next section.

6.3.1 Network-Aware Scheduling Algorithm

We propose a network-aware scheduling (NAS) algorithm to select the best schedule ordering among the many available orderings. The algorithm considers the network conditions when deploying the best schedule. The schedule fitness evaluation (SFE) component in the architecture runs the NAS algorithm and outputs the schedule to deploy to the computational resources.

In the NAS algorithm, we evaluate the schedules given by the AoT scheduler. The AoT scheduler gives a topological ordering of the DAG, which produces many different orderings - schedules. The NAS algorithm evaluates the selected schedules against the network model given by the performance map and computes their completion times. The output of the NAS algorithm is the schedule with the best completion time. At runtime, when the schedule is deployed, a change in network conditions necessitates a re-evaluation of the schedules. However, the switching algorithm discussed in the next section is employed to determine which schedules should be re-evaluated as tasks have been executed. So the new schedule needs to consider the previously executed tasks of the DAG, so it does not violate precedence constraints.

The NAS algorithm given in Algorithm 6 takes as input a set of schedules and computes the completion time for each of the schedules under a given set of network states. A task expected start time (EST) and expected finish time (EFT) is defined as follows: if a task is the first task in the schedule, then its EST is 0, otherwise a task EST is given as:

Algorithm 6: Network-Aware Scheduling (NAS) Algorithm

```
1  Input: S - list of schedules, I - set of DAG inter-transfer edges, N - Network
     data, T - Edge device trajectory
2  Output: Schedule s
3  function Nas(S, I, N, T)
4    variables: makespan
5    compute the makespan of each schedule as follows
6    for each schedule s in S
7      for each edge in the schedule s
8        if edge is not in set I
9          compute the EFT for the destination task using equation 6.3
10         makespan = makespan + EFT
11       else
12         get the position of the edge device from T
13         get the network state at that position
14         while the edge datasize > 0 do
15           compute the transfer time using equations 6.1
16         compute the EFT of the destination task using equation 6.3
17           makespan = makespan + EFT + transfer time
18   return the schedule with the least makespan
```

$$EST(T_i') = EFT(T_i) + \Gamma_{i,i'} \tag{6.2}$$

The EFT of a task is given as:

$$EFT(T_i) = EST(T_i) + c_i \tag{6.3}$$

At compile time, the schedule with the minimal completion time is selected for execution. In the algorithm, the selected schedules completion times are computed by taking the EST and EFT of each task in a given schedule. The completion time of a schedule is the EFT of the last task in that schedule. The task with the minimal completion time is given as the output of the algorithm. Ties are broken arbitrarily if more than one schedule has the minimal completion time.

6.3.2 Schedule Switching Algorithm

The switching algorithm is developed as a means to obtain an efficient schedule at runtime. Our application model employs different ordering of the task instances of the DAG. A continuously running application will deploy the optimal schedule at different time points. Consider that the network condition is constant, i.e., no variation in the access links, then we can deploy a single ordering of the DAG. However, since the network condition can vary at different points in time, we need to consider the ordering that delivers the best performance to the application at that point in time.

The schedule switching algorithm is shown in Algorithm 7. It takes a list of schedules along with the currently running schedule and generates a matrix that shows the schedules that can be switched to at different switch points. The switch points are the points in the task ordering where schedules can be switched.

The algorithm is used to determine the appropriate schedules that can be switched to at runtime. It considers tasks that have already been executed up until that switch point. The schedule that is being switched to needs to have the same subset of tasks as the current schedule that is running up until that switch point. The algorithm gives a matrix that shows the different schedules that can be switched to at different switch points.

The switching algorithm can be executed in the AoT scheduler component to determine the schedules that can be switched to. It can also be run at runtime to determine the appropriate schedules to switch to. However, running it in the AoT scheduler ahead of time saves time and only a lookup on the matrix needs to be done when a network change is detected.

6.3.3 Network-Unaware Scheduling Algorithm

We develop a network-unaware algorithm (NUS) as a baseline to evaluate network-aware algorithm. In the network-unaware scheduling algorithm, the schedules are evaluated without being aware of the network.

Algorithm 7: Switching Algorithm

```
1  Input: S - list of schedules
2  Output: A - Switch Matrix
3  function SwitchMatrix (S)
4    for each schedule s_i ∈ S do
5      for each task position t ∈ s_i do
6        A[i,t] = Switch(s_i, S, t)
7     return A
8  function Switch(c, S, t)
9    variables: c_t - set of tasks in current schedule c, s_t - set of tasks in a given
       schedule s, C: empty array
10   c_t = getTasks(c, t)
11   for each schedule s ∈ S do
12     if c ≠ s do
13       s_t = getTasks(s, t)
14       if c_t is equal to s_t do
15         add schedule s to C
16   return C
17 function getTasks(s, t)
18   variables: T: empty set
19   for i = 0 to t − 1 do
20     add task s[i] to T
21   return T
```

Algorithm 8: Network Unaware Scheduling (NUS) Algorithm

```
1  Input: S - list of schedules, I - set of DAG inter-transfer edges
2  Output: Schedule s
3  function Nus(S, I)
4    variables: count: integer, transfers: array of integers for each schedule
5    for each Schedule s in S
6      for each edge w(v_i, v_{i+1}) in the schedule s
7        if edge is in set I
8          increment count by 1
9        else
10         add count to transfers and reset count
11     assign the highest count in transfers array to schedule
12   return the schedule that has the least length of the longest sequence of
     consecutive transfers
```

Here, the objective is to choose the schedule that minimizes the number of inter-transfers among the tasks. We assume that the network has a constant bandwidth, therefore, the data transfer time is not affected. Ties are arbitrarily broken if more than one schedule has the minimal number of inter-transfer communication links.

The algorithm is given in Algorithm 8. In the algorithm, we count the number of sequential inter-communication links for each schedule. Each schedule is assigned the highest number of such links. The algorithm then returns the schedule that has the fewest number of sequential transfers among all schedules that are evaluated.

6.3.4 Speculative Scheduling

In this thesis, we introduce speculative scheduling as selecting the 'best' schedule in advance based on the predicted network states. By using the switch matrix, we can change our schedules to reflect the changing network conditions.

At compile time, we are given the states of the network and the trajectory of the edge device that will be running the application. The NAS algorithm is employed to select the best schedule to deploy. At runtime, when a change in network state (different from what was used at compile time) is anticipated, the current schedule

is evaluated against the alternate schedules provided by the AoT along with the switch matrix. The NAS algorithm is employed to evaluate the partial tasks in the schedules using the anticipated network states, and the schedule with the minimal completion time is selected for execution and a switch is performed at runtime.

If the network change prediction is correct, we deploy the new[2] schedule selected by the NAS algorithm. On the other hand, if there was no network change, the application is deployed with the current running schedule. Either way, the performance delivered to the application is maximized.

6.4 Example Walk-through

Given the application DAG in Fig. 6.5, we will illustrate the algorithms with the figure. A subset of the exhaustive schedules generated using the topological sort algorithm is given below:

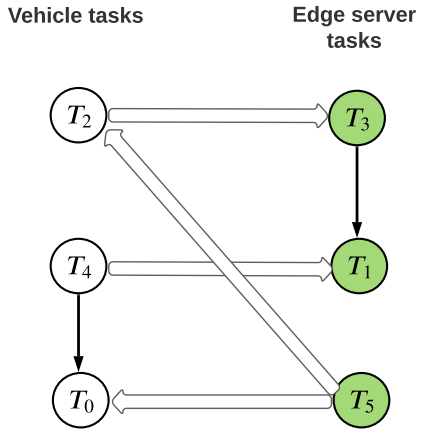

Figure 6.5 An Application DAG

1. $T_4, T_5, T_0, T_2, T_3, T_1$
2. $T_4, T_5, T_2, T_3, T_0, T_1$
3. $T_4, T_5, T_2, T_3, T_1, T_0$

[2]the schedule given by the NAS algorithm can be the current running schedule if that was the best schedule to deploy.

4. $T_5, T_2, T_3, T_4, T_0, T_1$

5. $T_5, T_2, T_4, T_0, T_3, T_1$

Using the five generated schedules, the switching algorithm in Algorithm 7 generates the switch matrix which is shown in Table 6.1. The columns are the switch points which are represented by the number of tasks in a schedule (or the DAG) and the rows are the schedules. At each switch point from 1 to n, where n is the number of tasks, we obtain the set of schedules that we can switch to at that switch point. The schedules must have the same set of tasks up until that switch point.

Table 6.1 Switch Matrix

Schedules/Switch points	1	2	3	4	5	6
S1	{}	{S2, S3}	{S2, S3}	{}	{S5}	{}
S2	{}	{S1, S3}	{S1, S3}	{S3}	{S3, S4}	{}
S3	{}	{S1, S2}	{S1, S2}	{S2, S5}	{S2, S4}	{}
S4	{}	{S5}	{S5}	{}	{S2, S3}	{}
S5	{}	{S4}	{S4}	{S2, S3}	{S1}	{}

Given the following network states $Z = HLHL$ and their respective duration as $z^d = \{5, 2, 4, 5\}$. A device is in the high network region from time 0 - 5, then moves into the low network region from time 5 - 7 and is back in the high network region from time 7 - 11. It goes into the low network region from time 11 - 16.

We will use the NAS algorithm to compute the completion time of the schedules. The execution time of the tasks as shown in Fig. 6.5 is given by the set $c_i = \{3, 3, 4, 1, 2, 2\}$ where the nodes are represented by $T = \{T_0, T_1, T_2, T_3, T_4, T_5\}$. The data bytes to be transferred on an inter-transfer link is 10 megabytes for our illustration purpose. The data transfer time is denoted as $\Gamma_{(T_i, T_i')}$. The bandwidth in the high and low network states are 10 MB/s and 5MB/s respectively.

We obtain the schedules completion times as follows:

1. S1 tasks are $T_4, T_5, T_0, T_2, T_3, T_1$. The EST and EFT for each task in the schedule is computed. $\text{EST}(T_4) = 0$ being the first task. $\text{EFT}(T_4) = 0+2 = 2$. $\text{EST}(T_5) = \text{EFT}(T_4) + \Gamma_{(T_4, T_5)}$. Since it is a non-transfer communicating link, the time is 0, so $\text{EST}(T_5) = 2 + 0 = 2$. $\text{EFT}(T_5) = 2 + 2 = 4$.

$\text{EST}(T_0) = \text{EFT}(T_5) + \Gamma_{(T_5,T_0)}$. Since the link between T_5 and T_0 is an inter-transfer communication link, there is a data transfer time involved. At time 4, the EFT of task T_5, the network is in the 'High - H' state since H occurs from time 0 - 5.

Using equations 6.1, we have

$$\Gamma_{T_5,T_0} = 10/10 = 1$$

The data transfer completes at the end of the network being in the high network state. Therefore, $\text{EST}(T_0) = 4 + 1 = 5$ and $\text{EFT}(T_0) = 5 + 3 = 8$.
$\text{EST}(T_2) = \text{EFT}(T_0) + \Gamma_{(T_2,T_0)}$. Given that it is a non communicating link, the transfer time is 0, therefore the $\text{EST}(T_2) = 8 + 0 = 8$ and the $\text{EFT}(T_2) = 8 + 4 = 12$.
$\text{EST}(T_3) = \text{EST}(T_2) + \Gamma_{(T_2,T_3)}$. Since it is an inter-transfer communicating link, we compute the transfer time using equation 6.1 as follows: At time 12 ($\text{EFT}(T_2)$), the network is in the Low network state, therefore the data transfer time is

$$\Gamma_{T_2,T_3} = 10/5 = 2$$

So, $\text{EST}(T_3) = 12 + 2 = 14$ and $\text{EFT}(T_3) = 14 + 1 = 15$.
Finally, $\text{EST}(T_1) = \text{EFT}(T_3) + \Gamma_{(T_3,T_1)}$. Since it is an intra-transfer communication link, the transfer time is 0. Therefore, $\text{EST}(T_1) = 15 + 0 = 15$ and $\text{EFT}(T_1) = 15 + 3 = 18$. So for Schedule S1, the completion time is 18.

2. S2 tasks are $T_4, T_5, T_2, T_3, T_0, T_1$. $\text{EST}(T_4) = 0$ being the first task. $\text{EFT}(T_4) = 0 + 2 = 2$. $\text{EST}(T_5) = \text{EST}(T_4) + \Gamma_{(T_4,T_5)}$. Since it is a non-transfer communicating link, the time is 0, so $\text{EST}(T_5) = 2+0 = 2$. $\text{EFT}(T_5) = 2+2 = 4$.

   $\text{EST}(T_2) = \text{EFT}(T_5) + \Gamma_{(T_5,T_2)}$. Since the link between T_5 and T_2 is an inter-transfer communication link, there is a data transfer time involved. At time 4, the EFT of task T_5, the network is in the 'High - H' state since H occurs

from time 0 - 5, so the data transfer time is

$$\Gamma_{T_5,T_2} = 10/10 = 1$$

EST(T_2)= $4 + 1 = 5$ and EFT(T_2) = $5 + 4 = 9$.

EST(T_3) = EFT(T_2) + $\Gamma_{(T_2,T_3)}$. Since it is an inter-transfer communication link, the transfer time is computed using 6.1. The network state at time 9 is the High network, so the data transfer time is 1s. Therefore, EST(T_3) = $9 + 1 = 10$ and EFT(T_3) = $10 + 1 = 11$.
EST(T_0) = EFT(T_3) + $\Gamma_{(T_3,T_0)}$. Since it is a non-transfer communication link, the data transfer is 0, therefore, EST(T_0) = $11 + 0 = 11$ and EFT(T_0) = $11 + 3 = 14$.
EST(T_1) = EFT(T_0) + $\Gamma_{(T_0,T_1)}$. Since it is a non-transfer communicating link, the data transfer time is 0. Therefore, EST(T_1) = $14 + 0 = 14$ and EFT(T_1) = $14 + 3 = 17$. So the schedule completion time for S2 is 17.

3. S3 tasks are $T_4, T_5, T_2, T_3, T_1, T_0$. It is very similar to schedule S2 until the 5th switch point where T_1 is to be scheduled [3]. At that point, EFT(T_3) is given as 11.

   EST(T_1) = EFT(T_3) + $\Gamma_{(T_3,T_1)}$. Since it is an intra-transfer communication link, the data transfer is 0, therefore, EST(T_1) = $11 + 0 = 11$ and EFT(T_1) = $11 + 3 = 14$.
   EST(T_0) = EFT(T_1) + $\Gamma_{(T_1,T_0)}$. Since it is a non-transfer communicating link, the data transfer time is 0. Therefore, EST(T_0) = $14 + 0 = 14$ and EFT(T_0) = $14 + 3 = 17$. So the schedule completion time for S3 is also 17.

4. S4 tasks are $T_5, T_2, T_3, T_4, T_0, T_1$. EST($T_5$) = 0 and EFT($T_5$) = $0 + 2 = 2$. EST(T_2) = EFT(T_5) + $\Gamma_{(T_5,T_2)}$. Since it is an inter-transfer communication link, we compute the transfer time. The network state at time 2 is High - H. So the data transfer time is $10/10 = 1$. Therefore, EST(T_2) = $2 + 1 = 3$ and EFT(T_2) = $3 + 4 = 7$.

[3] We will not repeat the steps as it is very similar to Schedule S2

$\text{EST}(T_3) = \text{EFT}(T_2) + \Gamma_{(T_2,T_3)}$. Since the transfer is an inter-transfer communication link, the transfer time is computed. At time 7, the network is in the High state, so the transfer time is 1. Therefore, $\text{EST}(T_3) = 7 + 1 = 8$ and $\text{EFT}(T_3) = 8 + 1 = 9$.
$\text{EST}(T_4) = \text{EFT}(T_3) + \Gamma_{(T_3,T_4)}$. Since T_4 is a non-transfer communication link, the transfer time is 0. So, $\text{EST}(T_4) = 9 + 0 = 9$ and $\text{EFT}(T_4) = 9 + 2 = 11$.

$\text{EST}(T_0) = \text{EFT}(T_4) + \Gamma_{(T_4,T_0)}$. Since it is a non-transfer communication link, the transfer time is 0. So, $\text{EST}(T_0) = 11+0 = 11$ and $\text{EFT}(T_0) = 11+3 = 14$.

$\text{EST}(T_1) = \text{EFT}(T_0) + \Gamma_{(T_0,T_1)}$. Since it is a non-transfer communication link, the transfer time is 0. So, $\text{EST}(T_1) = 14 + 0 = 14$ and $\text{EFT}(T_1) = 14 + 3 = 17$.

The computation time of the Schedule S4 is 17.

5. S5 tasks are $T_5, T_2, T_4, T_0, T_3, T_1$. Note that it is similar with S4 up until time-point 3. So we will evaluate from the third task. At that point, $\text{EFT}(T_2)$ is given as 7. $\text{EST}(T_4) = \text{EFT}(T_2) + \Gamma_{(T_2,T_4)}$. Given that, it is a non-transfer communication link, the transfer time is 0. So, $\text{EST}(T_4) = 7 + 0 = 7$ and $\text{EFT}(T_4)$ is $7 + 2 = 9$.
$\text{EST}(T_0) = \text{EFT}(T_4) + \Gamma_{(T_4,T_0)}$. Since it is an intra-transfer communication link, the transfer time is 0. Therefore, $\text{EST}(T_0) = 9 + 0 = 9$ and $\text{EFT}(T_0) = 9 + 3 = 12$.
$\text{EST}(T_3) = \text{EFT}(T_0) + \Gamma_{(T_0,T_3)}$. Since it a non-transfer communication task, the transfer time is 0. Therefore, $\text{EST}(T_3) = 12 + 0 = 12$ and $\text{EFT}(T_3) = 12 + 1 = 13$.
Finally, $\text{EST}(T_1) = \text{EFT}(T_3) + \Gamma_{(T_3,T_1)}$. It is an intra-transfer communication link, therefore, its transfer time is 0. So, $\text{EST}(T_1) = 13+0 = 13$ and $\text{EFT}(T_1) = 13 + 3 = 16$. The completion time for Schedule S5 is 16.

Ranking the completion times of all schedules, Schedule S5 gives the minimal completion time of 16 and is, therefore, the output of the NAS algorithm.

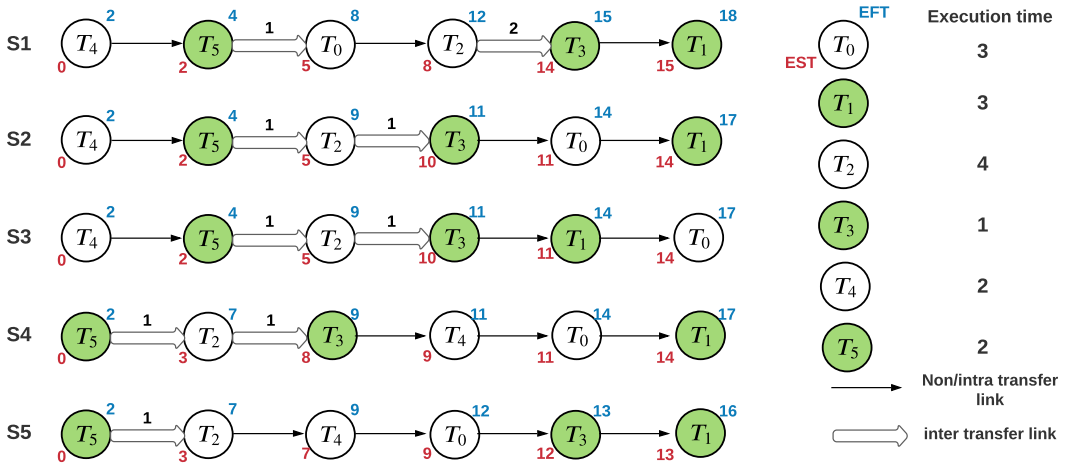

Figure 6.6 Illustrating the example walk-through

The example walk-through is illustrated using a diagram in Fig. 6.6.

To illustrate the network-unaware scheduling algorithm, we find the number of inter-transfers communication links in each schedule. We observe schedules that have the minimum number of consecutive inter-transfer links. In our example, schedules S2, S3 and S4 all have 1 consecutive inter-transfer link, so the algorithm returns any schedule among the three. For illustration purposes, the NUS algorithm returns schedule S4 because it arbitrarily selects a schedule if there is a tie.

6.5 Experiments & Simulation Results

6.5.1 Metrics Definition

To evaluate the effectiveness of the network-aware scheduling strategy, we perform comprehensive simulations. Two different metrics were used in the evaluations. The first is the makespan which is the turnaround time of executing the application DAG and the second is the reward which is the utility derived by the service provider in executing the application DAG. The makespan is the sum of the computation and communication times involved in executing the application, whereas the reward involves only the communication times.

The reward is defined as follows: let x_{z_i} be the dwell time and B_{z_i} be the bandwidth in network region z_i. Let p be a constant value (measured in \$s) and D be the total data size at a particular edge.

The total transfer time T_t across regions is derived as follows:

$$\begin{aligned} &while\ (D > 0): \\ &\quad z_i = \text{ probe device location} \\ &\quad T_t = T_t + \frac{min((B_{z_i} * x_{zi}), D)}{B_{zi}} \\ &\quad D = D - min((B_{z_i} * x_{zi}), D) \end{aligned} \tag{6.4}$$

The reward U is then given as follows:

$$U = \frac{p}{T_t} \tag{6.5}$$

where the reward is inversely proportional to the time spent carrying out the data transfer. In the high zone, the data transfer is done in less time incurring lower service cost and higher rewards for the service provider, whereas in the low zone, more time is spent transferring data incurring higher service cost and lower rewards.

6.5.2 Results

A simulator was implemented in Java and the task graphs were generated randomly using the following input parameters:

- Task size in the graph(v): the value v is assigned from the set {5, 10, 15}. The graphs produced are shown in Fig. 6.8 as DAG5, DAG10, and DAG15. The number of edges for each graph are 7, 23 and 47 while the number of schedules generated are 3, 54 and 913 respectively.

- Communication-to-Computation Ratio (CCR): the graph's CCR is the ratio of the average communication cost to the average computation cost. The CCR measure indicates whether a DAG is communication-intensive, computation-intensive, or balanced. We used the following values of CCR = { 0.1, 1.0, 10}, where 0.1 is for computationally intensive DAGs and

communication is of low significance, 1 is for balanced DAGs and 10 is for communication-intensive DAGs where the significance of the communication process is high.

The simulation setup also defines the execution time of each task that is obtained from a cloud workload[4] distribution. In addition, the system also defines a taxi mobility dataset obtained from CRAWDAD [168] and a network bandwidth dataset obtained from India[5].

The mobility and network setup is illustrated in Fig. 6.7. A sample car running an application in the city is shown in the figure. The red numbers represent the network data speed in MBps obtained at different locations in the city, while the blue numbers represent the location of the sample car and the black lines show its anticipated trajectory path in the city. The location numbers are obtained from the longitude and latitude coordinates in the mobility dataset and are transposed into a grid with x and y coordinates. We convert and store the grid coordinates as a one-dimensional array in the simulation. The experiments are carried out on a machine with the following hardware specification: it has 16 GB of main memory, a quad-core Intel Core i7 @ 3.1 GHz CPU.

We use different DAG configurations to evaluate the Network-Aware Scheduling (NAS) and Network-Unaware Scheduling (NUS) algorithms. The experiments were repeated 100 times and the experimental results were recorded. In Fig. 6.9, we compare the makespan of the network-aware scheduling algorithm against the network-unaware scheduling algorithm for the different DAG configurations. NAS algorithm provides better application performance across the three application DAGs as it offers an improved execution time. End-users benefit from faster response times compared to the NUS algorithm.

Fig. 6.10 shows the reward earned by the service provider when executing the application DAG configurations. The results show that the reward earned with the NAS algorithm is higher than the NUS algorithm. The NAS algorithm schedules

[4]http://fi.ict.ac.cn/data/cloud.html
[5]https://www.kaggle.com/anuragk240/mobile-data-speeds-of-all-india-during-march-2018

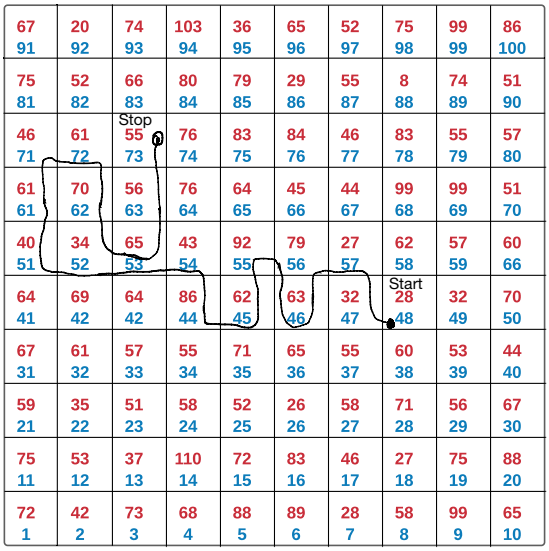

Figure 6.7 Network data and sample edge device mobility path in a Smart City

the application tasks by utilizing the network conditions, i.e., the task execution deferral in the low region allows the service provider to use their resources to cater for task transfers in the high network region, thereby increasing their reward.

We also evaluated the effect of changing network conditions on the NAS and NUS algorithms. This allows us to assess the performance of our switching algorithm. We introduce network changes by changing the network states at different time points in the simulation and observing the performance of both NAS and NUS algorithms. NAS algorithm speculatively changes schedules to the best schedule when it detects a network change while the NUS algorithm keeps running with the same schedule. We compared the performance of both algorithms using the DAG15 application configuration. The network changes were varied from 1 - 4, that is, we observed the algorithm behaviour for different network change injection points.

As seen in Fig. 6.11, the NAS algorithm provides a better makespan when compared to the NUS algorithm and it also gives higher rewards as shown in Fig. 6.12. Overall, we observe that when the network changes from a good state to a bad state, no improvement is noticed in the schedule deployed as that schedule is the

best schedule that can give the best performance given that network state condition. However, when the network changes from a bad state to a good state, then improvements to the performance of the application can be noticed in both algorithms. The switching algorithm deployed by NAS takes advantage of the better network states and provides better performance to the application in terms of reduced makespan and higher rewards.

In the experiments reported so far, the DAG configurations that were evaluated are computational-intensive, therefore the benefit of NAS over NUS is marginal in terms of makespan. In Figure 6.13, we evaluated the different CCR values for the DAG15 configuration for both algorithms. The results show that the makespan for the computation-intensive DAG is lower than the communication-intensive DAG, however, the NAS algorithm provides better makespan across the different values of CCR compared to the NUS algorithm.

Generally, for computation-intensive DAGs, the makespan is close for both NAS and NUS algorithms. With small CCR values, the communication is less important than the computation and network-awareness of the NAS algorithm does not lead to significant benefits over the NUS algorithm. The improvement of the NAS algorithm becomes significant for balanced (CCR = 1) and communication-intensive (CCR = 10) DAGs where the network-awareness can benefit from the increase in the number of data transfers.

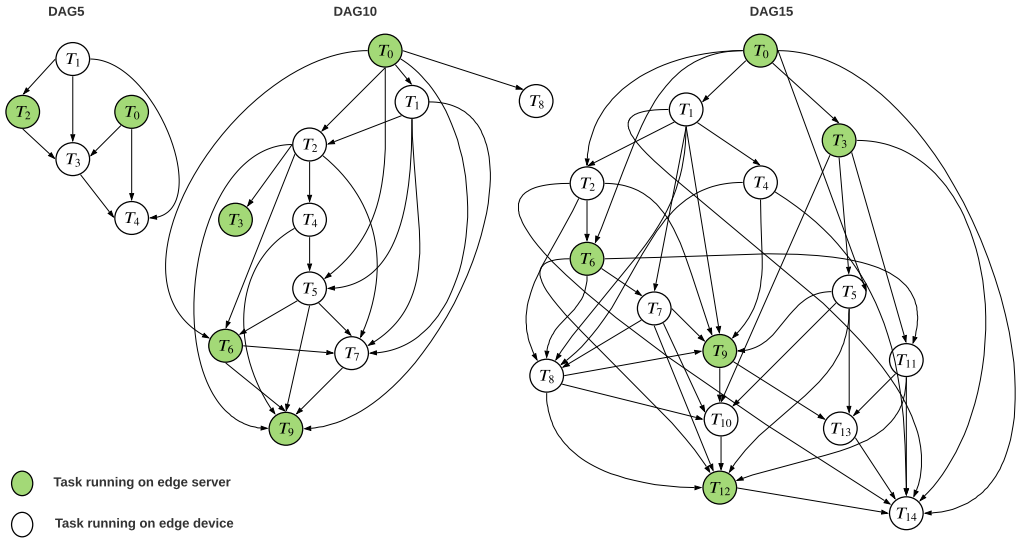

Figure 6.8 DAG test cases

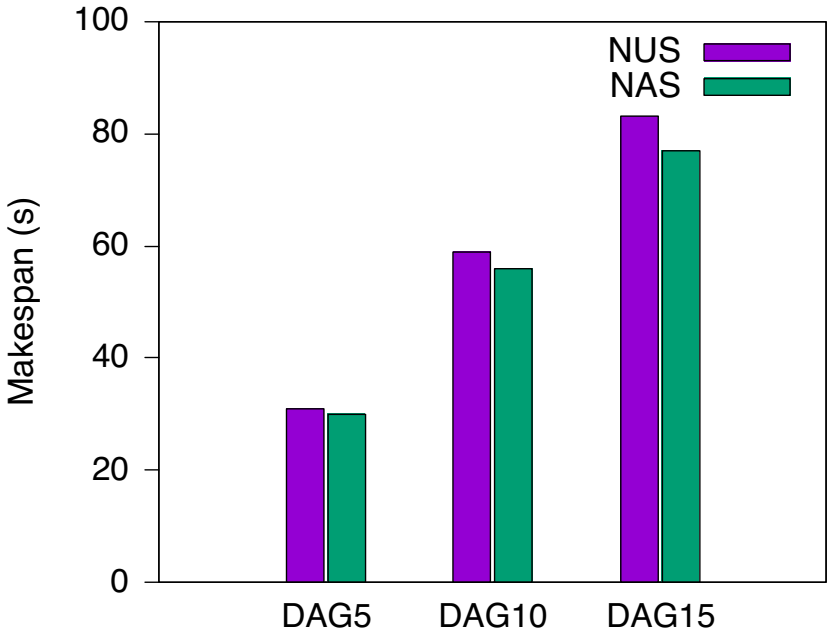

Figure 6.9 Makespan

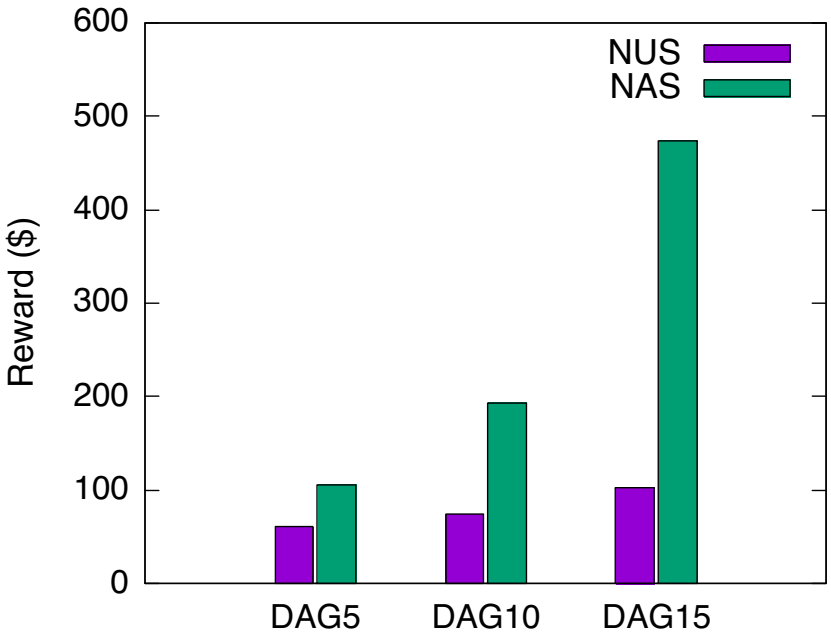

Figure 6.10 Reward earned by the Service Provider

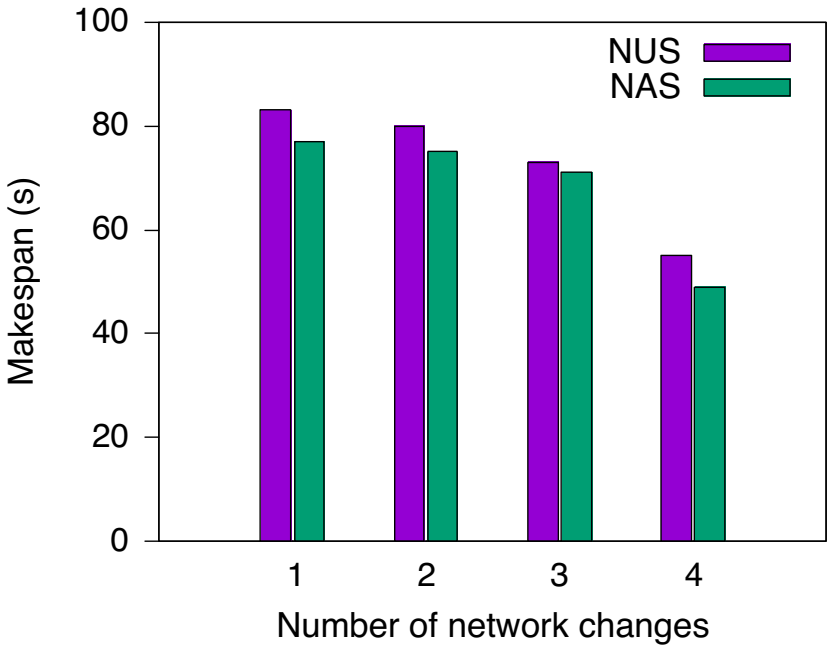

Figure 6.11 Makespan for different network changes

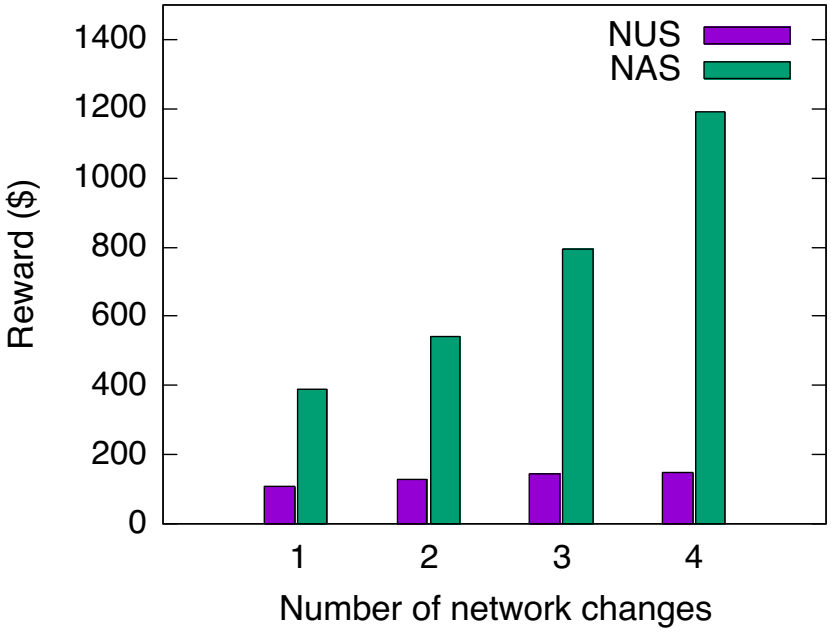

Figure 6.12 Reward for different network changes

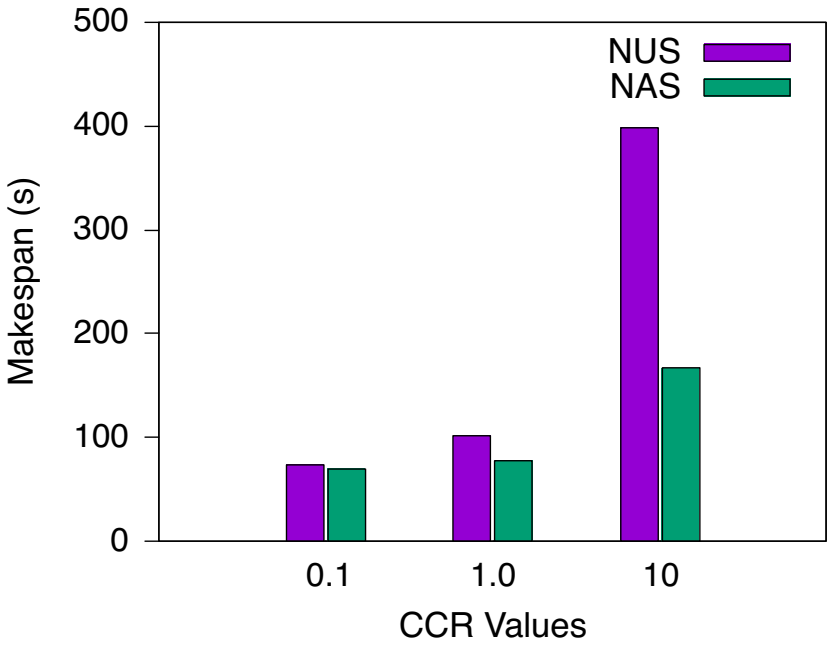

Figure 6.13 Makespan vs different CCR values

Chapter 7

Conclusion and Future Work

In this thesis, we explore task allocation and scheduling algorithms for different application types at the network edge. Distributing cloud resources to the edge enables applications to benefit from low latency access to server resources, and they can be programmed to use both edge and cloud resources. To make this work, we need task allocation and scheduling algorithms that can cater to the different applications running at the edge. We classify the applications into different types and propose algorithms that enhance the performance of the different application types.

Applications with independent batch tasks need to be sent to the most appropriate edge servers such that application-level metrics such as response times can be minimized without wasting resources. In Chapter 4, we propose a new resource and task allocation algorithm that achieves the stated objective. We compare our algorithm with a previously published algorithm and show that the proposed algorithm outperforms the previous algorithm under many application deployment scenarios. We also examine the performance of the task allocation algorithms under overload and failure scenarios and show that the proposed algorithm is capable of handling adverse conditions.

In Chapter 5, we focused on Edge Intelligence (EI) applications with a mix of real-time and non-time tasks. We highlight the challenges involved with schedul-

ing EI applications by identifying three problems. To tackle these problems, we propose a two-stage scheduling framework. In this framework, applications have three classes of tasks: real-time, interactive and batch. The first stage matches the tasks to the edge servers by taking their execution requirements (e.g., where the tasks want to run and what type of edge server they want to use). The first stage provides two possible schedules (termed Normal and Greedy) to the second stage. Additionally, the first stage places slack servers that can be used by the second stage to execute non real-time tasks. The second stage is responsible for adopting the best schedule between the two choices it gets from the first stage. The experimental results obtained from the prototype implementation show that the flexibility offered to the second stage by the first stage allows it to meet the real-time task constraints while maximizing the batch and interactive task execution rates or deadline compliance rates. Our experimental results show that we successfully tackled the identified problems.

Finally, in Chapter 6, we focused on applications with dependent tasks. We tackled the problem of network link variation in scheduling applications. We highlight the problem network performance variations can have for edge computing and motivate the need for network-aware task schedulers that would adjust the execution schedule to minimize data transfers in low network zones. We present the full architecture for network-aware scheduling for edge and develop the core algorithms that go into the architecture. Our architecture observes the state of the network and predicts the future state of the network and speculatively revises the task execution schedules to maximize the performance. The idea is to defer task execution such that the data transfers would avoid low network zones.

We developed a network-aware scheduling algorithm as part of this work. This algorithm returns the best schedule that minimizes the execution time given the network states and the switching algorithm. Our algorithm effects a schedule switching when a network change is detected if it deems such a switch is beneficial with regard to overall performance. At runtime, the combination of both algorithms in our framework offers better performance to the application from both the end-user and service provider perspectives. Results obtained show that the

algorithms offer minimized execution time for the end-user and better rewards for the service provider.

7.1 Future Work

One area of future work is to extend the task matching algorithms developed in Chapter 5 to be proximity aware (i.e., consider the edge server locations). By considering the location, we can directly control the latency in the matching step. Also, real-time tasks are matched to resources by a single attribute i.e., even though resources have more than one attribute and real-time requests for more than one matching, resources are only matched to a single column. We will extend this by ensuring real-time tasks can specify a flexible attribute-aware matching condition that dictates how the edge server to task matching is carried out. Furthermore, as part of the future work, we will examine how the results of the L0-Stage can be used by the L1-Stage to develop efficient matching schemes based on real-time task throughput values.

In Chapter 6, while we consider the application scheduling from a single edge device running an application on the network, in the future, we hope to consider multiple edge devices running applications and scheduling them as a group. We can schedule multiple application tasks such that the overall performance of all edge devices is optimized. Furthermore, we can extend the solution to consider the impact of queuing and heterogeneous delays to deliver a more complete solution.

In addition, we considered an application model where a batch mode is used for data transfers, i.e., tasks finish their execution before data transfer can begin. As future work, we will consider streaming data models where data can continuously be transferred while the task execution is ongoing. Also, as part of future work, we will consider time-constrained tasks, that is, applications with real-time deadlines. Scheduling such real-time tasks with network constraints is a challenging but interesting research problem.

Finally, we hope to fully implement our solutions into an open source framework

and programming language for edge computing applications [169, 170] and also carry out the evaluation using a live test bed under real-life scenarios.

Bibliography

[1] R. Mahmud, R. Kotagiri, and R. Buyya, *Fog Computing: A Taxonomy, Survey and Future Directions*, pp. 103–130. Singapore: Springer Singapore, 2018.

[2] M. Aazam and E.-N. Huh, "Fog computing micro datacenter based dynamic resource estimation and pricing model for IoT," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pp. 687–694, IEEE, 2015.

[3] Ericsson, "Iot connections outlook," 2021. `https://www.ericsson.com/en/mobility-report/dataforecasts/iot-connections-outlook`.

[4] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. A. Lee, and J. Kubiatowicz, "The Cloud is Not Enough: Saving IoT from the Cloud.," in *HotStorage*, 2015.

[5] P. Corcoran and S. K. Datta, "Mobile-edge computing and the internet of things for consumers: Extending cloud computing and services to the edge of the network," *IEEE Consumer Electronics Magazine*, vol. 5, no. 4, pp. 73–74, 2016.

[6] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource provisioning for iot services in the fog," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 32–39, Nov 2016.

[7] H. El-Sayed, S. Sankar, M. Prasad, D. Puthal, A. Gupta, M. Mohanty, and C.-T. Lin, "Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment," *IEEE Access*, vol. 6, pp. 1706–1717, 2017.

[8] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the internet of things," *IEEE access*, vol. 6, pp. 6900–6919, 2017.

[9] N. Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran, "The role of edge computing in internet of things," *IEEE Communications Magazine*, vol. 56, no. 11, pp. 110–115, 2018.

[10] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[11] E. Peltonen, M. Bennis, M. Capobianco, M. Debbah, A. Ding, F. Gil-Castiñeira, M. Jurmu, T. Karvonen, M. Kelanti, A. Kliks, *et al.*, "6g white paper on edge intelligence," *arXiv preprint arXiv:2004.14850*, 2020.

[12] A. H. Lodhi, B. Akgün, and Ö. Özkasap, "State-of-the-art techniques in deep edge intelligence," *arXiv preprint arXiv:2008.00824*, 2020.

[13] A. Keivani and J.-R. Tapamo, "Task scheduling in cloud computing: A review," in *2019 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*, pp. 1–6, 2019.

[14] Y. Ai, M. Peng, and K. Zhang, "Edge computing technologies for internet of things: a primer," *Digital Communications and Networks*, vol. 4, no. 2, pp. 77–86, 2018.

[15] J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, 2018.

[16] Y. Shao, C. Li, Z. Fu, L. Jia, and Y. Luo, "Cost-effective replication management and scheduling in edge computing," *Journal of Network and Computer Applications*, vol. 129, pp. 46–61, 2019.

[17] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 20–26, 2016.

[18] Amazon, "Amazon Web Services," 2021. `https://aws.amazon.com`.

[19] Microsoft, "Microsft Azure," 2021. `https://azure.microsoft.com/en-ca/`.

[20] Google, "Google App Engine," 2021. `https://cloud.google.com/appengine`.

[21] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, "Toward edge intelligence: Multiaccess edge computing for 5g and internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6722–6747, 2020.

[22] M. Bhayani, M. Patel, and C. Bhatt, "Internet of things (iot): In a way of smart world," in *Proceedings of the international congress on information and communication technology*, pp. 343–350, Springer, 2016.

[23] S. Raileanu, T. Borangiu, O. Morariu, and I. Iacob, "Edge computing in industrial iot framework for cloud-based manufacturing control," in *2018 22nd International Conference on System Theory, Control and Computing (ICSTCC)*, pp. 261–266, 2018.

[24] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the internet of things: A case study," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1275–1284, 2018.

[25] B. Omoniwa, R. Hussain, M. A. Javed, S. H. Bouk, and S. A. Malik, "Fog/edge computing-based iot (feciot): Architecture, applications, and research issues," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4118–4149, 2019.

[26] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the internet of things: A case study," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1275–1284, 2018.

[27] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[28] T. Oo and Y.-B. Ko, "Application-aware task scheduling in heterogeneous edge cloud," in *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1316–1320, 2019.

[29] M. R. Alizadeh, V. Khajehvand, A. M. Rahmani, and E. Akbari, "Task scheduling approaches in fog computing: A systematic review," *International Journal of Communication Systems*, vol. 33, no. 16, p. e4583, 2020.

[30] 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.

[31] Y. Li, Y. Chen, T. Lan, and G. Venkataramani, "Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1261–1270, 2017.

[32] D. Xu, A. Zhou, X. Zhang, G. Wang, X. Liu, C. An, Y. Shi, L. Liu, and H. Ma, "Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, (New York, NY, USA), p. 479–494, Association for Computing Machinery, 2020.

[33] J. Patman, P. Lovett, A. Banning, A. Barnert, D. Chemodanov, and P. Calvam, "Data-driven edge computing resource scheduling for protest crowds incident management," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pp. 1–8, 2018.

[34] A. Samanta and J. Tang, "Dyme: Dynamic microservice scheduling in edge computing enabled iot," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6164–6174, 2020.

[35] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh, "White space networking with wi-fi like connectivity," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 27–38, 2009.

[36] O. Fadahunsi and M. Maheswaran, "Locality sensitive request distribution for fog and cloud servers," *Service Oriented Computing and Applications*, pp. 1–14, 2019.

[37] O. Fadahunsi, Y. Ma, and M. Maheswaran, "Edge scheduling framework for real-time and non real-time tasks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 719–728, 2021.

[38] C. Doukas and I. Maglogiannis, "Bringing iot and cloud computing towards pervasive healthcare," in *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 922–926, IEEE, 2012.

[39] A. R. Biswas and R. Giaffreda, "Iot and cloud convergence: Opportunities and challenges," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pp. 375–376, IEEE, 2014.

[40] M. Aazam, E.-N. Huh, M. St-Hilaire, C.-H. Lung, and I. Lambadaris, "Cloud of things: integration of iot with cloud computing," in *Robots and sensor clouds*, pp. 77–94, Springer, 2016.

[41] S. Svorobej, P. Takako Endo, M. Bendechache, C. Filelis-Papadopoulos, K. M. Giannoutakis, G. A. Gravvanis, D. Tzovaras, J. Byrne, and T. Lynn,

"Simulating fog and edge computing scenarios: An overview and research challenges," *Future Internet*, vol. 11, no. 3, p. 55, 2019.

[42] M. Iorga, L. Feldman, R. Barton, M. Martin, N. Goren, and C. Mahmoudi, "The nist definition of fog computing," tech. rep., National Institute of Standards and Technology, 2017.

[43] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky, "Key ingredients in an iot recipe: Fog computing, cloud computing, and more fog computing," in *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pp. 325–329, IEEE, 2014.

[44] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[45] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[46] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 20–26, IEEE, 2016.

[47] X. Deng, J. Li, E. Liu, and H. Zhang, "Task allocation algorithm and optimization model on edge collaboration," *Journal of Systems Architecture*, vol. 110, p. 101778, 2020.

[48] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan, "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers," *Computer Networks*, vol. 130, pp. 94–120, 2018.

[49] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, "Toward edge intelligence: Multiaccess edge computing for 5g and internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6722–6747, 2020.

[50] H. Guo, J. Liu, and J. Zhang, "Efficient computation offloading for multi-access edge computing in 5g hetnets," in *2018 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2018.

[51] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, "A survey on end-edge-cloud orchestrated network computing paradigms: transparent computing, mobile edge computing, fog computing, and cloudlet," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–36, 2019.

[52] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, (New York, NY, USA), pp. 13–16, ACM, 2012.

[53] A. V. Dastjerdi and R. Buyya, "Fog computing: Helping the internet of things realize its potential," *Computer*, vol. 49, no. 8, pp. 112–116, 2016.

[54] 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.

[55] 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.

[56] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, (New York, NY, USA), pp. 13–16, ACM, 2012.

[57] L. M. Vaquero and L. Rodero-Merino, "Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 27–32, Oct. 2014.

[58] M. Beltrán, "Automatic provisioning of multi-tier applications in cloud computing environments," *The Journal of Supercomputing*, vol. 71, no. 6, pp. 2221–2250, 2015.

[59] H. Liu and D. Orban, "Gridbatch: Cloud computing for large-scale data-intensive batch applications," in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 295–305, IEEE, 2008.

[60] M. Barcelo, A. Correa, J. Llorca, A. M. Tulino, J. L. Vicario, and A. Morell, "Iot-cloud service optimization in next generation smart environments," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 4077–4090, 2016.

[61] G. Peralta, M. Iglesias-Urkia, M. Barcelo, R. Gomez, A. Moran, and J. Bilbao, "Fog computing based efficient iot scheme for the industry 4.0," in *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, pp. 1–6, IEEE, 2017.

[62] M. Satyanarayanan, G. Lewis, E. Morris, S. Simanta, J. Boleng, and K. Ha, "The role of cloudlets in hostile environments," *IEEE Pervasive Computing*, vol. 12, no. 4, pp. 40–49, 2013.

[63] 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.

[64] F. Andriopoulou, T. Dagiuklas, and T. Orphanoudakis, "Integrating iot and fog computing for healthcare service delivery," in *Components and services for IoT platforms*, pp. 213–232, Springer, 2017.

[65] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and Applications," in *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pp. 73–78, IEEE, 2015.

[66] Y. Cao, S. Chen, P. Hou, and D. Brown, "FAST: A fog computing assisted distributed analytics system to monitor fall for stroke mitigation," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pp. 2–11, IEEE, 2015.

[67] T. Malche and P. Maheshwary, "Internet of things (iot) for building smart home system," in *2017 International conference on I-SMAC (IoT in social, mobile, analytics and cloud)(I-SMAC)*, pp. 65–70, IEEE, 2017.

[68] C. Wei, Z. M. Fadlullah, N. Kato, and I. Stojmenovic, "On optimally reducing power loss in micro-grids with power storage devices," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 7, pp. 1361–1370, 2014.

[69] T. H. Luan, L. X. Cai, J. Chen, X. Shen, and F. Bai, "VTube: Towards the media rich city life with autonomous vehicular content distribution," in *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2011 8th Annual IEEE Communications Society Conference on*, pp. 359–367, IEEE, 2011.

[70] M. Eltoweissy, S. Olariu, and M. Younis, "Towards autonomous vehicular clouds," in *International Conference on Ad Hoc Networks*, pp. 1–16, Springer, 2010.

[71] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[72] A. M. Alakeel *et al.*, "A guide to dynamic load balancing in distributed computer systems," *International Journal of Computer Science and Information Security*, vol. 10, no. 6, pp. 153–160, 2010.

[73] Y.-Y. Shih, W.-H. Chung, A.-C. Pang, T.-C. Chiu, and H.-Y. Wei, "Enabling low-latency applications in fog-radio access networks," *IEEE network*, vol. 31, no. 1, pp. 52–58, 2016.

[74] 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.

[75] J. Hiller, M. Henze, M. Serror, E. Wagner, J. N. Richter, and K. Wehrle, "Secure low latency communication for constrained industrial iot scenarios," in *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, pp. 614–622, IEEE, 2018.

[76] M. Ghobaei-Arani, A. Souri, and A. A. Rahmanian, "Resource management approaches in fog computing: a comprehensive review," *Journal of Grid Computing*, pp. 1–42, 2019.

[77] R. O. Aburukba, M. AliKarrar, T. Landolsi, and K. El-Fakih, "Scheduling internet of things requests to minimize latency in hybrid fog–cloud computing," *Future Generation Computer Systems*, vol. 111, pp. 539–551, 2020.

[78] 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.

[79] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency iot services in multi-access edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 668–682, 2019.

[80] Z. Zhou, H. Liao, B. Gu, S. Mumtaz, and J. Rodriguez, "Resource sharing and task offloading in iot fog computing: A contract-learning approach," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 3, pp. 227–240, 2019.

[81] A. Elgazar, K. Harras, M. Aazam, and A. Mtibaa, "Towards intelligent edge storage management: Determining and predicting mobile file popularity,"

in *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pp. 23–28, IEEE, 2018.

[82] H. R. Arkian, A. Diyanat, and A. Pourkhalili, "Mist: Fog-based data analytics scheme with cost-efficient resource provisioning for iot crowdsensing applications," *Journal of Network and Computer Applications*, vol. 82, pp. 152–165, 2017.

[83] M. Etemadi, M. Ghobaei-Arani, and A. Shahidinejad, "Resource provisioning for iot services in the fog computing environment: An autonomic approach," *Computer Communications*, vol. 161, pp. 109–131, 2020.

[84] R. Kitchin, "The real-time city? big data and smart urbanism," *GeoJournal*, vol. 79, no. 1, pp. 1–14, 2014.

[85] J. Liu, J. Li, L. Zhang, F. Dai, Y. Zhang, X. Meng, and J. Shen, "Secure intelligent traffic light control using fog computing," *Future Generation Computer Systems*, vol. 78, pp. 817–824, 2018.

[86] A. Y. Nee and S.-K. Ong, "Virtual and augmented reality applications in manufacturing," *IFAC proceedings volumes*, vol. 46, no. 9, pp. 15–26, 2013.

[87] D. You, T. V. Doan, R. Torre, M. Mehrabi, A. Kropp, V. Nguyen, H. Salah, G. T. Nguyen, and F. H. Fitzek, "Fog computing as an enabler for immersive media: Service scenarios and research opportunities," *IEEE Access*, vol. 7, pp. 65797–65810, 2019.

[88] S. M. Salman, T. A. Sitompul, A. V. Papadopoulos, and T. Nolte, "Fog computing for augmented reality: Trends, challenges and opportunities," in *2020 IEEE International Conference on Fog Computing (ICFC)*, pp. 56–63, IEEE, 2020.

[89] Z. Tan, H. Qu, J. Zhao, S. Zhou, and W. Wang, "Uav-aided edge/fog computing in smart iot community for social augmented reality," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4872–4884, 2020.

[90] P. Siano, "Demand response and smart grids—a survey," *Renewable and sustainable energy reviews*, vol. 30, pp. 461–478, 2014.

[91] M. Hussain, M. Beg, *et al.*, "Fog computing for internet of things (iot)-aided smart grid architectures," *Big Data and cognitive computing*, vol. 3, no. 1, p. 8, 2019.

[92] A. Kumari, S. Tanwar, S. Tyagi, N. Kumar, M. S. Obaidat, and J. J. Rodrigues, "Fog computing for smart grid systems in the 5g environment: Challenges and solutions," *IEEE Wireless Communications*, vol. 26, no. 3, pp. 47–53, 2019.

[93] G. M. Gilbert, S. Naiman, H. Kimaro, and B. Bagile, "A critical review of edge and fog computing for smart grid applications," in *International Conference on Social Implications of Computers in Developing Countries*, pp. 763–775, Springer, 2019.

[94] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Towards fault tolerant fog computing for iot-based smart city applications," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0752–0757, IEEE, 2019.

[95] A. Kumari, S. Tanwar, S. Tyagi, and N. Kumar, "Fog computing for healthcare 4.0 environment: Opportunities and challenges," *Computers & Electrical Engineering*, vol. 72, pp. 1–13, 2018.

[96] L. Mai, N.-N. Dao, and M. Park, "Real-time task assignment approach leveraging reinforcement learning with evolution strategies for long-term latency minimization in fog computing," *Sensors*, vol. 18, no. 9, p. 2830, 2018.

[97] C. Puliafito, E. Mingozzi, and G. Anastasi, "Fog computing for the internet of mobile things: Issues and challenges," in *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 1–6, 2017.

[98] C. Lin, G. Han, X. Qi, M. Guizani, and L. Shu, "A distributed mobile fog computing scheme for mobile delay-sensitive applications in sdn-enabled vehicular networks," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 5, pp. 5481–5493, 2020.

[99] X. Zhang, M. Pipattanasomporn, T. Chen, and S. Rahman, "An iot-based thermal model learning framework for smart buildings," *IEEE Internet of Things Journal*, vol. 7, no. 1, pp. 518–527, 2019.

[100] R. Casado-Vara, Z. Vale, J. Prieto, and J. M. Corchado, "Fault-tolerant temperature control algorithm for iot networks in smart buildings," *Energies*, vol. 11, no. 12, p. 3430, 2018.

[101] R. Casado-Vara, A. Martin-del Rey, S. Affes, J. Prieto, and J. M. Corchado, "Iot network slicing on virtual layers of homogeneous data for improved algorithm operation in smart buildings," *Future Generation Computer Systems*, vol. 102, pp. 965–977, 2020.

[102] R. Carli, G. Cavone, S. Ben Othman, and M. Dotoli, "Iot based architecture for model predictive control of hvac systems in smart buildings," *Sensors*, vol. 20, no. 3, p. 781, 2020.

[103] D. Raj, S. S. Lekshmi, J. Guruprasad, M. Urmila, T. A. Lakshmi, A. Vinod, T. Swathi, *et al.*, "Enabling technologies to realise smart mall concept in 5g era," in *2018 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pp. 1–6, IEEE, 2018.

[104] A. Sutagundar, M. Ettinamani, and A. Attar, "Iot based smart shopping mall," in *2018 Second International Conference on Green Computing and Internet of Things (ICGCIoT)*, pp. 355–360, IEEE, 2018.

[105] J. Rezazadeh, K. Sandrasegaran, and X. Kong, "A location-based smart shopping system with iot technology," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pp. 748–753, IEEE, 2018.

[106] M. Wang, J. Wu, G. Li, J. Li, Q. Li, and S. Wang, "Toward mobility support for information-centric iov in smart city using fog computing," in *2017 IEEE International Conference on Smart Energy Grid Engineering (SEGE)*, pp. 357–361, IEEE, 2017.

[107] C. Chellaswamy, H. Famitha, T. Anusuya, and S. B. Amirthavarshini, "Iot based humps and pothole detection on roads and information sharing," in *2018 International Conference on Computation of Power, Energy, Information and Communication (ICCPEIC)*, pp. 084–090, 2018.

[108] U. Ozeer, X. Etchevers, L. Letondeur, F.-G. Ottogalli, G. Salaün, and J.-M. Vincent, "Resilience of stateful iot applications in a dynamic fog environment," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pp. 332–341, 2018.

[109] V. Cardellini, G. Mencagli, D. Talia, and M. Torquati, "New landscapes of the data stream processing in the era of fog computing," *Future Generation Computer Systems*, vol. 99, pp. 646–650, 2019.

[110] H. Madiha, L. Lei, A. A. Laghari, and S. Karim, "Quality of experience and quality of service of gaming services in fog computing," in *Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences*, pp. 225–228, 2020.

[111] H. Lu, Q. Liu, D. Tian, Y. Li, H. Kim, and S. Serikawa, "The cognitive internet of vehicles for autonomous driving," *IEEE Network*, vol. 33, no. 3, pp. 65–73, 2019.

[112] J. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[113] I.-H. Hou, T. Zhao, S. Wang, and K. Chan, "Asymptotically Optimal Algorithm for Online Reconfiguration of Edge-clouds," in *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '16, (New York, NY, USA), pp. 291–300, ACM, 2016.

[114] J. Oueis, E. C. Strinati, and S. Barbarossa, "The Fog Balancing: Load Distribution for Small Cell Cloud Computing," in *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pp. 1–6, May 2015.

[115] Y. Song, S. S. Yau, R. Yu, X. Zhang, and G. Xue, "An Approach to QoS-based Task Distribution in Edge Computing Networks for IoT Applications," in *2017 IEEE International Conference on Edge Computing (EDGE)*, pp. 32–39, June 2017.

[116] D. Hoang and T. D. Dang, "FBRC: Optimization of task Scheduling in Fog-Based Region and Cloud," in *2017 IEEE Trustcom/BigDataSE/ICESS*, pp. 1109–1114, Aug 2017.

[117] A. Yousefpour, G. Ishigaki, and J. P. Jue, "Fog Computing: Towards Minimizing Delay in the Internet of Things," in *2017 IEEE International Conference on Edge Computing (EDGE)*, pp. 17–24, June 2017.

[118] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized IoT Service Placement in the Fog," *Service Oriented Computing and Applications*, vol. 11, no. 4, pp. 427–443, 2017.

[119] D. Isovic and G. Fohler, "Handling mixed sets of tasks in combined offline and online scheduled real-time systems," *Real-time systems*, vol. 43, no. 3, pp. 296–325, 2009.

[120] H. Tang, P. Ramanathan, and K. Morrow, "Inserting placeholder slack to improve run-time scheduling of non-preemptible real-time tasks in heterogeneous systems," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pp. 168–173, 2014.

[121] H. Tang, P. Ramanathan, and K. Compton, "Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources," in *2011 International Conference on Parallel Processing*, pp. 753–762, 2011.

[122] W. Jeon, W.Kim, H. Lee, and C. Lee, "Online slack-stealing scheduling with modified laedf in real-time systems," *Electronics*, vol. 8, no. 11, p. 1286, 2019.

[123] H. Chu and K. Nahrstedt, "Cpu service classes for multimedia applications," in *Proceedings IEEE International Conference on Multimedia Computing and Systems*, vol. 1, pp. 296–301, IEEE, 1999.

[124] B. Lin and P. Dinda, "Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling," in *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pp. 8–8, IEEE, 2005.

[125] T. Yang, T. Liu, E. Berger, S. Kaplan, and J. Moss, "Redline: First class support for interactivity in commodity operating systems.," in *OSDI*, vol. 8, pp. 73–86, 2008.

[126] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE transactions on software engineering*, no. 5, pp. 564–577, 1987.

[127] C. Shen, O. Gonzalez, K. Ramamritham, and I. Mizunuma, "User level scheduling of communicating real-time tasks.," in *IEEE Real Time Technology and Applications Symposium*, pp. 164–175, 1999.

[128] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proceedings of the 2013 conference on Internet measurement conference*, pp. 191–204, 2013.

[129] E. Arslan, M. Shekhar, and T. Kosar, "Locality and network-aware reduce task scheduling for data-intensive applications," in *2014 5th International Workshop on Data-Intensive Computing in the Clouds*, pp. 17–24, IEEE, 2014.

[130] X. He and P. Shenoy, "Firebird: Network-aware task scheduling for spark using sdns," in *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–10, IEEE, 2016.

[131] U. Fiore, F. Palmieri, A. Castiglione, and A. De Santis, "A cluster-based data-centric model for network-aware task scheduling in distributed systems," *International Journal of Parallel Programming*, vol. 42, no. 5, pp. 755–775, 2014.

[132] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, pp. 351–359, 2019.

[133] A. Lebre, J. Pastor, A. Simonet, and F. Desprez, "Revising openstack to operate fog/edge computing infrastructures," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 138–148, April 2017.

[134] E. Moradi and M. Bidkhori, "Single facility location problem," in *Facility Location*, pp. 37–68, Springer, 2009.

[135] M. Karatas, N. Razi, and H. Tozan, "A comparison of p-median and maximal coverage location models with q–coverage requirement," *Procedia Engineering*, vol. 149, pp. 169–176, 2016.

[136] J. Postel, "Transmission Control Protocol." `https://tools.ietf.org/html/rfc793`, 1981. Accessed: 23-03-2017.

[137] P. L'Ecuyer, L. Meliani, and J. Vaucher, "SSJ: A framework for stochastic simulation in Java," in *Proceedings of the 2002 Winter Simulation Conference* (E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.), pp. 234–242, IEEE Press, 2002. Available at `http://simul.iro.umontreal.ca/ssj/indexe.html`.

[138] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 1094–1104, Oct 2001.

[139] The University of Adelaide, "The Internet Topology Zoo." `http://www.topology-zoo.org/files/Cogentco.gml`. Accessed: 09-06-2017.

[140] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[141] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7457–7469, 2020.

[142] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[143] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," in *2017 Global Internet of Things Summit (GIoTS)*, pp. 1–6, 2017.

[144] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[145] J. Zhang and K. B. Letaief, "Mobile edge intelligence and computing for the internet of vehicles," *Proceedings of the IEEE*, vol. 108, no. 2, pp. 246–261, 2019.

[146] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[147] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3709–3716, 2018.

[148] S. Zhang, W. Li, Y. Wu, P. Watson, and A. Zomaya, "Enabling edge intelligence for activity recognition in smart homes," in *2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pp. 228–236, IEEE, 2018.

[149] E. Gilman, S. Tamminen, R. Yasmin, E. Ristimella, E. Peltonen, M. Harju, L. Lovén, J. Riekki, and S. Pirttikangas, "Internet of things for smart spaces: A university campus case study," *Sensors*, vol. 20, no. 13, p. 3716, 2020.

[150] R. Ke, Y. Zhuang, Z. Pu, and Y. Wang, "A smart, efficient, and reliable parking surveillance system with edge artificial intelligence on iot devices," *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[151] T. Rausch and S. Dustdar, "Edge intelligence: The convergence of humans, things, and ai," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 86–96, IEEE, 2019.

[152] L. U. Khan, I. Yaqoob, M. Imran, Z. Han, and C. S. Hong, "6g wireless systems: A vision, architectural elements, and future directions," *IEEE Access*, vol. 8, pp. 147029–147044, 2020.

[153] R. Shafin, L. Liu, V. Chandrasekhar, H. Chen, J. Reed, and J. C. Zhang, "Artificial intelligence-enabled cellular networks: A critical path to beyond-5g and 6g," *IEEE Wireless Communications*, vol. 27, no. 2, pp. 212–217, 2020.

[154] R. Gupta, D. Reebadiya, and S. Tanwar, "6g-enabled edge intelligence for ultra-reliable low latency applications: Vision and mission," *Computer Standards & Interfaces*, vol. 77, p. 103521, 2021.

[155] S. El Zaatari, M. Marei, W. Li, and Z. Usman, "Cobot programming for collaborative industrial tasks: An overview," *Robotics and Autonomous Systems*, vol. 116, pp. 162–180, 2019.

[156] S. Saewong and R. Rajkumar, "Coexistence of real-time and interactive amp; batch tasks in dvs systems," in *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 24–33, 2008.

[157] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pp. 129–139, 1991.

[158] F. Cottet, J. Delacroix, Z. Mammeri, and C. Kaiser, *Scheduling in real-time systems*. Wiley Online Library, 2002.

[159] M. Nasri, S. Baruah, G. Fohler, and M. Kargahi, "On the optimality of rm and edf for non-preemptive real-time harmonic tasks," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, (New York, NY, USA), p. 331–340, Association for Computing Machinery, 2014.

[160] J. Liang, K. Li, C. Liu, and K. Li, "Joint offloading and scheduling decisions for dag applications in mobile edge computing," *Neurocomputing*, vol. 424, pp. 160–171, 2021.

[161] Y. Sahni, J. Cao, L. Yang, and Y. Ji, "Multi-hop offloading of multiple dag tasks in collaborative edge computing," *IEEE Internet of Things Journal*, pp. 1–1, 2020.

[162] D. Kliazovich, J. E. Pecero, A. Tchernykh, P. Bouvry, S. U. Khan, and A. Y. Zomaya, "Ca-dag: Modeling communication-aware applications for scheduling in cloud computing," *Journal of Grid Computing*, vol. 14, no. 1, pp. 23–39, 2016.

[163] M. W. Convolbo and J. Chou, "Cost-aware dag scheduling algorithms for minimizing execution cost on cloud resources," *The Journal of Supercomputing*, vol. 72, no. 3, pp. 985–1012, 2016.

[164] B. Hu and Z. Cao, "Minimizing resource consumption cost of dag applications with reliability requirement on heterogeneous processor systems," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 12, pp. 7437–7447, 2019.

[165] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, no. 4, pp. 11–20, 2020.

[166] A. D. Kalvin and Y. L. Varol, "On the generation of all topological sortings," *Journal of Algorithms*, vol. 4, no. 2, pp. 150–162, 1983.

[167] R. N. Pantos, E. Vershen, and W. B. May Jr, "Http live streaming dateranges," Dec. 3 2015. US Patent App. 14/503,165.

[168] L. Bracciale, M. Bonola, P. Loreti, G. Bianchi, R. Amici, and A. Rabuffi, "Crawdad dataset roma/taxi (v. 2014-07-17)," 2014. `https://crawdad.org/roma/taxi/20140717`.

[169] R. Wenger, X. Zhu, J. Krishnamurthy, and M. Maheswaran, "A programming language and system for heterogeneous cloud of things," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pp. 169–177, IEEE, 2016.

[170] M. Maheswaran, "JAMScript - A Programming Language for Edge-Oriented Mobile IoT," 2021. `https://citelab.github.io/JAMScript/`.