FogBus2: A Lightweight and Distributed Container-based Framework for Integration of IoT-enabled Systems with Edge and Cloud Computing

by

Qifan Deng

Supervised under Prof. Rajkumar Buyya

A thesis submitted for the
75-pts Research Project COMP90070
and degree of Master of Science (Computer Science)

in the
School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

August 2021

THE UNIVERSITY OF MELBOURNE

Abstract

School of Computing and Information Systems

Master of Science (Computer Science)

by Qifan Deng

Supervised under Prof. Rajkumar Buyya

Edge/Fog computing is a novel computing paradigm that provides resource-limited Internet of Things (IoT) devices with scalable computing and storage resources. Compared to cloud computing, edge/fog servers have fewer resources, but they can be accessed with higher bandwidth and less communication latency. Thus, integrating edge/fog and cloud infrastructures can support the execution of diverse latency-sensitive and computation-intensive IoT applications. Although some frameworks attempt to provide such integration, there are still several challenges to be addressed, such as dynamic scheduling of different IoT applications, scalability mechanisms, multi-platform support, and supporting different interaction models. To overcome these challenges, we propose a lightweight and distributed container-based framework, called FogBus2. It provides a mechanism for scheduling heterogeneous IoT applications and implements several scheduling policies. Also, it proposes an optimized genetic algorithm to obtain fast convergence to well-suited solutions. Besides, it offers a scalability mechanism to ensure efficient responsiveness when either the number of IoT devices increases or the resources become overburdened. Also, the dynamic resource discovery mechanism of FogBus2 assists new entities to quickly join the system. We have also developed two IoT applications, called Conway's Game of Life and Video Optical Character Recognition to demonstrate the effectiveness of FogBus2 for handling real-time and non-real-time IoT applications. Experimental results show FogBus2's scheduling policy improves the response time of IoT applications by 53% compared to other policies. Also, the scalability mechanism can reduce up to 48% of the queuing waiting time compared to frameworks that do not support scalability.

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Internet of Things (IoT) devices has become an inseparable part of our daily lives, where IoT applications provide diverse solutions for intelligent healthcare, transportation, and entertainment, to mention a few [1]. IoT applications often produce a massive amount of data for processing and storage. IoT devices connect to different networks with numerous sensors interacting with or sensing the internal and external environments on every second [2]. However, the computing and storage resources of IoT devices are limited. Therefore, IoT devices are usually integrated with resourceful surrogate resource providers to obtain better services for their users. Cloud computing, as a centralized computing paradigm, is one of the main enablers of IoT that offers unlimited computing and storage resources [3, 4]. IoT devices can place whole or some parts of their applications to cloud servers for processing and storage. These applications benefit from cloud computing as the computing is delivered in the form of services provided by the cloud, which are stable and with high performance, [5–7].

However, the networking between IoT devices and cloud clusters or data centers (which are usually aggregated) involves transportation of massive data over the Internet. This physical structure brings high latency when IoT applications use cloud computing as a part of the primary computing module in the design. The emergence of real-time IoT applications indicates that cloud computing cannot solely provide efficient services for latency-sensitive IoT applications due to its high access latency and low bandwidth [8, 9]. Moreover, although cloud services providers provide a 'pay as you go' plan [10–12] that charges only on-demand computing, storage, and networking resources, it is expensive when the amount of data needed to be processed, transported and stored is enormous [13]. To address these issues, edge/fog computing, which is a novel distributed computing paradigm, is proposed, providing distributed computing and storage resources in the proximity of IoT devices with higher access bandwidth and lower communication latency
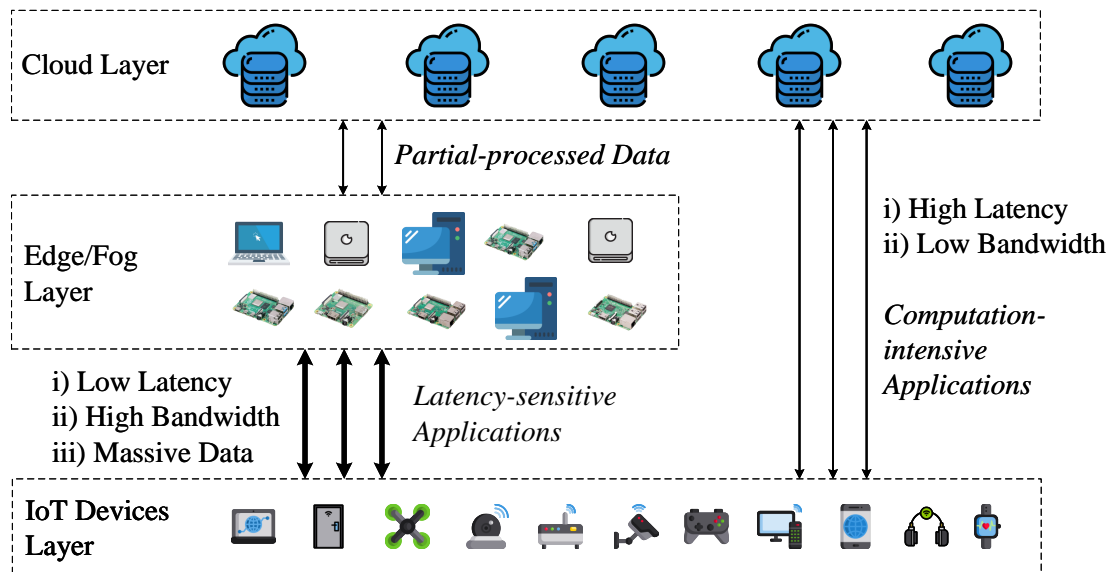
FIGURE 1.1: Environments and applications of IoT, Edg/fog and Cloud layers

[14]. Edge computing exists closely to IoT devices where data is generated. It computes, stores, and forms networks to serve IoT devices with low latency, high bandwidth, and distributed mobility [15–17]. Edge servers process and analyzes data around where it is generated, the overload of transmission to a centralized data center is dramatically reduced. Combining with techniques like 5G networks, edge computing further enables faster data processing which boosts the evaluation of applications like autonomous electric transport vehicles, autonomous drones, healthcare devices, and autonomous robots in factories [18, 19]. The environment and applications of IoT, Edg/fog and Cloud layers are presented in Figure 1.1.

1.1   Challenges and Motivation

Compared to cloud servers' resources, edge/fog servers have limited computing and storage resources, and hence they cannot efficiently execute computation-intensive tasks of IoT devices. To address this issue, edge/fog servers can collaboratively use their resources or cloud servers'. However, there is overload when simultaneously using distributed resources. Not only the algorithms and communication protocols need to be carefully designed, but networking bandwidth and latency also need to be considered. Thus, seamless integration of edge/fog and cloud infrastructures to support different IoT applications is an important research topic. Resources of distributed edge/fog servers and cloud servers are highly heterogeneous in terms of computing capabilities, processors' architectures, RAM capacity, and supported communication protocols [20]. Also,

IoT applications are heterogeneous in terms of applications' granularity (i.e., task, service), dependency model of constituent parts of IoT applications (i.e., independent tasks, sequential dependency, and complex dependent tasks), and their quality of service requirements (such as computation-intensive or latency-sensitive applications).

According to these factors, there are several framework design challenges to be considered. First, frameworks working in the integrated platform should support platform-independent techniques to overcome communication and run-time obstacles. Second, due to the heterogeneity of resources and the requirements of IoT applications, distributed scheduling mechanisms are required to place/offload tasks/data of IoT applications on suitable servers for processing and storage. Third, fast application deployments and scalability support are required in this integrated environment to provide services for IoT devices in a timely manner. Fourth, to efficiently reuse the resources, the containerization concepts can be adopted for the software components of the framework and IoT applications.

1.2   Research Problems

To address the challenges we discussed, the following questions are identified and investigated in this thesis. By answering these questions with solutions in the framework design, the challenges in the previous chapter should be completed.

- **How to use edge/fog servers and cloud servers simultaneously in an efficient way?** When cloud servers are centralized, and with high performance, the access latency is always high. Thus, cloud servers cannot solely provide efficient services for latency-sensitive applications. Edge/fog servers are with lower performance but very high bandwidth and low latency to IoT devices complete cloud servers and offset the drawbacks of cloud servers. An efficient framework has to use both the resources of edge/fog servers and cloud servers collaboratively.

- **How to provide a scheduling mechanism for incoming requests from different types of IoT applications (latency-sensitive and computation-intensive)?** When the performance and resources diverse in the heterogeneous edge/fog servers and cloud servers, numerous applications may be latency-sensitive and or computation-intensive. A platform needs to intelligently decide how to arrange the executions for all the kind of applications. Typically, there is a time limitation for the scheduling algorithm to perform, which requires the algorithm to finish in a short time but finishes with a reasonable solution of the arrangement of the execution.

- **How to provide a scalable platform in these heterogeneous computing environments?** With the scheduling mechanism, a framework allows real-time scheduling for the heterogeneous application execution request. However, the demands need to be rapidly responded even there are a huge amount of requests coming to the system at the same time. To respond to the requests efficiently, scalability is required for frameworks that automatically scale, create, or allocate resources responding to the execution requests of IoT applications. This mechanism helps reduce user-side waiting time for resource placement, thus bring users a better experience.

- **How to support automatically discover resources and reuse for such platform?** The amount of IoT devices and edge/fog servers is countless and keep growing all the time. Under this background, a framework should have the ability to automatically identify resources when IoT devices and edge/fog servers join or leave the system unpredictably. Nevertheless, after discovering resources, an effective mechanism should be designed to operate the resources reasonably, including the reuse of the resources. Frameworks should recognize this problem during the design phase to better advantage all the heterogeneity IoT, edge/fog servers, and cloud servers.

1.3   Thesis Contributions

Although there are some frameworks to manage integrated resources in edge/fog computing [21, 22], they barely consider platform-independent techniques, scheduling of heterogeneous IoT applications with complex dependent structures, scalability mechanisms of distributed resource managers, and containerization (see Chapter 2).

To address these limitations and solve these research problems, we propose and develop a lightweight and distributed container-based framework called FogBus2, which is partially published in [23]. Our framework supports (1) different inter and intra interaction models among edge/fog servers and cloud servers to support the requirements of different IoT application scenarios when using IP and ports addresses, (2) containerization of software components of the framework for fast deployments when supporting different programming languages by following the communication protocol design, (3) containerization of constituent parts of IoT applications as dependent tasks or independent tasks, (4) scheduling of multiple IoT applications and scalability mechanisms based on monitoring the whole system by integrating logs from every component (5) concurrent execution of different types of IoT applications including computation-intensive

and latency-sensitive as well as complex dependent or simply related modules, and (6) efficient reuse of resources.

The main contributions of this paper are summarized as follows:

- A lightweight and distributed container-based framework, called FogBus2, is proposed to integrate edge/fog and cloud infrastructures to support the execution of heterogeneous IoT applications.

- Containerization-support for software components of the framework and IoT applications is proposed for fast deployment and efficient reuse of resources.

- Dynamic scheduling, scalability, and resource discovery mechanisms are developed for fast adaptation as the characteristics of environment change.

- A real-world prototype is developed using FogBus2 with a real-time IoT application named Conway's Game of Life, and a non-real-time IoT application, called Video Optical Character Recognition (VOCR).

1.4   Evaluation Methodologies

To conduct the experiments and evaluate the performance of our proposed framework, we had two potential approaches, namely simulation and real-world system setup. We first run all the virtual environment components, allocating different resources to virtual machines, simulating the heterogeneity of the integration edge/fog and cloud infrastructures. However, there was a considerable difference from the result of real-world infrastructures. Thus, after tried simulation and got no ideal results, we shifted our experimental setup to real-world devices, including Raspberry Pis, desktop, and cloud instances provided by different providers worldwide. The specifications of the infrastructures for experiments will be introduced in the performance evaluation sections, respectively.

The features of FogBus2 are evaluated using one real-world application we developed, named Conway's Game of Life, which is both computation-intensive and latency-sensitive. The application is implemented with 62 modules (tasks). For each data frame, every two modules execute for the computation of grids at the same size. The execution of modules with different sizes depending on their parent module's execution result. This structure strongly increases the complexity of the problem to schedule the execution of the application.

Theoretically, when there are N hosts distributed in the system, N^{62} different potential solutions exist to execute Conway's Game of Life. Because the resource of each host and bandwidth/data rate between any two pair of hosts are not the same, it is impossible to find a good enough solution in a short time with a native approach such as brute force. Thus, we implemented two popular policies and compared them with our proposed policy to evaluate our framework. We use response time, a lower better metric, to assess the performance of scheduling. The details of this experiment will be presented in Chapter 4.3.

However, the scheduling mechanism is not the only requirement of the framework to use edge/fog servers and cloud servers simultaneously efficiently. When there are numerous requests for the execution in the distributed system, the requests can not be put in a waiting queue too long, which brings users horrible experiences and wastes edge/fog servers and cloud servers by leaving them idle. Thus, the scalability mechanism is evaluated following the scheduling mechanism. Increasing numbers of simultaneous requests are tested in the experiment to determine how efficiently the scalability mechanism of FogBus2 is. We define and use Scheduling Finish Time (SFT). This metric is also lower better. The details will be presented in Chapter 4.4.

Moreover, as we use the containerization technique in the implementation, we experiment to evaluate how much Resource Ready Time (RRT) can be reduced when the containers for execution can be reused. In the experiment, we tested five different applications. Before requesting each of them, there are already execution containers in the cool-off period and ready to be reused. Thus, to verify the performance of the resources reuse mechanism, an experiment has been conducted to compare how long it would take for the resources placed to be ready when heterogeneous IoT applications are requested. With the defined metric, resource ready time, reuse mechanism is applied and bypassed for two IoT applications. The applications have different complexity regarding the number of modules (containers) required. The performance of our reuse mechanism will be examined in the experiment. The details will be shown in Chapter 4.5

Finally, startup time and RAM usages are studied in Chapter 4.6 comparing with FogBus [21] which shows the FogBus2 framework is lightweight.

Chapter 1
Introduction

Background, Motivation, Research Problems, Thesis Contribution, Evaluation Methodologies, Thesis Organization

Chapter 2
Literature Review

Inregration, Multi-platform Support, Multi-application Support, Scheduling, Scaling, Resource Discovery, Containerization and Summary.

Chapter 3
FogBus2 Framework for IoT-enabled Systems and Application

Hardware Components, Software Components and Interaction Diagram.

Chapter 4
Performance Evaluation

Sample Applications, Scheduling Analysis, Scalability Analysis, Resources Reuse Analysis, Startup Time and RAM Usage Analysis.

Chapter 5
Conclusions and Future Directions

Conclusions, Future Directions and Thesis Summary.

FIGURE 1.2: Visualized Thesis Organization

1.5   Thesis Organization

The rest of the paper is organized as follows. Relevant frameworks are reviewed, and features of them are summarized in topologies in Chapter 2. Chapter 3 presents the hardware and software components of the FogBus2 and some of the details of framework design. The performance of FogBus2 is evaluated in Chapter 4. Finally, Chapter 5 concludes the paper and draws future works. A visualized thesis organization is presented in Figure 1.2.

Chapter 2

Literature Review

This chapter discusses related frameworks integrating IoT-enabled systems with edge/-fog and cloud infrastructures. We have discussed research questions that include 1) how to use edge/fog devices and cloud servers simultaneously efficiently, 2) how to provide a scheduling mechanism for incoming requests from different types of IoT applications, 3) how to provide a scalable platform in these heterogeneous computing environments, and 4) how to support automatically discover resources and reuse for such platform. To answer these questions, five aspects are identified as follows based on the importance to consider the heterogeneity of infrastructures, heterogeneity of IoT applications, efficient use of heterogeneous resources, intelligently discovering resources, and making the strength of containerization technologies:

- **Integration** IoT applications produce a massive amount of data for processing and storage when edge computing exists extremely close to IoT devices. Edge/fog devices compute, store and form networks to serve IoT devices with low latency, high bandwidth, and distributed mobility. Cloud computing that provides unlimited computing and storage resources is necessary for computation-intensive applications with high performance even though cloud computing has low bandwidth and high latency. Seamless integration of edge/fog and cloud infrastructures to support different IoT applications is an important research topic. It is a challenge that a framework makes the most of strength by using the three efficiently in the framework design. Chapter 2.1 reviews frameworks that integrate IoT, edge/fog, and or the cloud.

- **Multi-platform Support** There are vast amounts of IoT devices, and they are gracefully embracing the heterogeneity of multiple platforms. The emergence of IoT devices brings a growing capacity of programming for distinct and different

devices when they use various CPU architectures, and a wide variety of operating systems [24]. A framework designed for running on IoT devices, ECs, and cloud servers with high diversity needs to support multiple platforms to integrate edge/fog and cloud infrastructures to support the execution of heterogeneous IoT applications and helps to reduce the burden of developers to support a variety of platforms. Chapter 2.2 introduces related frameworks which support multi-platform.

- **Multi-application Support** IoT devices diverse when they have various hardware and sensors when the data they produce and traffic demand also require different resources to transport and compute [25]. Consequently, IoT applications developed on top of various sensors and hardware are varied in a large number. A framework needs to support multiple IoT applications, which may be latency-sensitive, computation-intensive, and or bandwidth-sensitive. When different applications usually run simultaneously and distribute over the network, they can run separately or co-operate with each other. Thus, a framework needs to have the ability to support such different kinds of applications without requiring much effort that developers need to make. Chapter 2.3 reviews related frameworks which support multi-application.

- **Scheduling Mechanism** Resources of distributed edge/fog devices and cloud servers are highly heterogeneous in computing capabilities, processors' architectures, RAM capacity, and supported communication protocols. The scheduling task is usually a complex undertaking to manage such countless and growing resources [26]. Thus, an efficient scheduling mechanism is required to arrange heterogeneous IoT applications' execution using distributed edge/fog devices and cloud server resources. A scheduling algorithm (policy) real-timely configures the executions of different modules of an application, usually decides how to arrange and control the executions based on the current environment of the system, such as how many resources are available. The framework needs to obtain the ability to integrate multiple scheduling policies. Chapter 2.4 surveys related frameworks that use scheduling mechanisms.

- **Scaling Mechanism** The scaling mechanism is eminent to automatically keep Quality of Services (QoS) at a reasonable level which gives users a seamless experience when the workloads come into a system are unpredictable [27]. The scaling mechanism in a framework is responsible for automatically adding or allocating resources when the workload feeding into a system is growing. IoT applications are close to the natural environment and humans' living environment. Therefore,

the demand for the execution of the applications changes dynamically and unpredictably. A framework should have scaling mechanism support to dynamically control the distributed resources with such changing IoT applications execution demand. Chapter 2.5 studies a framework on its scaling mechanism.

- **Resource Discovery Mechanism** Resource Discovery, sometimes called Service Discovery, is the capability to automatically find and locate resources and or services that are not predefined or addressed in a network [28]. IoT devices, edge/fog devices, and cloud servers are fully distributed geographically and logically; thus, the resource discovery mechanism empowers the hosts to discover others, form a subnet, and share their resources. A framework needs to obtain the capability of resource discovery mechanism trying to use distributed resources as efficiently as possible. Chapter 2.6 investigates a framework on its resource discovery mechanism.

- **Containerization Technique** Containerization is the technique that provides operating system-level isolation of a process, a program, or a complete operating system environment. It is as secure as a virtual machine but more lightweight since it does not depend on the host hardware emulation [29]. The containerization technique allows both the framework components and IoT applications to be containerized, which benefits faster deployments of IoT applications, easy configuration of the framework itself, and automatic scaling mechanism. A framework supporting the containerization technique can be quickly released when retaining and taking advantage of the features of the containerization technique. Chapter 2.7 statisticizes related frameworks which adopts containerization technique.

2.1   Integration

Tuli et al. proposed the FogBus framework based on a master-worker approach [21]. This framework integrates IoT systems to the cloud and edge infrastructures when harnessing IoT resources, edge resources, and cloud resources. FogPlan, developed by Yousefpour et al., is a container-enabled framework integrating IoT devices with edge/fog devices and cloud devices to minimize the response time of IoT applications [30]. Merlino et al. developed a framework recognizing offloading patterns [20]. They use a middleware platform to integrates IoT, edge/fog devices, and cloud devices, which tries to improve OpenStack and Stack4Things. Nguyen et al. [31] proposed a privacy-preserving framework that handles requests and data in edge/fog devices and cloud devices, combining local execution at the edge and remote processing at the cloud. An et al. developed the EiF framework to bring artificial intelligence services to the edge of the network [22].

It manages service dependencies and relations of IoT applications over the network of edge devices and cloud devices. Borthakur et al. developed the SmartFog framework, integrating IoT devices with edge/fog devices to analyze pathological speech data obtained from wearable sensors [32]. Yigitoglu et al. developed a container-enabled Foggy framework that provides automatic resource control in heterogeneous infrastructures such as IoT devices, edge/fog devices, and cloud devices [33]. Bellavista et al. proposed a centralized framework extending Kura framework, and the design includes components running on IoT, edge/fog devices, and cloud devices [34]. Ferrer et al. developed an Adhoc-based framework to support the integration of IoT devices with multi-hop edge/fog devices [35]. Noor et al. developed a centralized container-enabled IoTDoc framework to manage interactions between IoT devices and cloud resources [36].

2.2 Multi-platform Support

Tuli et al. proposed FogBus, which overcomes the difficulty of executing applications in heterogeneous infrastructures by developing the framework in a cross-platform programming language [21]. By doing so, the framework can be barrier-freely deployed on heterogeneous infrastructures without any implementation changes. FogPlan pulls services (applications) which were automatically pushed to the cloud [30]. This design ensures the new versions of their services are always fresh in the cloud database. This deployment mechanism masks the heterogeneity of platforms and enables the migration of their services between hosts. Merlino et al. proposed a middleware platform based on OpenStack, which encompasses edge/fog devices and cloud devices [20]. It is designed for big data processing in a hierarchical design involved cloud devices. In the design of the Foggy framework, the authors present a list of deployment workflow with the assumption that the required environment of applications is maintained by other Orchestration Server [33]. By following the workflow, the framework helps developers reduce the effort to considering the complex heterogeneity of IoT infrastructures, edge/fog infrastructures, and cloud infrastructures.

2.3 Multi-application Support

Nguyen et al. proposed a framework in which the edge/fog devices support content, services, and applications from different providers to serve their customers by proactively distribute the data into edge/fog devices [31]. Bellavista et al. proposed a framework that adds brokers at gateways side trying to solve the problem that a flat topology

is insufficient for both envisioned IoT applications and various real-word applications domain [34].

2.4 Scheduling Mechanism

Ferrer et al. designed a mechanism to utilize the existing capacity of edge/fog devices, which responses to the increasing demands from IoT with a massive volume of data [35]. FogPlan has the scheduling mechanism to deploy or release services by monitoring the incoming traffic and other parameters with the hypothesis that their service controllers only maintain the edge/fog hosts in a particular topographical subnet[30]. Nguyen et al. developed a centralized resource allocation technique that considers the current resources of edge/fog devices and cloud devices[31]. An et al. proposed the SmartFog framework, which uses an unsupervised clustering method analyzing the lower-resources workload on Intel Edison and Raspberry Pi [32]. Foggy integrates a mechanism to scheduling tasks aiming to optimize overall resources utilization, minimize latency between IoT, edge/fog devices, and cloud devices [33].

2.5 Scaling Mechanism

Bellavista et al. use docker containers and the Kubernetes to scale computing infrastructures that support geographically distributed IoT applications and their deployment mechanism [34]. FogPlan supports a simple mechanism of scalability based on the monitored aggregated traffic rate, which uses the ping approach [30]. However, FogPlan does not support policy integration which makes the scaling mechanism fixed and not extensible.

2.6 Resource Discovery

The middleware discovers resources when assumes edge/fog devices join or leave the system unpredictably [20]. Using a Cloud Manager monitoring subsystem, the discovery is triggered when a signal shows the reducing synthetic manipulation performance. FogPlan has mechanism of edge/fog resources discovery. The hosts at edge advertise their IP addresses to IoT devices that run IoT applications [30]. FogPlan also carries URI in communication protocol routing the requests to edge hosts, which achieves resource discovery.

2.7 Containerization Technique

FogPlan has the assumption that the applications running on their framework are containerized, which allows to automate the deployment and to release of services [30]. The services are also considered stateless; thus, migration, deployment, and release of the services are faster than VM-based migration procedures. Merlino et al. containerized edge/fog devices and cloud devices resources in their work [20]. Foggy uses containerization techniques to detach subnets and attempts to decrease the overhead on the constraint resource among hosts over edge/fog devices, and cloud devices [33]. Docker containers and the Kubernetes technologies are used by Bellavista et al. to manage the execution of various IoT applications over the heterogeneous resources of edge/fog devices and cloud devices [34]. Ferrer et al. employ workload virtualization, which is containerized to facilitate the execution in heterogeneous edge/fog devices environments [35]. Noor et al. proposed a framework that is also developed with containerization technology [36].

2.8 Summary

Table 2.1 identifies and compares the main elements of related frameworks with ours. These frameworks often do not support platform-independent techniques and or containerization of software components of the framework and IoT applications. Moreover, most of these frameworks do not offer scheduling, scalability, and resource discovery mechanisms. FogBus2 provides a lightweight and container-enabled distributed framework for computation-intensive and latency-sensitive IoT applications to overcome these limitations. It dynamically schedules heterogeneous IoT applications and scales the resources to serve IoT users efficiently.

Tuli et al. proposed the FogBus framework based on a master-worker approach to process data generated from sensors on edge/fog devices or cloud devices [21]. Due to platform-independent technologies used in the FogBus, it can work on multiple platforms. However, it does not provide any mechanism for dynamic scheduling of IoT applications, scalability, and resource discovery. Besides, it does not support different communication topologies between workers and the master. Moreover, FogBus is not a container-enabled framework, which negatively affects the deployment cost of IoT applications and software components. Yousefpour et al. developed a container-enabled

TABLE 2.1: A qualitative comparison of related works with ours

Work	Integration	Multi Platform Support	Heterogeneous Multi application Support	Dynamic Scheduling Mechanism and Policy Integration	Dynamic Scaling and Policy Integration	Dynamic Resource Discovery	Container Support
[21]	IoT, Edge, Cloud	✓	✕	✕	✕	✕	✕
[30]	IoT, Edge, Cloud	✓	✕	✓	✕	✓	✓
[20]	IoT, Edge, Cloud	✓	✓	✕	✕	✓	✓
[31]	IoT, Edge, Cloud	✓	✓	✓	✕	✕	✕
[22]	IoT, Edge, Cloud	✕	✕	✕	✕	✕	✕
[37]	IoT, Edge, Cloud	✕	✕	✓	✕	✕	✕
[32]	IoT, Edge	✓	✕	✕	✕	✕	✕
[33]	IoT, Edge, Cloud	✓	✕	✓	✕	✕	✓
[34]	IoT, Edge, Cloud	✕	✓	✕	✓	✕	✓
[35]	IoT, Edge	✕	✕	✓	✕	✕	✓
[36]	IoT, Cloud	✕	✕	✓	✕	✕	✓
FogBus2	IoT, Edge, Cloud	✓	✓	✓	✓	✓	✓

framework, called FogPlan, integrating IoT devices with edge/fog devices and cloud devices to minimize the response time of IoT applications [30]. FogPlan supports dynamic resource discovery, scheduling of IoT applications, and simple scalability mechanism and policies. Merlino et al. developed a container-enabled framework for container discovery at edge/fog devices and cloud devices, and horizontal and vertical offloading [20]. However, it does not provide any policies for the dynamic scheduling of IoT applications and the scalability of resources. Nguyen et al. proposed a privacy-preserving framework, which uses obfuscation to keep users' information private meanwhile tasks are computed [31]. Besides, they developed a centralized resource allocation technique that considers the current resources of edge/fog devices and cloud devices. An et al. developed the EiF framework to bring artificial intelligence services to the edge of the network [22]. Although the EiF provides some resource allocation techniques for network resources, it does not offer any scheduling and scalability mechanisms for IoT applications. A mobility-aware framework, called Mobi-IoST, is developed by Ghosh et al. [37], which uses a probabilistic approach for the placement of IoT applications. Borthakur et al. developed the SmartFog framework, integrating IoT devices with edge/fog devices to analyze pathological speech data obtained from wearable sensors [32]. It embeds machine learning techniques to analyze the generated data at the proximity of patients. Yigitoglu et al. developed a container-enabled Foggy framework that supports dynamic scheduling of containerized IoT applications with dependent tasks [33]. Bellavista et al. proposed a centralized container-enabled framework that uses docker containers and

the Kubernetes to scale computing infrastructures [34]. However, it does not provide any policies to support scalability, scheduling, and resource discovery. Moreover, as the cloud orchestrator manages the deployments of applications, it may negatively affect the response time of latency-sensitive IoT applications. Ferrer et al. developed a container-enabled Adhoc-based framework to support the integration of IoT devices with multi-hop edge/fog devices [35]. Noor et al.developed a centralized container-enabled IoTDoc framework to manage interactions between IoT devices and cloud resources [36].

Chapter 3

FogBus2 Framework for IoT-enabled Systems and Applications

This chapter describes the hardware and software components of FogBus2 in detail. Chapter 3.1 presents the hardware components and related design of the framework. Fig. 3.1 presents a high-level overview of computing environment supported by FogBus2. Chapter 3.2 presents components and sub-components of the framework design.

3.1   Hardware Components

FogBus2 supports heterogeneous hardware resources such as different IoT devices, Edge/-Fog servers, and multiple cloud data centers. In Chapter 4, devices of three of the infrastructures in the real world have been used to run FogBu2's components. Since the containerization technique is used in our framework for all the components, FogBus2 should also be compatible with other devices that are supported by the containerization technique. As Fig. 3.1 shows, multiple different subnets are existing over the three layers simultaneously. And one particular host can run several components according to what resources it owns.
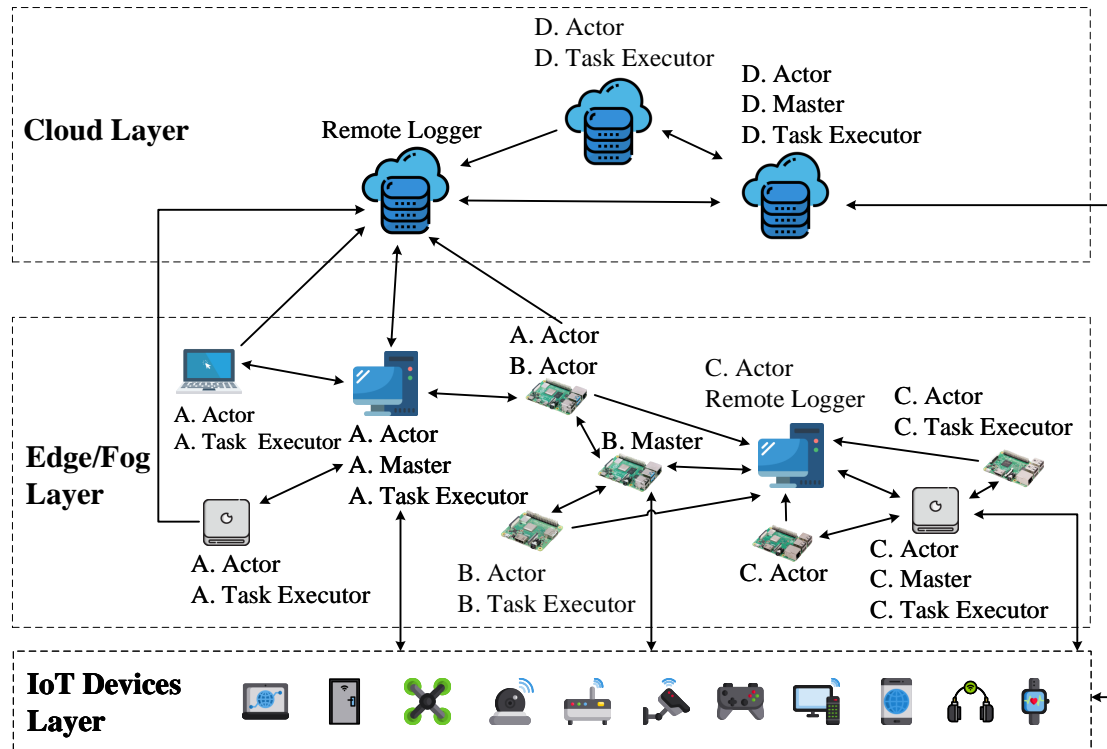
FIGURE 3.1: FogBus2 high-level computing environment

3.1.1 IoT devices layer

IoT devices layer consists of heterogeneous types of resource-limited IoT devices (such as drones, smart cars, smartphones, security cameras, any types of sensors such as humidity sensors, etc) that perceive data from the environment and perform physical actions on the environment. FogBus2 provides a distributed platform for IoT devices to connect with proximate and remote service providers through different communication protocols such as WiFi, Bluetooth, Zigbee, etc. Hence, the generated data from IoT devices can be processed and stored on surrogate servers with higher resources, which significantly helps to reduce the processing time of data generated from IoT devices.

In this layer, the IoT devices play the role of *user* when they sense the data from the internal or external environment and request various applications running on the system. After the requests have been approved and resources placed are ready, devices in this layer send data to the system and receive the result. The actuator of a device itself will finally consume the result, and or actuators in other peers will do.

3.1.2 Edge/Fog layer

FogBus2 provides IoT devices with low-latency and high-bandwidth access to heterogeneous edge/fog resources distributed in their proximity. These heterogeneous edge/fog

servers can be either one-hop away from IoT devices (such as Raspberry pis (RPi), personal computers, etc.) or multi-hop away (such as routers, gateways, etc.). Moreover, to extend the computing and storage capacity of edge/fog servers, FogBus2 supports the collaborative execution of IoT applications among different edge/fog servers in a distributed manner. Hence, FogBus2 offers a wide range of service options for different types of IoT devices with heterogeneous service-level requirements.

FogBus2 components are primarily running in this layer since they are close to IoT devices, with low-latency and high-bandwidth advantages. By collaborating with others hosts in this layer, the components running in this layer dramatically boost the execution of IoT applications.

3.1.3 Cloud layer

FogBus2 expands IoT devices' computing and storage resources by supporting multiple cloud data centers in different geo-location areas, which bring location-independency for IoT applications. Moreover, cloud resources can either be used to process and or store computation and or storage-intensive tasks or when the edge/fog servers resources become overloaded. When the requested application from *user* components from IoT applications is computation-intensive, and there is a lack of resources, the execution of the application is likely to be arranged to the cloud layer.

3.2 Software Components

FogBus2, which is developed from scratch, consists of five main containerized components (using the docker containers technique) developed in Python. Since FogBus2 is a distributed framework, these components can run on different hosts based on the application scenario, as depicted in Fig. 3.1. FogBus2's main components, sub-components (Sub-C), and their respective interactions is shown in Fig. 3.2. In each component, a *message handler* Sub-C is embedded for inter-component communications.

3.2.1 User component

This component runs on *user*s' IoT devices and consists of *sensor* and *actuator*. It can send placement requests to the *master* component for each IoT application, developed
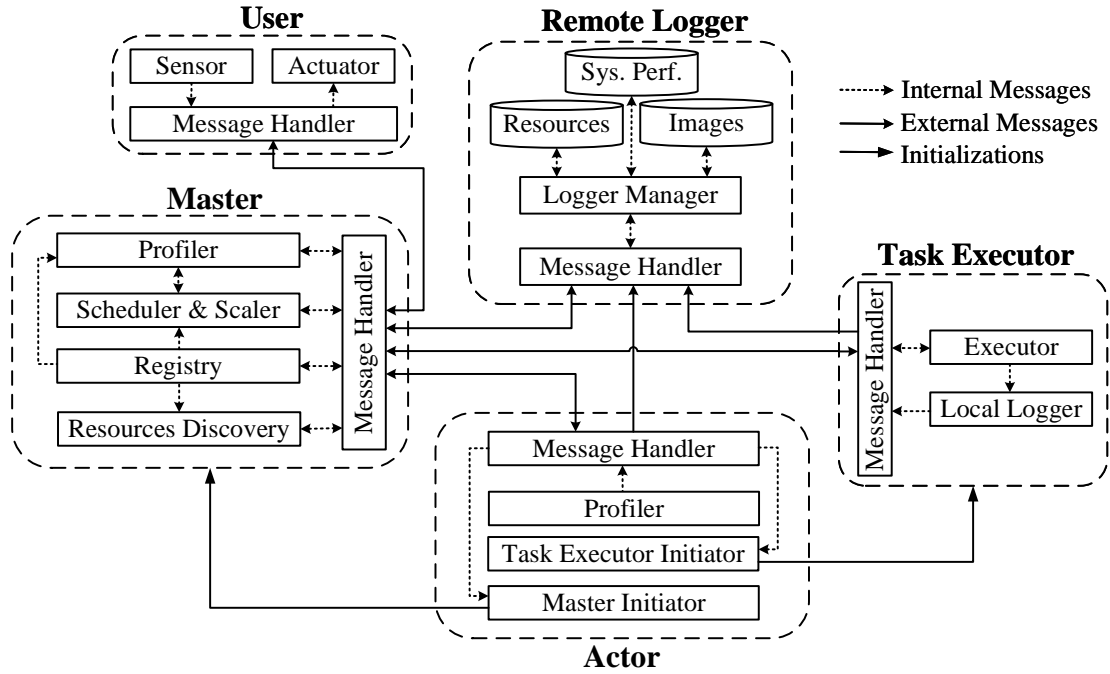
FIGURE 3.2: FogBus2 software components and interactions

with either dependent or independent tasks. Also, it handles the sensors' raw data and collects the processed data from *master*.

3.2.1.1 Sensor

This Sub-C controls the sensing intervals of physical sensors and captures and serializes the sensors' raw data.

3.2.1.2 Actuator

This Sub-C collects processed data from *master* and executes an action based on the application scenario. To support multiple application scenarios, the *actuator* can perform actions in real-time or perform periodic actions based on aggregated data.

3.2.2 Master Component

This component can run on any hosts, either in edge/fog or cloud layers, based on the application scenario. It dynamically profiles the environment and performs resource discovery to find available computing and storage resources. Besides, the *master* component receives placement requests from IoT devices, schedules them, and manages the execution of IoT applications.

3.2.2.1 Registry

When the *master* receives joining requests from *actors* or *task executors*, it records their information and assigns them a unique identifier for the rest of communications. Moreover, it handles placement requests of *users*, assigns them a unique identifier, and initiates the *scheduler & scaler*. The *master* uses each *user*'s unique identifier to distinguish heterogeneous data arriving from other *users*. Also, it can manage authentication mechanisms for the *actors* and *task executors*.

3.2.2.2 Profiler

This Sub-C initially receives information about available resources (such as CPU specifications, RAM), network characteristics (such as average bandwidth and latency), and IoT applications' properties (such as the number of tasks, dependency models) from *registry* Sub-C. Afterward, the *profiler* periodically updates its information from stored data in the *remote logger* component. Moreover, if the required data is not available in the *remote logger* or the *master* requires updated information, it can directly communicate with IoT devices, *actors*, or *task executors* to obtain the data. Also, it keeps track of the status of the *master* and its available resources.

3.2.2.3 Scheduler

When the IoT *user* registered in the *master*, its placement request will be forwarded to the *scheduler & scaler* and will be queued based on First-In-First-Out (FIFO) policy. Algorithm 1 describes the scheduling mechanism and the integrated Optimized History-based Non-dominated Sorting Genetic Algorithm (*OHNSGA*) scheduling policy. The *scheduler* de-queues each placement request based on the FIFO policy. Next, the *scheduler* receives the list of *actors* from the *registry* Sub-C, and continues the scheduling procedure if there exists at least one registered *actor*. Otherwise, it notifies the *user* that there are not enough resources for the scheduling (lines 1-4). Afterward, the *scheduler* examines the local resources of the host. If the CPU utilization is above the threshold (*max_cpu_util*) or the received placement requests exceeds the threshold (*max_shed_count*), it attempts to find a substitute *master* (*sub_master*) to serve this request in order to reduce the waiting time of *user*'s placement request in the queue. If there exists other *master* components in the computing environment, it attempts to find the best *sub_master* (with lowest access latency), otherwise it runs the *scaler* to initiate a new *master* component. (lines 5-12). If the current host has enough resources for the scheduling, the *scheduler* retrieves the application and its dependency model

---

Algorithm 1: Scheduler

/* *req*: user request, *prev_dec*: decisions history, *prof*: hosts profiles,
   curr_sched_count: current scheduling threads count, *max_sched_count*: max
   scheduling threads count, *curr_cpu_util*: current CPU utilization, *max_cpu_util*:
   max CPU utilization, *dependencies*: tasks dependencies, *task_actrs_map*: map task
   to actors   */

1 $actrs \leftarrow$ GETALLACTORS()
2 **if** *actrs is empty* **then**
3 | WARNUSER(*req*)
4 | **return**

5 $curr\_cpu\_util \leftarrow$ GETCPUUTILIZATION()
6 $curr\_sched\_count \leftarrow$ GETSCHEDULECOUNT()
   /* If busy, forward request or scale a new Master   */
7 **if** *curr_cpu_util > max_cpu_util or curr_sched_count > max_sched_count* **then**
8 | $sub\_master \leftarrow$ GETBESTMASTER(*req, masters*)
9 | **if** *sub_master is null* **then**
10 | | $sub\_master \leftarrow$ SCALER(*req, actrs*)
11 | NOTIFYUSER(*req, sub_master*)
12 | **return**

   /* Otherwise schedule   */
13 $dependencies, task\_list \leftarrow$ GETDEPENDENCIESANDTASKLIST(*req*)
14 $i, task\_actrs\_map \leftarrow 0, []$
15 **foreach** *task_list* **do**
16 | $j, task\_actrs\_map[i] \leftarrow 0, []$
17 | **foreach** *actrs* **do**
18 | | **if** *actr has image of task* **then**
19 | | | $task\_actrs\_map[i][j] \leftarrow actr$
20 | | | $j \leftarrow j + 1$
21 | $i \leftarrow i + 1$

   /* Use OHNSGA to schedule   */
22 $prev\_dec \leftarrow$ LOADHISTORY(*req*)
23 $res \leftarrow$ OHNSGA(*prev_dec, pop_size, prof, task_actrs_map, req*)
24 **for** *k from 0 to i − 1* **do**
25 | $actr \leftarrow res[k]$
26 | $task\_exec\_list \leftarrow$ GETIDLELIST(*actr, task_list[k]*)
27 | **if** *task_exec_list is empty* **then**
28 | | SENDINITTASKEXECUTORMSG(*actr, task_list[k], dependencies*)
29 | | **continue**
30 | SENDREUSETASKEXECUTORMSG(*task_exec_list[0], dependencies*)

(for IoT applications with dependent tasks) from the placement request. Moreover, it finds the list of *actors* that can serve each task of an IoT application and stores them in *task_actrs_map* (lines 13-21). The *scheduler* then retrieves the history of previous decisions for this application (line 22). Next, the *scheduler* initiates the *OHNSGA* to find a suitable set of *actors* for the IoT application to minimize its response time. (line 23). The response time of an IoT application is defined as the time difference when a *user* component starts sending data to the time it receives the result.

Algorithm 2: OHNSGA

```
/* hist_ratio:  ratio indicating the number of individuals generated based on history,
   init_pop:  initial population, n_offsprings:  number of offsprings, pop:
   population                                                              */
```

1   $max\_num\_hist\_indv \leftarrow \lceil pop\_size / hist\_ratio \rceil$
2   **if** $len(prev\_dec) > max\_num\_hist\_indv$ **then**
3     $prev\_dec \leftarrow prev\_dec[0 : max\_num\_hist\_indv]$

4   $random\_indv \leftarrow \textsc{RandomIndiv}(pop\_size - len(prev\_dec))$
5   $init\_pop \leftarrow \textsc{Merge}(prev\_dec, random\_indv)$
6   $pop \leftarrow \textsc{RemoveDuplicates}(init\_pop)$
7   **for** i *from 0 to max_iteration_num* **do**
8   **while** *True* **do**
9       $parents \leftarrow \textsc{TournamentSelection}(pop, n\_parents)$
10      $offsprings \leftarrow \textsc{SimBinCrossover}(parents, n\_offsprings)$
11      $offsprings \leftarrow \textsc{PolynomialMutation}(offsprings)$
12      $pop \leftarrow \textsc{Merge}(parents, offsprings)$
13      $pop \leftarrow \textsc{RemoveDuplicates}(pop)$
14   **if** $len(pop) >= pop\_size$ **then**
15        $pop \leftarrow pop\_size[0 : pop\_size]$
16   break

17   $pop \leftarrow \textsc{Sort}(pop)$
18   **return** $pop[0]$

The *OHNSGA* works based on a genetic algorithm (GA) which is a population-based evolutionary algorithm. Each candidate solution for assignments of *actors* to tasks is called an individual, and the set of candidate individuals creates the population. The *OHNSGA* attempts to find better individuals in each iteration of the algorithm to converge to the best solution. *OHNSGA* uses the history of previous decisions of each application to initialize a portion of the first population while the rest of the population is randomly generated. It helps the *OHNSGA* to start from a better initial state and reduces the convergence time of this technique. Also, as a portion of the population is randomly generated, the *OHNSGA* keeps the randomness as well, which significantly helps to jump out of local-optimal solutions. The *OHNSGA* uses the Tournament selection method to find the best individuals in each iteration. Then, to generate the population of the next iteration, *OHNSGA* uses the Simulated Binary Crossover operator, that its efficiency is proved in [38], and Polynomial mutation operator. Algorithm 2 presents an overview of the *OHNSGA*. According to the outcome of *OHNSGA*, the scheduler notifies the actors to run task executors or reuse the available ones for the current IoT application (lines 24-30).

Algorithm 3: Scaler

```
/* my_addr:  address of this host, cpu_util:  CPU utilization, cpu_freq:  CPU
   frequency                                                                  */
```
1   $best\_actr \leftarrow actrs[0]$
2   $cpu\_util \leftarrow$ GetCPUUtilization($best\_actr$)
3   $cpu\_freq \leftarrow$ GetCPUFrequency($best\_actr$)
4   $best\_score \leftarrow (1 - cpu\_util) * cpu\_freq$
5   $min\_latency \leftarrow$ FindLatency($user, best\_actr$)
6   **foreach** *actrs* **do**
7      $latency \leftarrow$ FindLatency($actr$)
8   **if** $latency > min\_latency$ **then**
9   continue
10     $cpu\_util \leftarrow$ GetCPUUtilization($actr$)
11     $cpu\_freq \leftarrow$ GetCPUFrequency($actr$)
12     $score = (1 - cpu\_util) * cpu\_freq$
13   **if** $latency == min\_latency$ *and* $score < best\_score$ **then**
14   continue
15     $best\_actr \leftarrow actr$
16     $best\_score \leftarrow score$
17     $min\_latency \leftarrow latency$
18   SendInitNewMasterMsg($best\_actr, my\_addr$)

3.2.2.4   Scaler

The scaler is called when the current master requires to initiate a new master container. Algorithm 3 depicts how scaler works. The scaler receives the list of registered actors and iterates over them to find the actor with the minimum latency and highest score. The scaler first considers the access latency of actors (line 7). Then, if the latency of the actor is equal to or less than the best-obtained latency, the scaler calculates a score value for that actor. The score value is obtained from current CPU utilization and the average CPU frequency of the host on which the actor is running (lines 8-12). Finally, the scaler selects the actor with the minimum latency whose score is higher and sends a message to the chosen actor to initiate a master container.

3.2.2.5   Resource Discovery

The key responsibility of this Sub-C is to find *master* and *actor* containers in the network. Algorithm 4 describes how resource discovery periodically works. This Sub-C receives the list of its registered *actors* from the *registry* (line 8). Then, it examines the network to find the list of all available neighbors (line 9). Next, this Sub-C checks each neighbor to find running *master* and *actor* containers. If the neighbor runs the *master* container, the resource discovery adds the neighbor to its *known_masters* list and receives the list of registered *actors* on the neighbor (lines 12-14). This mechanism helps *master* containers to automatically know each other in the network and share the information of their registered *actors*. Besides, if the neighbor runs the *actor* container,

---

Algorithm 4: Resource Discovery

```
/* prev_ad_ts:  timestamp of the previous advertising, actrs:  all registered actors
   in current master, neighbours:  neighbours in the network, interval:  discovery
   period                                                                            */
```
1   $prev\_ad\_ts \leftarrow$ TIMESTAMP()
2   **while** *True* **do**
```
      /* Sleep for an interval                                                       */
```
3     $ts \leftarrow$ TIMESTAMP()
4   **if** $ts - prev\_ad\_ts < interval$ **then**
5   SLEEPFORAWHILE()
6   continue
```
      /* Record current timestamp                                                    */
```
7     $prev\_ad\_ts \leftarrow ts$
8     $actrs \leftarrow$ GETALLACTORS()
```
      /* Advertise itself to neighbours                                              */
```
9     $neighbours \leftarrow$ GETALLHOSTS($net\_gateway$, net_mask)
10     $new\_actrs \leftarrow []$
11   **foreach** $neighbours$ **do**
12   **if** $neighbour$ is Master **then**
13         $known\_masters \xleftarrow{+} neighbour$
14         $new\_actrs \xleftarrow{+}$ GETACTORSADDRFROM($neighbour$)
15   **if** $neighbour$ is Actor **then**
16         $new\_actrs \xleftarrow{+} neighbour$
17   **foreach** new_actrs **do**
18   **if** new_actr is not in $actrs$ **then**
19   ADVERTISESELF(new_actr)

---

the address of the *actor* will be recorded in *new_actrs*. Finally, the resource discovery Sub-C advertises the *master* to all *actors* that are not registered in its *actor* list, *actrs* (lines 17-19).

Resource Discovery discovers a component by sending a probing message to an address. Any running component responds with its component information to the source address where the probing message is sent from. With this mechanism, resource discovery scans the network periodically and thus obtain the ability to discover new joined resources.

3.2.3   Actor component

This component can run on any hosts in edge/fog or cloud layers. The *actor* profiles the host's resources and starts the *task executors* for the execution of IoT applications' tasks. Besides, it can initiate the *master* container on the host for the scalability scenarios.

3.2.3.1   Profiler

This *actor's profiler* works the same as the *master's profiler* and records the available resources of the host and network characteristics. However, contrary to *master's profiler*, it does not have profiling information of other hosts. The *actor* periodically sends its profiling information to the *remote logger* component as well as *master* component where it has registered at.

3.2.3.2   Task executor initiator

Whenever a *master* component assigns a task of an IoT application to an *actor* for the execution, the *task executor initiator* is called. It initiates the *task executor* and defines where the results of the *task executor* should be forwarded.

3.2.3.3   Master initiator

This Sub-C is only called when a *master* component (e.g., *master A*) runs its *scaler* procedure and decides to initiate a new *master* component (e.g., *master B*) on other hosts. Hence, the selected *actor* receives a message from its *master* component (*master A*) and runs *master initiator* Sub-C. Then, the *master initiator* runs the *new master* component B. *Master* component B receives the list of registered *actors* from *master* component A to advertise itself. After the initiation of *master* component B, it can also serve the placement requests of IoT *Uers*.

3.2.4   Task Executor Component

IoT applications can be separated into multiple dependent/independent *task executor* containers based on the properties of the IoT application. Thus, an application can be easily deployed on several hosts for distributed execution. Moreover, *task executors* can be efficiently reused for other requests of the same type, which significantly reduces the tasks' deployment time. To obtain this, when a *task executor* finishes the execution of a specific *user's* task, it goes into a cooling-off period. In this period, the container can be reused to serve another request.

3.2.4.1 Executor

The *executor* Sub-C performs the run command to start the task. Also, it sends the results to the dependent children *task executors* (in IoT applications with dependent tasks) or *master* component (when there is no dependency).

3.2.5 Remote Logger Component

To support different application scenarios, this component can run on any hosts in edge/fog or cloud layers. All components send their periodical or event-driven logs to the *Remote Logger*. This component collects the data and stores them in persistent storage, either using a file system or database. The *Remote Logger* can connect to different databases distributed on any hosts, which enable IoT application scenarios that require distributed databases. In our current implementation, however, we run three databases in one host, including images (keeps the information about available docker images on different hosts), resources (keeps the information about hardware specifications of hosts), and system performance (keeps the information about response time, processing time, packet sizes, etc. of IoT applications). Moreover, the databases are containerized for faster deployments. Fig 3.3 depicts an overview of databases and their tables.

3.2.5.1 Logger Manager

The *logger manager* Sub-C receives logs from *masters*, *actors*, and *task executors* and keeps them in the persistent storage. For efficient and quick tracking of logs, the *local manager* keeps the records of system performance, available resources, and containers' information on different storage. Also, *logger manager* Sub-C can provide the latest logs of the system for the *master* components. Besides, the stored logs can be used to analyze the overall status of the system.

3.2.5.2 Database Design

As Fig 3.3 indicates, there are three main databases designed for the working of the FogBus2 framework. The Image Database contains the information of docker images on every host which have been seen by the framework. Image information is profiled by *actor* periodically, and it is sent to *master* within the registration message at the very beginning of the *actor* registers. The image information will also be uploaded periodically to *remote logger* to keep the information updated. Since *master* synchronizes its logs with *remote logger*, *master* will ultimately get the image information of *actors* over
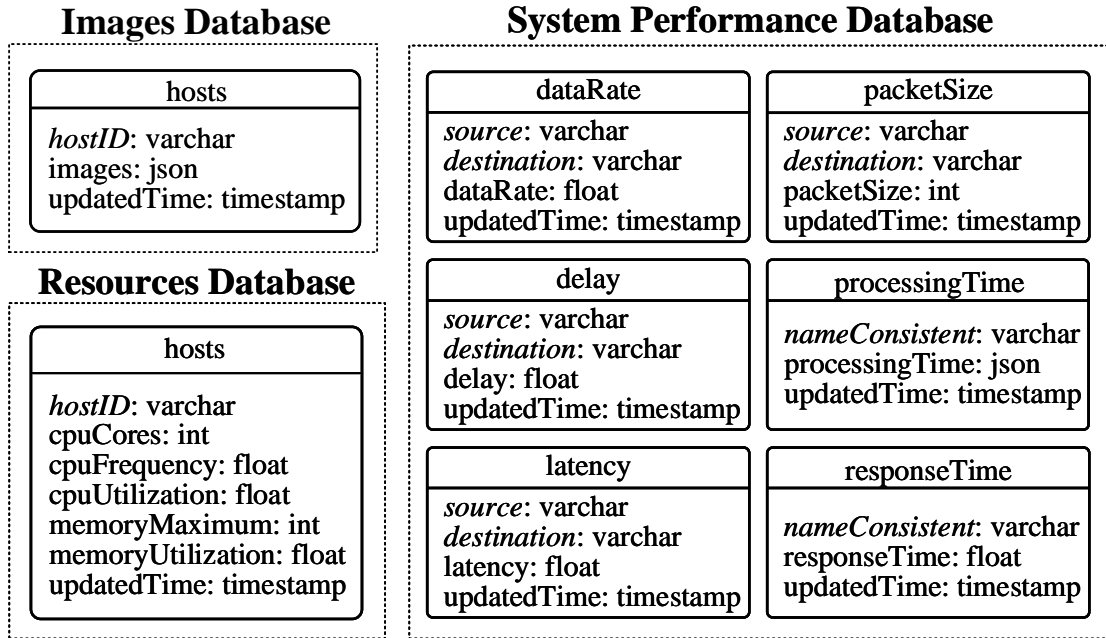
Images Database

System Performance Database

hosts
hostID: varchar images: json updatedTime: timestamp

dataRate
source: varchar *destination*: varchar dataRate: float updatedTime: timestamp

packetSize
source: varchar *destination*: varchar packetSize: int updatedTime: timestamp

Resources Database

delay
source: varchar *destination*: varchar delay: float updatedTime: timestamp

processingTime
nameConsistent: varchar processingTime: json updatedTime: timestamp

hosts
hostID: varchar cpuCores: int cpuFrequency: float cpuUtilization: float memoryMaximum: int memoryUtilization: float updatedTime: timestamp

latency
source: varchar *destination*: varchar latency: float updatedTime: timestamp

responseTime
nameConsistent: varchar responseTime: float updatedTime: timestamp

FIGURE 3.3: Database design

the network. Resources Database keeps the system's dynamic logs such as CPU cores, CPU frequency, CPU utilization, memory capacity, and memory utilization of every seen host. This information help scheduler and scaler work more reasonably, enabling the scheduler and scaler algorithms to find a better host for the workload. Compared with Resources Database and Image Database, System Performance Database is updated more frequently because it contains more dynamic information of the running system, including data rate, packet size, delay and latency between each host pairs, as well as processing time of *task executor* for particular task and response time from *user*'s perspective.

3.3   Interaction Diagram

We have discussed the five components in FogBus2, *user*, *master*, *actor*, *task executor*, and *remote logger*. Figure 3.4 presents the interaction diagram, which explains how an application request from *user* is scheduled and what functionality will be invoked during the procedure. The sequence begins with *user*'s request to *master*. After the *master* receives the request from the *user*, it applies scheduling policies trying to decide how to arrange the execution. Once the scheduling algorithm finishes, the decision will be parsed to the respective *actors* or *task executor*. For *actors*, the placement messages are sent to initiated new *task executor* containers. For *task executor*, the reuse messages
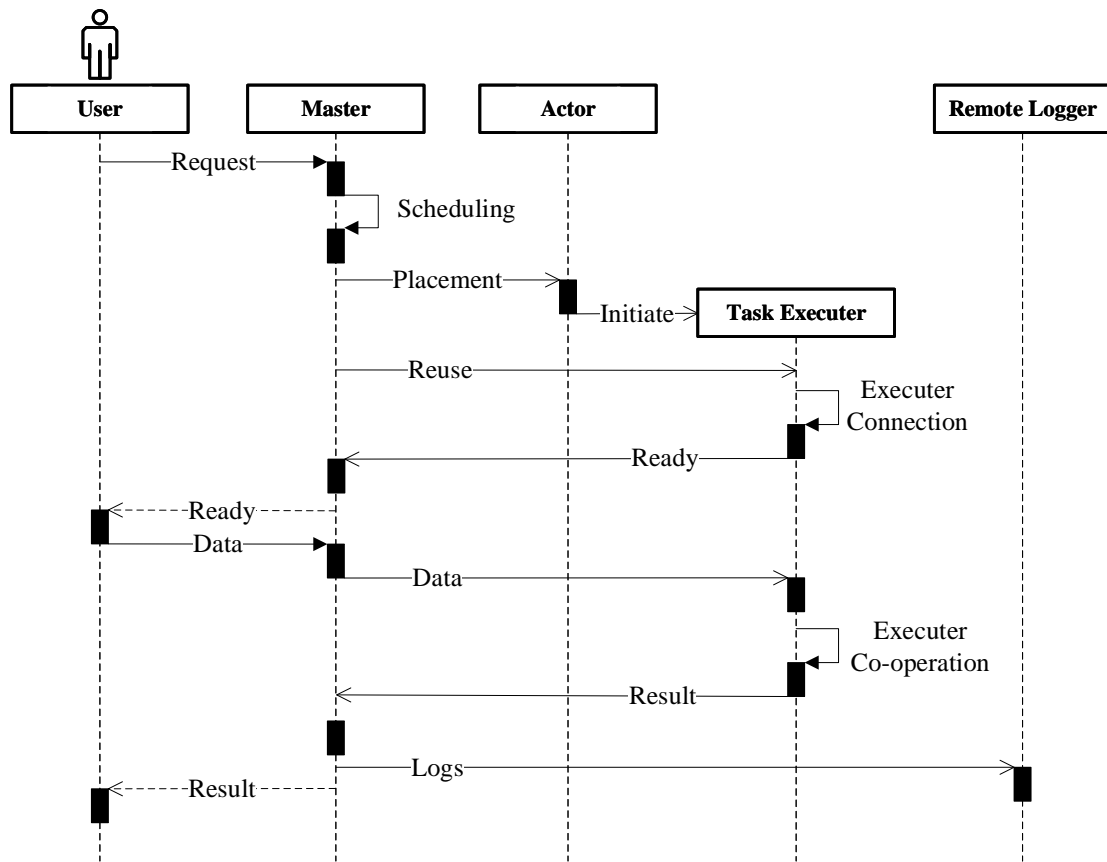
FIGURE 3.4: Interaction Diagram

stop *task executor*s' cool-off period and trigger *task executor* to execute for the new placement. No matter the placement is initiating a new *task executor* or reusing cool-off *task executor*, the *task executor* who received the placement message always connects the dependent peers based on which application is requested. The relationship is contained in the placement message. Next, a *task executor* has to connect to its dependent peers, after which it acknowledges the *master* with a message to indicate its ready state. If all the required *task executor* are ready, *master* then acknowledges *user* that the resources needed for the requested application are all ready. Once the *user* receives the ready message, it starts to send the data that need to be processed to the system and receives the response. The logs during the procedure will be uploaded to *remote logger* for dynamical system performance monitoring, administrator maintenance, and further analysis. To check the list of important messages, which are used in FogBus2 framework, refer to [39].

Chapter 4

Performance Evaluation

This chapter discusses the properties of two sample container-based applications to represent real-time and non-real-time IoT applications. Also, we describe our experiments and evaluate the performance of the FogBus2 framework in real-world environments.

4.1 Sample Container-based Applications

4.1.1 Conway's Game of Life

It is a well-known 2D simulation game that consists of a grid of cells, where each cell can be either black or white. To obtain the next state of the grid, a local function must be applied to each cell simultaneously [40]. In our implementation, each cell is defined as a pixel, and a group of pixels is defined as a rectangle. Our 2d world is separated into several rectangles of different sizes, incurring different computation sizes. Besides, these rectangles have a pyramid structure that defines a dependency model between different rectangles. Hence, we consider Conway's Game of Life as a real-time application with 62 dependent *task executor* containers (one for each rectangle) with different computation sizes.

4.1.2 Video Optical Character Recognition (VOCR)

Compared to the pure OCR application, our implemented VOCR does not require any manual image input from users. The VOCR can either receive a live stream or pre-recorded video and automatically identify key-frames containing text. To filter key-frames, we used two different techniques, called Perceptual Image Hashing (pHash) and Hamming Distance. Then, for each keyframe, the text is extracted using the OCR

technique. Finally, we apply the Editing Distance technique to filter the extracted texts which are similar. Our VOCR application can be used to extract text from books and important information about objects, such as objects in museums. We consider the VOCR as the non-real-time application in its current use-case since the text outputs are not required in real-time for users. However, the VOCR can also be used by smart vehicles in real-time scenarios such as reading traffic signs and warning messages on the road.

4.2   Discussion on Experiments

To study the performance of FogBus2 and its integrated policies, three experiments are conducted. In the first experiment, we analyze the scheduling mechanism of FogBus2 using different scheduling policies. Therefore, we integrate our proposed scheduling policy alongside two other policies in the FogBus2 framework. These policies attempt to approximate the real response time of IoT applications while considering different server configurations and find the best possible server configuration for the execution of IoT applications. Since all integrated scheduling policies are based on evolutionary algorithms, the estimated response time of IoT applications in different iterations is obtained to analyze the convergence rate of different scheduling policies. Moreover, we evaluate the real response time of IoT applications based on the obtained solutions from scheduling policies.

In the second experiment, we analyze the performance of the scalability mechanism of the FogBus2 framework. Typically, IoT integrates thousands and millions of devices that may send their requests to distributed *master* components. These *master* components are geographically distributed, and each one serves several IoT devices so that alongside other *master* components, they can serve thousands or millions of IoT devices. So, in this experiment, IoT devices send a different number of simultaneous placement requests to each one of available *master* components in the environments. Therefore, we study how efficiently the scalability mechanism of the FogBus2 framework can perform when the number of simultaneous requests to each *master* component increases.

In the third experiment, we analyze the performance of the reuse mechanism of the FogBus2 framework. We normalized the resource reuse time by the VOCR without a reuse mechanism by comparing the time to evaluate the performance of the reuse mechanism for complex and straightforward dependent module applications.

In the fourth experiment, we analyze and compare the resource usage of our framework in terms of its startup time and RAM usage with its counterparts.
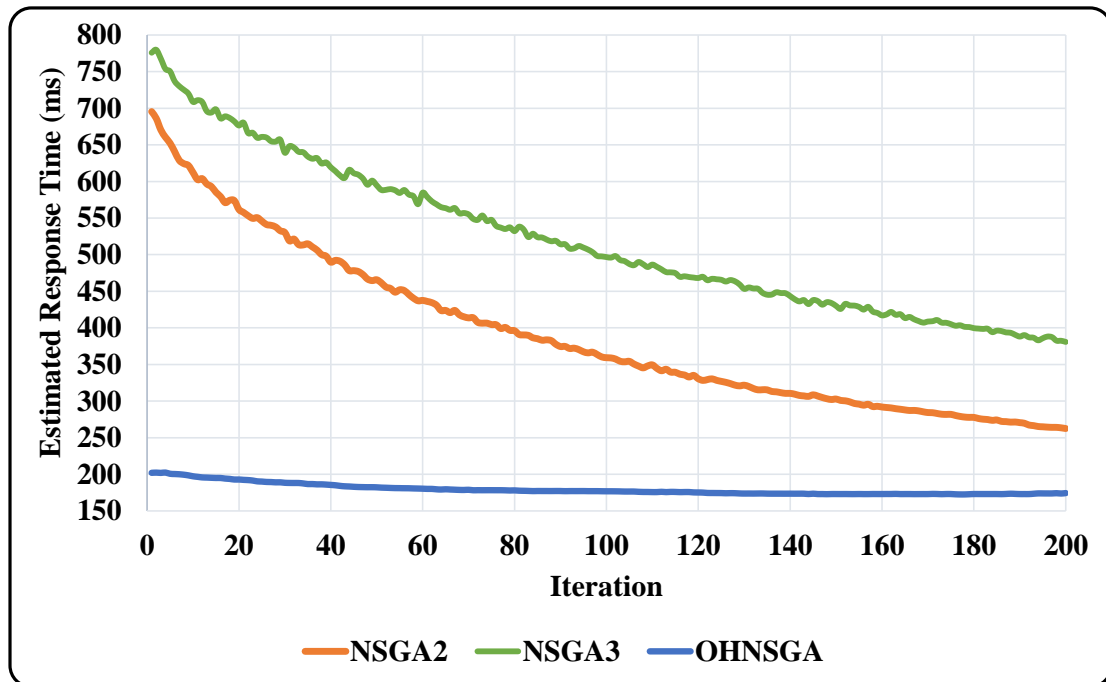
FIGURE 4.1: Scheduling performance in different iterations

4.3 Analysis of Scheduling Policies

This experiment studies the performance of our proposed *OHNSGA* scheduling algorithm and compares it with two other integrated scheduling policies in FogBus2, called Non-dominated Sorting Genetic Algorithm 2 (*NSGA2*) as used in [41], and Non-dominated Sorting Genetic Algorithm 3 (*NSGA3*) [38]. To keep fairness, the parameters of all scheduling policies are the same, including population size, maximum iteration number, and crossover probability.

In this experiment, the environment contains 2 RPi 4B (ARM Cortex-A72 4 cores @1.5GHz CPU, and one with 2GB and another one with 4GB of RAM), and 1 Desktop (Intel Core i7 CPU @3.6GHz and 16 GB of RAM) to show the heterogeneity of servers in the edge layer. Also, the cloud layer contains 2 computing instances provisioned from Huawei Cloud (Intel Xeon 2 cores and 4 cores @2.6GHz CPU with 4GB and 8GB of RAM, respectively). The Desktop acts as a *master* while it also can act as *actor* to start *tasks executors*. The rest of the hosts acts as *actors* and runs *task executors*. *Master profiler* dynamically collects data about network characteristics of the environment (bandwidth and latency), IoT devices, and IoT applications. In this experiment, IoT devices send their requests for the execution of Conway's Game of Life application.

Fig. 4.1 shows the average estimated response time of Conway's Game of Life application, obtained from different policies as the number of iterations increases. The *OHNSGA* outperforms other policies and converges faster to better solutions. *OHNSGA* keeps the
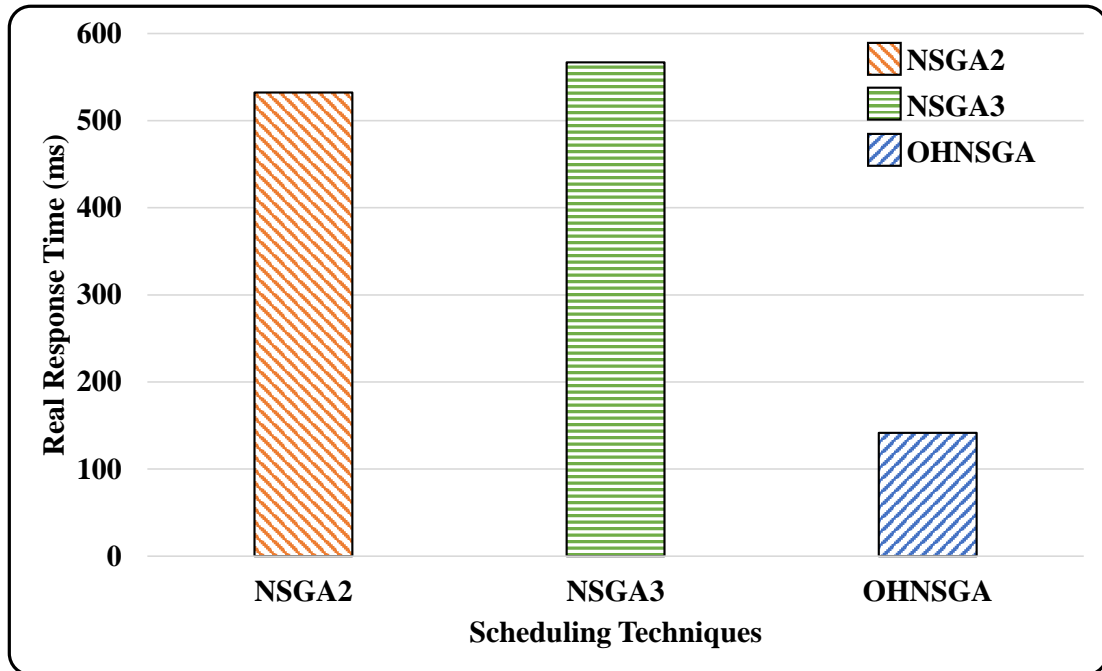
FIGURE 4.2: Real response time of scheduling policies

records of previous decisions and profiling information for each application and initializes a part of its population using its recorded history. Besides, the optimized selection step of *OHNSGA* ensures that non-duplicated best individuals can be copied to the next population. Therefore, *OHNSGA* starts with better individuals compared to *NSGA2* and *NSGA3* due to its more intelligent initialization and keeps its diversity by selecting non-duplicated individuals for the next population. Accordingly, *OHNSGA* can obtain faster convergence to better solutions in comparison to its counterparts.

Fig. 4.2 depicts the real-world response time of Conway's Game of Life application, obtained from the execution of tasks in the real-world environment while considering different scheduling policies. As *OHNSGA* tracks the prior execution behaviors of each application, its obtained real response time is less than other techniques. It proves that not only the *OHNSGA* converge faster to a better solution compared to other policies, but its estimated solutions can better represent the behavior of the Game of Life in real-world environments.

4.4 Analysis of Master Components' Scalability

In this experiment, the environment contains 4 RPi 4B (all with ARM Cortex-A72 4 cores @1.5GHz CPU, where two have 2GB RAM and the other two have 4GB RAM), 1 Desktop (Intel Core(TM) i7 CPU @3.6GHz and 16 GB of RAM) to represent the
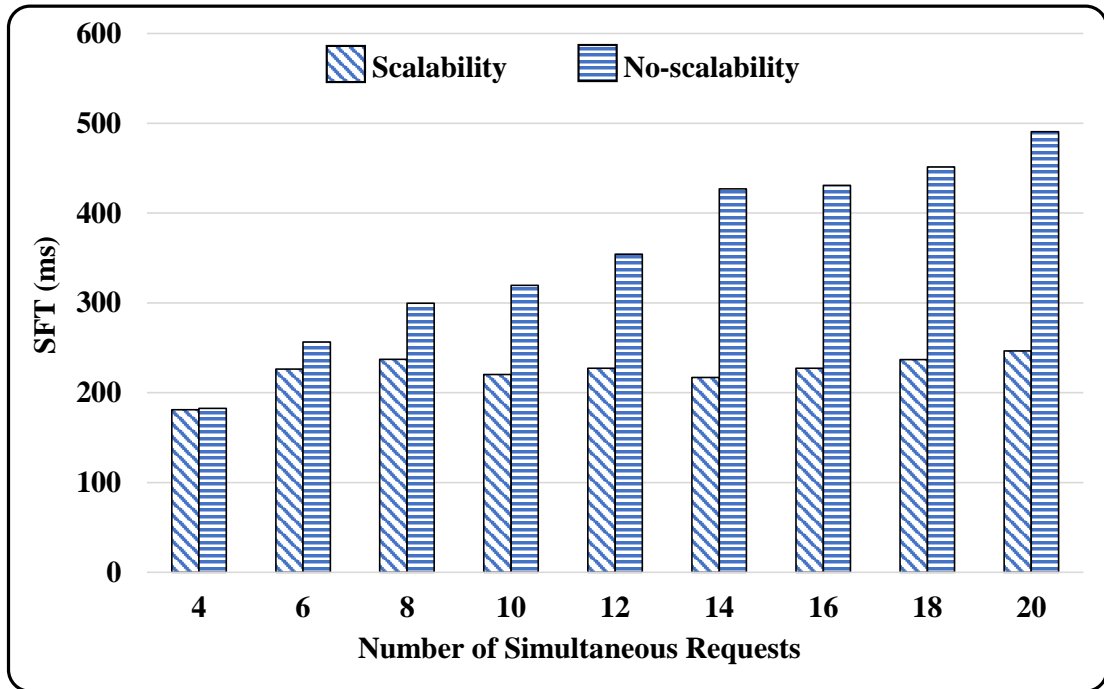
FIGURE 4.3: Analysis of master components' scalability

heterogeneity of servers in the edge layer. Moreover, the cloud layer contains five computing instances provisioned from Huawei Cloud (three instances with Intel Xeon 2 cores @2.6GHz CPU with 4 GB of RAM, and two instances with Intel Xeon 4 cores @2.6GHz CPU with 8 GB RAM). The *master* and *actors* are set as the same as in the previous experiment. Also, IoT devices send simultaneous requests of Conway's Game of Life and VOCR to the *master*. We analyze two scenarios, called scalability and no-scalability. In the scalability scenario, the FogBus2's *master* container scales up either when the number of received IoT requests increases or when the CPU utilization of the host on which the *master* container is running goes above a threshold. The new *master* container can be initiated on any host with sufficient resources, and the rest of the incoming requests can be managed by all available *master* containers. In the no-scalability scenario, incoming requests to the *master* container will be queued until enough resources for scheduling becomes available. Here, we define a Scheduling Finish Time (SFT) metric as the time difference when each IoT device sends its request to the *master* until the *master* container finishes the scheduling of the request. Hence, the SFT contains the queuing time of the request in the *master* plus the scheduling time.

Fig. 4.3 shows the scalability results as the number of simultaneous requests from IoT devices increases. The SFT values of both scenarios are roughly the same when the number of concurrent requests is small. However, as the number of requests increases, the SFT values of the no-scalability scheme dramatically increase compared to the scalability scenario. It shows the importance of supporting scalability mechanisms and policies in
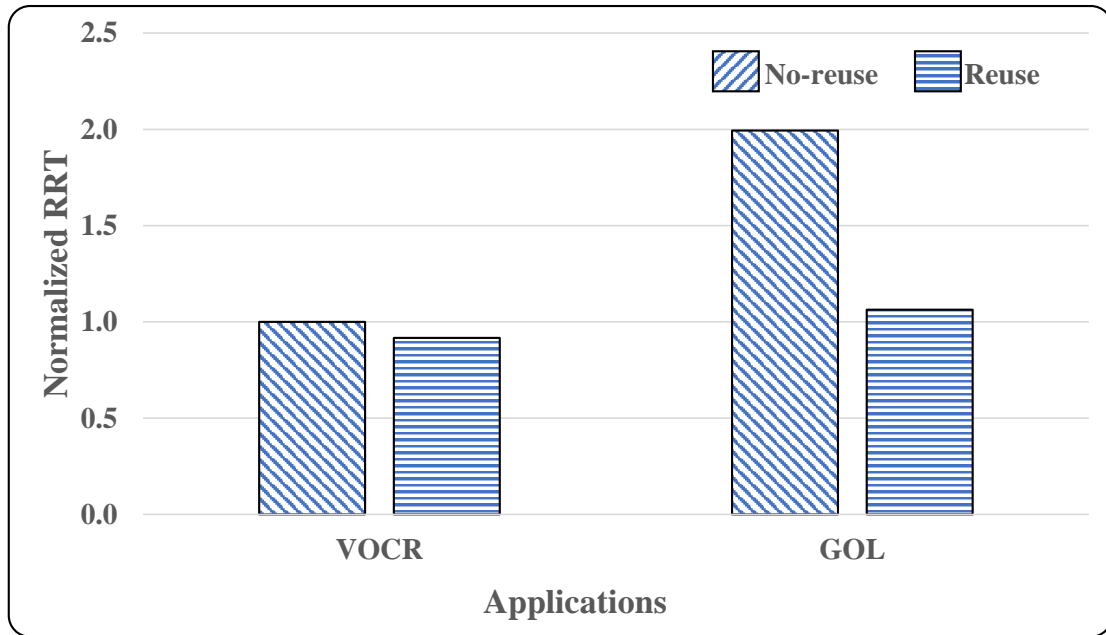
FIGURE 4.4: Analysis of reuse of task executor component; GOL is for Conway's Game of Life

FogBus2. The *master* containers are scaled up as the number of requests increases, which significantly reduces the queuing time of requests.

4.5 Analysis of Reusing Task Executor Components' Container

Fig. 4.4 presents the normalized resource ready time (RRT) of Conway's Game of Life and VOCR. RRT is considered from when $User$ sends the request to when it is informed that required resources are ready, i.e., *task executors* are all ready. The experiment has been conducted on the Desktop only because we only care how much time the mechanism can save when with the reuse mechanism. The RRTs showing in figure 4.4 are normalized by the time of VOCR, which is without a reuse mechanism. For VOCR, a computation-intensive application, the reuse mechanism saves time but slightly. Because VOCR only requires few modules (tasks), i.e., containers, to be prepared. Compared with VOCR, Conway's Game of Life requires various more containers, dramatically raising the RRT without the reuse mechanism. However, when the reuse mechanism is used, even requiring resources to be placed and ready, the RRT of Conway's Game of Life decreases nearly 50 percent, almost equal to VOCR when VOCR requires just a few containers. It proves that the reuse mechanism runs efficiently, particularly for applications with complicated dependencies and relationships, because it avoids repeated container creation and initiation.
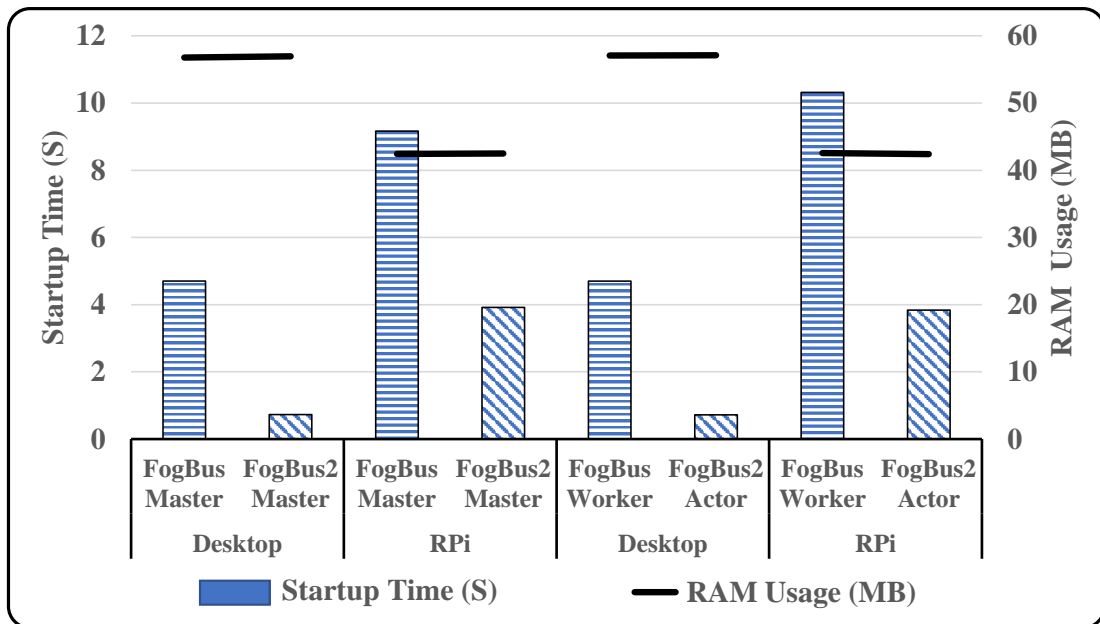
FIGURE 4.5: Startup time and RAM usage analysis

4.6 Analysis of Startup Time and RAM Usage

This experiment studies the startup time and RAM usage of our framework, FogBus2, and compares it with FogBus framework [21]. Fig. 4.5 shows the average startup time and RAM usage of *master* and *actor* components on different hosts. As the results are roughly the same for other components in our framework, we only present the obtained results for these two components. It can be seen the RAM usage of FogBus and our proposed framework, FogBus2, is roughly the same for different framework components. However, the startup time of FogBus2 is roughly 80% and 60% faster compared to FogBus on Desktop and RPi, which makes it a suitable option for fast deployment of any IoT-enabled systems.

Chapter 5

Conclusions and Future Directions

5.1 Conclusions

In this work, we proposed FogBus2, a lightweight and distributed container-based framework to integrate heterogeneous IoT-enabled systems with edge/fog and cloud servers, with the following main contributions:

- **FogBus2 offers fast and low-overhead deployments of applications using containerization.** By using the containerization technique, components of FogBus2 run in isolation environments on edge/fog servers and cloud servers. The runtime environments are secure and lightweight, which benefits FogBu2 and empowers FogBus2 with quick deployment. The required running environments of components are pre-compiled and packaged in images when the images can create as many containers as needed. Moreover, the containerization also decreases the difficulty of designing a scaling mechanism and scaling policies since the needed environments are already prepared in the images.

- **FogBus2 provides scheduling, scalability, resource discovery, and dynamic profiling mechanisms, assisting IoT developers in defining and deploying their targeted IoT applications on FogBus2.** We proposed $OHNSGA$ to dynamically schedule for heterogeneous requested IoT applications when $OHNSGA$ converges extremely fast compared with $NSGA2$ and $NSGA3$, and the decisions of $OHNSGA$ are examined to be better (53% lower response time) in a real-world experiment. Besides, considering the workload growing into the system is unpredictable, an efficient mechanism to automatically scale resources

has to be developed when the investigation shows a 48% improvement compared with no scalability is applied. Moreover, the resource discovery mechanism and dynamic profiling mechanism are integrated into FogBus2 to locate undefined resources and monitor system performance automatically.

- **FogBus2 does not have any constraints on communication topology between its entities and supports different topologies such as mesh, peer-to-peer, and client-server.** The communication of every components in FogBus2 framework design, *master*, *actor*, *user*, *task executor*, and *remote logger*, do not couple with others. For example, *task executor*s can communicate to each other without the participation of *master*, which dramatically increases the efficiency for *task executor*s to co-operate with others when the computation requires content exchanging. This loose design of communication enables various communication topology and make FogBus2 more compatible with different networking environment. It also gives developers more freedom the develop their applications over our framework and contribute to FogBus2.

5.2 Future Directions

Due to modular design and containerization support, IoT developers can easily extend this framework and integrate new software components and policies. Hence, this framework can be further developed by,

- **Integrating dynamic clustering mechanisms and policies to cluster resources either horizontally or vertically.** When seamless integration of edge/-fog and cloud infrastructures has been developed in FogBus2 to support heterogeneous IoT applications, a horizontal and vertical dynamic clustering mechanism can also be integrated upon the integration capability. To intelligently cluster highly heterogeneous resources, a real-world mechanism may be developed learning from simulations like the mechanism proposed in [42].

- **Integrating container-orchestration techniques to automate the management of application deployments and scaling.** FogBus2 manages the containers of different components and applications using the Docker API with the developed algorithm of the scaler. However, automatic management techniques to deploy and scale containers, like Kubernetes [43], can be integrated to obtain more efficient practical performance.

- **Mobility-support in different layers of edge/fog computing environment.** Since IoT devices are usually tiny and some of them are mobile such as in-vehicle

cameras, the mobility support for IoT users and edge/fog servers to provide users seamless and stable experience is also essential. The potential research may refer to Sufyan et al. [44].

- **Integrating lightweight security mechanisms to ensure data confidentiality and integrity** Integration of security mechanisms usually requires extra overhead, but there are several works [21, 45–48] that can be referred to integrate blockchain technique which minimizes the overhead on security monitoring and also keeps tracking the sensitive information during the execution of applications and serving of the framework.

- **Privacy preservation support for the users' private information and edge/fog servers.** Privacy preservation is significant for IoT applications when the IoT devices are close to the natural environment and to humans. For example, healthcare applications highly concern about privacy and security because the applications service and run with patients' sensitive data. When the transmission of such data is over the edge networking, the protection and preservation need to be considered, which can be referred to Bakkiam et al. [49] and Sahi et al. [50].

- **Integrating machine learning techniques to analyze the current state of edge/fog computing environment.** It is difficult to understand and efficiently manage an edge/fog computing environment because the current state of such distributed system is incredibly dynamic and complex. Referring to [14, 51–53], the machine learning techniques can be integrated into the FogBus2 framework to understand and analyze the system, as a consequence, develop and improve policies for scheduling, scaling, and resource discovery mechanism.

Bibliography

[1] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of Network and Computer Applications*, 98:27–42, 2017.

[2] Andrew A Cook, Göksel Mısırlı, and Zhong Fan. Anomaly detection for iot time-series data: A survey. *IEEE Internet of Things Journal*, 7(7):6481–6494, 2019.

[3] Shuiguang Deng, Zhengzhe Xiang, Javid Taheri, Khoshkholghi Ali Mohammad, Jianwei Yin, Albert Zomaya, and Schahram Dustdar. Optimal application deployment in resource constrained distributed edges. *IEEE Transactions on Mobile Computing*, 2020.

[4] Mohammad Goudarzi, Zeinab Movahedi, and Masoud Nazari. Mobile cloud computing: a multisite computation offloading. In *2016 8th International Symposium on Telecommunications (IST)*, pages 660–665. IEEE, 2016.

[5] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. High-performance cloud computing: A view of scientific applications. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 4–16, 2009. doi: 10.1109/I-SPAN.2009.150.

[6] Borivoje Furht, Armando Escalante, et al. *Handbook of cloud computing*, volume 3. Springer, 2010.

[7] Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. High performance cloud computing. *Future Generation Computer Systems*, 29(6):1408–1416, 2013.

[8] Xiaolong Xu, Qingxiang Liu, Yun Luo, Kai Peng, Xuyun Zhang, Shunmei Meng, and Lianyong Qi. A computation offloading method over big data for iot-enabled cloud-edge computing. *Future Generation Computer Systems*, 95:522–533, 2019.

[9] Mohammad Goudarzi, Huaming Wu, Marimuthu S Palaniswami, and Rajkumar Buyya. An application placement technique for concurrent iot applications in edge and fog computing environments. *IEEE Transactions on Mobile Computing*, 20(4):1298 – 1311, 2021.

[10] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *2009 International Conference on High Performance Computing Simulation*, pages 1–11, 2009. doi: 10.1109/HPCSIM.2009.5192685.

[11] Rajkumar Buyya. Market-oriented cloud computing: vision, hype, and reality of delivering computing as the 5th utility. In *2009 Fourth ChinaGrid Annual Conference*, pages xii–xv, 2009. doi: 10.1109/ChinaGrid.2009.6.

[12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755–768, 2012. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2011.04.017. Special Section: Energy efficiency in large-scale distributed systems.

[13] Thar Baker, Muhammad Asim, Hissam Tawfik, Bandar Aldawsari, and Rajkumar Buyya. An energy-aware service composition algorithm for multiple cloud-based iot applications. *Journal of Network and Computer Applications*, 89:96–108, 2017. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2017.03.008. Emerging Services for Internet of Things (IoT).

[14] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. A fog-driven dynamic resource allocation technique in ultra dense femtocell networks. *Journal of Network and Computer Applications*, 145:102407, 2019. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2019.102407.

[15] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1):450–465, 2018. doi: 10.1109/JIOT.2017.2750180.

[16] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1): 30–39, 2017. doi: 10.1109/MC.2017.9.

[17] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. doi: 10.1109/JIOT.2016.2579198.

[18] Ishan Mistry, Sudeep Tanwar, Sudhanshu Tyagi, and Neeraj Kumar. Blockchain for 5g-enabled iot for industrial automation: A systematic review, solutions, and challenges. *Mechanical Systems and Signal Processing*, 135:106382, 2020. ISSN 0888-3270. doi: https://doi.org/10.1016/j.ymssp.2019.106382.

[19] Dan Wang, Dong Chen, Bin Song, Nadra Guizani, Xiaoyan Yu, and Xiaojiang Du. From iot to 5g i-iot: The next generation iot-based intelligent algorithms

and 5g technologies. *IEEE Communications Magazine*, 56(10):114–120, 2018. doi: 10.1109/MCOM.2018.1701310.

[20] Giovanni Merlino, Rustem Dautov, Salvatore Distefano, and Dario Bruneo. Enabling workload engineering in edge, fog, and cloud computing through openstack-based middleware. *ACM Transactions on Internet Technology (TOIT)*, 19(2):1–22, 2019.

[21] Shreshth Tuli, Redowan Mahmud, Shikhar Tuli, and Rajkumar Buyya. Fogbus: A blockchain-based lightweight framework for edge and fog computing. *Journal of Systems and Software*, 154:22–36, 2019.

[22] JongGwan An, Wenbin Li, Franck Le Gall, Ernoe Kovac, Jaeho Kim, Tarik Taleb, and JaeSeung Song. Eif: Toward an elastic iot fog framework for ai services. *IEEE Communications Magazine*, 57(5):28–33, 2019.

[23] Qifan Deng, Mohammad Goudarzi, and Rajkumar Buyya. Fogbus2: a lightweight and distributed container-based framework for integration of iot-enabled systems with edge and cloud computing. In *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments*, pages 1–8, 2021.

[24] Rafael Auler, Carlos Eduardo Millani, Alexandre Brisighello, Alisson Linhares, and Edson Borin. Handling iot platform heterogeneity with coisa, a compact openisa virtual platform. *Concurrency and Computation: Practice and Experience*, 29(22): e3932, 2017.

[25] Zhiwei Zhao, Geyong Min, Weifeng Gao, Yulei Wu, Hancong Duan, and Qiang Ni. Deploying edge computing nodes for large-scale iot: A diversity aware approach. *IEEE Internet of Things Journal*, 5(5):3606–3614, 2018.

[26] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220, 2002.

[27] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 13–31. Springer, 2010.

[28] Soumen Chakrabarti, Martin Van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific web resource discovery. *Computer networks*, 31 (11-16):1623–1640, 1999.

[29] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614. IEEE, 2014.

[30] Ashkan Yousefpour, Ashish Patil, Genya Ishigaki, Inwoong Kim, Xi Wang, Hakki C Cankaya, Qiong Zhang, Weisheng Xie, and Jason P Jue. Fogplan: a lightweight qos-aware dynamic fog service provisioning framework. *IEEE Internet of Things Journal*, 6(3):5080–5096, 2019.

[31] Duong Tung Nguyen, Long Bao Le, and Vijay K Bhargava. A market-based framework for multi-resource allocation in fog computing. *IEEE/ACM Transactions on Networking*, 27(3):1151–1164, 2019.

[32] Debanjan Borthakur, Harishchandra Dubey, Nicholas Constant, Leslie Mahler, and Kunal Mankodiya. Smart fog: Fog computing framework for unsupervised clustering analytics in wearable internet of things. In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 472–476. IEEE, 2017.

[33] Emre Yigitoglu, Mohamed Mohamed, Ling Liu, and Heiko Ludwig. Foggy: A framework for continuous automated iot application deployment in fog computing. In *2017 IEEE International Conference on AI & Mobile Services*, pages 38–45. IEEE, 2017.

[34] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In *Proceedings of the 18th international conference on distributed computing and networking*, pages 1–10, 2017.

[35] Ana Juan Ferrer, Joan Manuel Marques, and Josep Jorba. Ad-hoc edge cloud: A framework for dynamic creation of edge computing infrastructures. In *2019 28th International Conference on Computer Communication and Networks*, pages 1–7. IEEE, 2019.

[36] Shahid Noor, Bridget Koehler, Abby Steenson, Jesus Caballero, David Ellenberger, and Lucas Heilman. Iotdoc: A docker-container based architecture of iot-enabled cloud system. In *3rd IEEE/ACIS International Conference on Big Data, Cloud Computing, and Data Science Engineering*, pages 51–68. Springer, 2019.

[37] Shreya Ghosh, Anwesha Mukherjee, Soumya K Ghosh, and Rajkumar Buyya. Mobiiost: mobility-aware cloud-fog-edge-iot collaborative framework for time-critical applications. *IEEE Transactions on Network Science and Engineering*, 2019.

[38] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE transactions on evolutionary computation*, 18(4):577–601, 2013.

[39] Mohammad Goudarzi, Qifan Deng, and Rajkumar Buyya. Resource management in edge and fog computing using fogbus2 framework. *arXiv preprint arXiv:2108.00591*, 2021.

[40] Paul Rendell. Turing universality of the game of life. In *Collision-based computing*, pages 513–539. Springer, 2002.

[41] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[42] Dilip Kumar, Trilok C Aseri, and RB2009 Patel. Eehc: Energy efficient heterogeneous clustered scheme for wireless sensor networks. *computer communications*, 32 (4):662–667, 2009.

[43] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[44] Sufyan Almajali, Haythem Bany Salameh, Moussa Ayyash, and Hany Elgala. A framework for efficient and secured mobility of iot devices in mobile edge computing. In *2018 third international conference on fog and mobile edge computing (FMEC)*, pages 58–62. IEEE, 2018.

[45] Muneeb Ul Hassan, Mubashir Husain Rehmani, and Jinjun Chen. Privacy preservation in blockchain based iot systems: Integration issues, prospects, challenges, and future research directions. *Future Generation Computer Systems*, 97:512–529, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2019.02.060.

[46] Ali Dorri, Salil S. Kanhere, Raja Jurdak, and Praveen Gauravaram. Blockchain for iot security and privacy: The case study of a smart home. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 618–623, 2017. doi: 10.1109/PERCOMW.2017.7917634.

[47] Oscar Novo. Blockchain meets iot: An architecture for scalable access management in iot. *IEEE Internet of Things Journal*, 5(2):1184–1195, 2018. doi: 10.1109/JIOT. 2018.2812239.

[48] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Towards an optimized blockchain for iot. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 173–178, 2017.

[49] B. D. Deebak, Fadi Al-Turjman, Moayad Aloqaily, and Omar Alfandi. An authentic-based privacy preservation protocol for smart e-healthcare systems in iot. *IEEE Access*, 7:135632–135649, 2019. doi: 10.1109/ACCESS.2019.2941575.

[50] Muneeb Ahmed Sahi, Haider Abbas, Kashif Saleem, Xiaodong Yang, Abdelouahid Derhab, Mehmet A. Orgun, Waseem Iqbal, Imran Rashid, and Asif Yaseen. Privacy preservation in e-healthcare environments: State of the art and future directions. *IEEE Access*, 6:464–478, 2018. doi: 10.1109/ACCESS.2017.2767561.

[51] Shreshth Tuli, Nipam Basumatary, Sukhpal Singh Gill, Mohsen Kahani, Rajesh Chand Arya, Gurpreet Singh Wander, and Rajkumar Buyya. Healthfog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated iot and fog computing environments. *Future Generation Computer Systems*, 104:187–200, 2020. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2019.10.043.

[52] Shreshth Tuli, Shashikant Ilager, Kotagiri Ramamohanarao, and Rajkumar Buyya. Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks. *IEEE Transactions on Mobile Computing*, 2020.

[53] Muhammad Hafizhuddin Hilman, Maria Alejandra Rodriguez, and Rajkumar Buyya. Task runtime prediction in scientific workflows using an online incremental learning approach. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 93–102, 2018. doi: 10.1109/UCC.2018.00018.