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

Conference Paper · June 2021

DOI: 10.1145/3460866.3461768

Qifan Deng

University of Melbourne

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

Qifan Deng, Mohammad Goudarzi and Rajkumar Buyya
The Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia

ABSTRACT

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.

KEYWORDS

Internet of Things, Edge/Fog Computing, Containers Scheduling, Scalability

ACM Reference Format:
Qifan Deng, Mohammad Goudarzi and Rajkumar Buyya. 2021. FogBus2: A Lightweight and Distributed Container-based Framework for Integration of IoT-enabled Systems with Edge and Cloud Computing. In *Big Data in Emergent Distributed Environments (BiDEDE'21), June 20, 2021, Xi'an, Shaanxi, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3460866.3461768

BiDEDE'21, June 20, 2021, Xi'an, Shaanxi, China

ACM ISBN 978-1-4503-8465-0/21/06…$15.00
https://doi.org/10.1145/3460866.3461768

1 INTRODUCTION

Internet of Things (IoT) devices have become an inseparable part of our daily lives, where IoT applications provide diverse solutions for smart healthcare, transportation, and entertainment, just to mention a few [1]. IoT applications often produce a huge amount of data for processing and storage. 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 [2, 3]. IoT devices can place whole or some parts of their applications to cloud servers (CSs) for processing and storage. However, 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 [4, 5]. To address this issue, 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 [6]. Compared to CSs' resources, edge/fog servers (ESs) have limited computing and storage resources, and hence they cannot efficiently execute computation-intensive tasks of IoT devices. To address this issue, ESs can collaboratively use their resources or use CSs. Thus, seamless integration of edge/fog and cloud infrastructures to support different IoT applications is an important research topic.

Resources of distributed ESs and CSs are highly heterogeneous in terms of computing capabilities, processors' architectures, RAM capacity, and supported communication protocols [7]. 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.

Although there are some frameworks to manage integrated resources in edge/fog computing [8, 9], they barely consider platform-independent techniques, scheduling of heterogeneous IoT applications with complex dependent structures, scalability mechanisms of distributed resource managers, and containerization. To address these limitations, we propose and develop a lightweight and distributed container-based framework, called FogBus2. Our framework supports (1) different inter and intra interaction models among ESs and CSs to support the requirements of different IoT application scenarios, (2) containerization of software components of the framework for fast deployments, (3) containerization of constituent parts of IoT applications as dependent tasks or independent tasks, (4) scheduling of multiple IoT applications and scalability mechanisms (5) concurrent execution of different types of IoT applications, 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).

The rest of the paper is organized as follows. Relevant frameworks are described in Section 2. Section 3 presents the hardware and software components of the FogBus2 and their detailed implementations. The performance of FogBus2 is evaluated in Section 4. Finally, Section 5 concludes the paper and draws future works.

2 RELATED WORK

This section discusses related frameworks integrating IoT-enabled systems with edge/fog and cloud infrastructures.

Tuli et al. [8] proposed the FogBus framework based on a master-worker approach to process data generated from sensors on ESs or CSs. 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. [10] developed a container-enabled framework, called FogPlan, integrating IoT devices with ESs and CSs to minimize the response time of IoT applications. FogPlan supports dynamic resource discovery and scheduling of IoT applications, however, it does not provide any scalability mechanism and policies. Merlino et al. [7] developed a container-enabled framework for container discovery at ESs and CSs, and horizontal and vertical offloading. However, it does not provide any policies for the dynamic scheduling of IoT applications and the scalability of resources. Nguyen et al. [11] proposed a privacy-preserving framework, which uses obfuscation to keep users' information private meanwhile tasks are computed. Besides, it developed a centralized resource allocation technique that considers the current resources of ESs and CSs. An et al. [9] developed the EiF framework to bring artificial intelligence services to the edge of the network. 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. [12], which uses a probabilistic approach for the placement of IoT applications. Borthakur et al. [13] developed the SmartFog framework, integrating IoT devices with ESs to analyze pathological speech data obtained from wearable sensors. It embeds machine learning techniques to analyze the generated data at the proximity of patients. Yigitoglu et al. developed a container-enabled Foggy framework [14] that supports dynamic scheduling of containerized IoT applications with dependent tasks. Bellavista et al. [15] proposed a centralized container-enabled framework that uses docker containers and the Kubernetes to scale computing infrastructures. 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. [16] developed a container-enabled Adhoc-based framework to support the integration of IoT devices with multi-hop ESs. Noor et al. [17] developed a centralized container-enabled IoTDoc framework to manage interactions between IoT devices and cloud resources.

Table 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
[8]	IoT, Edge, Cloud	✓	×	×	×	×	×
[10]	IoT, Edge, Cloud	×	×	✓	×	✓	✓
[7]	IoT, Edge, Cloud	✓	✓	×	×	✓	✓
[11]	IoT, Edge, Cloud	×	✓	✓	×	×	×
[9]	IoT, Edge, Cloud	×	×	×	×	×	×
[12]	IoT, Edge, Cloud	×	×	✓	×	×	×
[13]	IoT, Edge	✓	×	×	×	×	×
[14]	IoT, Edge, Cloud	✓	×	✓	×	×	✓
[15]	IoT, Edge, Cloud	×	✓	×	✓	×	✓
[16]	IoT, Edge	×	×	✓	×	×	✓
[17]	IoT, Cloud	×	×	✓	×	×	✓
FogBus2	IoT, Edge, Cloud	✓	✓	✓	✓	✓	✓

Table 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

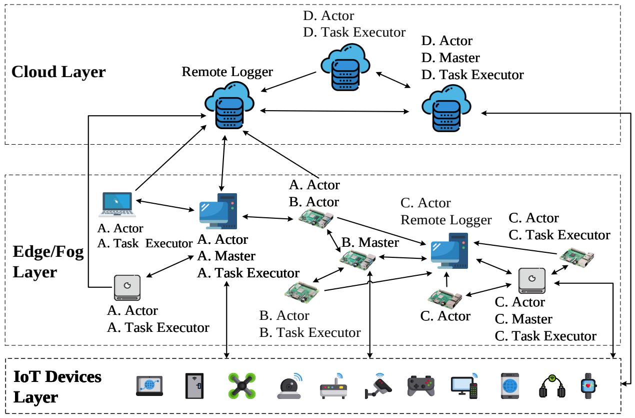

Figure 1: FogBus2 high-level computing environment

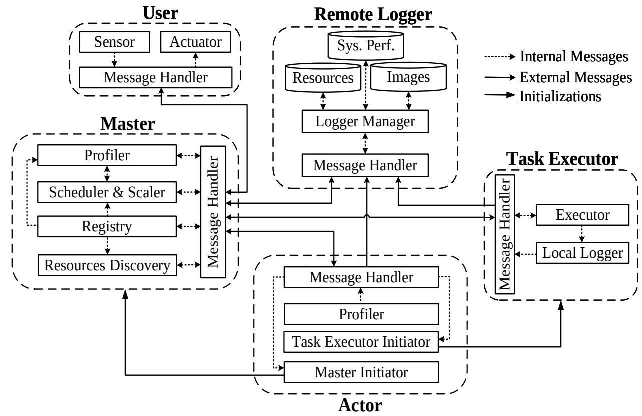

Figure 2: FogBus2 software components and interactions

resource discovery mechanisms. To overcome these limitations, FogBus2 offers a lightweight and container-enabled distributed framework for computation-intensive and latency-sensitive IoT applications. It dynamically schedules heterogeneous IoT applications and scales the resources to efficiently serve IoT users.

3 FOGBUS2 FRAMEWORK

This section describes the hardware and software components of FogBus2 in detail. Fig. 1 presents a high-level overview of computing environment supported by FogBus2.

3.1 Hardware Components

FogBus2 supports heterogeneous hardware resources such as different IoT devices, Edge/Fog servers, and multiple cloud data-centers.

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.

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 ESs 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 ESs, FogBus2 supports the collaborative execution of IoT applications among different ESs in a distributed manner. Hence, FogBus2 offers a wide range of service options for different types of IoT devices with heterogeneous service-level requirements.

3.1.3 ***Cloud layer.*** FogBus2 expands the computing and storage resources of IoT devices 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 ESs resources become overloaded.

3.2 Software Components

FogBus2 consists of five main containerized components (using docker containers) 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. 1. FogBus2's main components, sub-components (Sub-C) and their respective interactions is shown in Fig. 2. In each component, a *message handler* Sub-C is embedded for inter-component communications.

3.2.1 ***User component.*** This component runs on users' IoT devices and consists of *sensor* and *actuator*. It can send placement requests to the *master* component for each IoT application, developed with either dependent or independent tasks. Also, it handles the sensors' raw data and collects the processed data from *master*.

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

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.

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*.

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.

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 (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.

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 [18], and Polynomial mutation operator. Algorithm 2 presents an

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 ← GetAllActors()
2  if actrs is empty then
3      WarnUser(req)
4      return
5  curr_cpu_util ← GetCPUUtilization()
6  curr_sched_count ← 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 ← GetBestMaster(req, masters)
9      if sub_master is null then
10         sub_master ← Scaler(req, actrs)
11     NotifyUser(req, sub_master)
12     return
   /* Otherwise schedule                                      */
13 dependencies, task_list ← GetDependenciesAndTaskList(req)
14 i, task_actrs_map ← 0, []
15 foreach task_list do
16     j, task_actrs_map[i] ← 0, []
17     foreach actrs do
18         if actr has image of task then
19             task_actrs_map[i][j] ← actr
20             j ← j + 1
21     i ← i + 1
   /* Use OHNSGA to schedule                                  */
22 prev_dec ← LoadHistory(req)
23 res ← OHNSGA(prev_dec, pop_size, prof, task_actrs_map, req )
24 for k from 0 to i − 1 do
25     actr ← res[k]
26     task_exec_list ← 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)
```

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).

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 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 selected actor to initiate a master container.

Resources 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,

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 */
max_num_hist_indv ← ⌈pop_size/hist_ratio⌉
if len(prev_dec) > max_num_hist_indv then
    prev_dec ← prev_dec[0 : max_num_hist_indv]
random_indv ← RandomIndiv(pop_size − len(prev_dec))
init_pop ← Merge(prev_dec, random_indv)
pop ← RemoveDuplicates(init_pop)
for i from 0 to max_iteration_num do
    while True do
        parents ← TournamentSelection(pop, n_parents)
        offsprings ← SimBinCrossover(parents, n_offsprings)
        offsprings ← PolynomialMutation(offsprings)
        pop ← Merge(parents, offsprings)
        pop ← RemoveDuplicates(pop)
        if len(pop) >= pop_size then
            pop ← pop_size[0 : pop_size]
            break
pop ← Sort(pop)
return pop[0]
```

Algorithm 3: Scaler

```
/* my_addr: address of this host, cpu_util: CPU utilization,
   cpu_freq: CPU frequency */
best_actr ← actrs[0]
cpu_util ← GetCPUUtilization(best_actr)
cpu_freq ← GetCPUFrequency(best_actr)
best_score ← (1 − cpu_util) ∗ cpu_freq
min_latency ← FindLatency(user, best_actr)
foreach actrs do
    latency ← FindLatency(actr)
    if latency > min_latency then
        continue
    cpu_util ← GetCPUUtilization(actr)
    cpu_freq ← GetCPUFrequency(actr)
    score = (1 − cpu_util) ∗ cpu_freq
    if latency == min_latency and score < best_score then
        continue
    best_actr ← actr
    best_score ← score
    min_latency ← latency
SendInitNewMasterMsg(best_actr, my_addr)
```

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 ← Timestamp()
2  while True do
       /* Sleep for an interval */
3      ts ← Timestamp()
4      if ts − prev_ad_ts < interval then
5          SleepForAWhile()
6          continue
       /* Record current timestamp */
7      prev_ad_ts ← ts
8      actrs ← GetAllActors()
       /* Advertise itself to neighbours */
9      neighbours ← GetAllHosts(net_gateway, net_mask)
10     new_actrs ← []
11     foreach neighbours do
12         if neighbour is Master then
13             known_masters +← neighbour
14             new_actrs +← GetActorsAddrFrom(neighbour)
15         if neighbour is Actor then
16             new_actrs +← neighbour
17     foreach new_actrs do
18         if new_actr is not in actrs then
19             AdvertiseSelf(new_actr )
```

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, 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).

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.

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.

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.

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 *users*.

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 reduce 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.

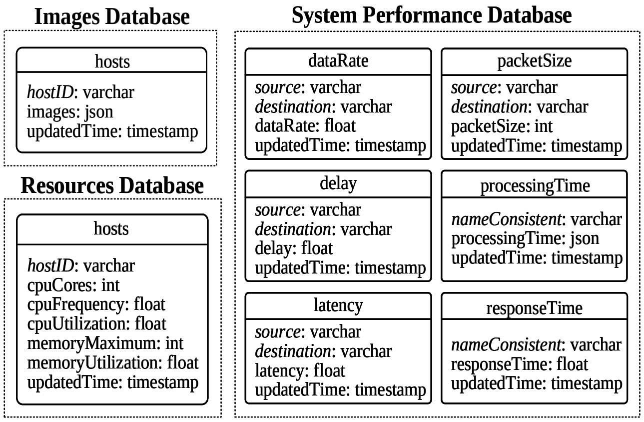

Figure 3: Database design

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 which require distributed databases. In our current implementation, however, we run three databases in one host, including images (keeps information about available docker images on different hosts), resources (keeps information about hardware specifications of hosts), and system performance (keeps information about response time, processing time, packet sizes, etc of IoT applications). Moreover, the databases are containerized for faster deployments. Fig 3 depicts an overview of databases and their tables.

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.

4 PERFORMANCE EVALUATION

In this section, we discuss 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

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 next state of the grid, a local function must be applied to each cell simultaneously [19]. 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 32 dependent *task executor* containers (one for each rectangle) with different computation sizes.

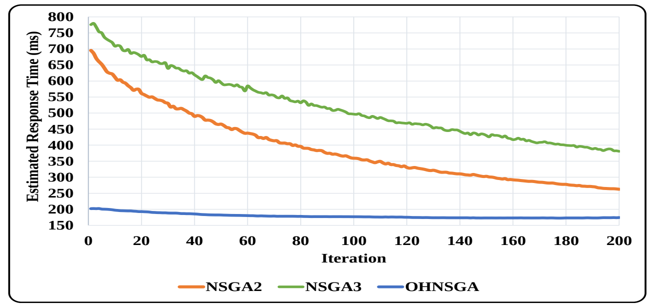

Figure 4: Scheduling performance in different iterations

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 key-frame, 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* components increases.

In the third experiment, we analyze and compare the resource usage of our framework in terms of its startup time and RAM usage with its counterparts.

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 [20], and Non-dominated Sorting Genetic Algorithm 3 (*NSGA3*) [18]. 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 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 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. 5 depicts the real response time of Conway's Game of Life application, obtained from the execution of tasks in the real 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* converges faster to better solution compared to other policies, but its estimated solutions can better represent the behavior of the Game of Life in real environments.

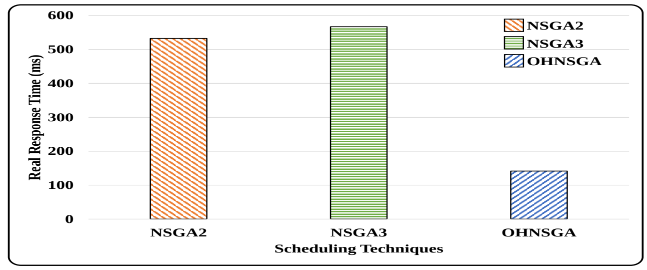

Figure 5: Real response time of scheduling policies

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 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 the 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. 6 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 simultaneous requests is small. However, as the number of requests increases, the SFT values of the no-scalability scenario dramatically increase compared to the scalability scenario. It shows the importance of supporting scalability mechanisms and policies in FogBus2. The *master* containers are scaled up as the number of requests increases, which significantly reduces the queuing time of requests.

4.5 Startup Time and RAM Usage Analysis

This experiment studies the startup time and RAM usage of our framework, FogBus2, and compares it with FogBus framework [8]. Fig. 7 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

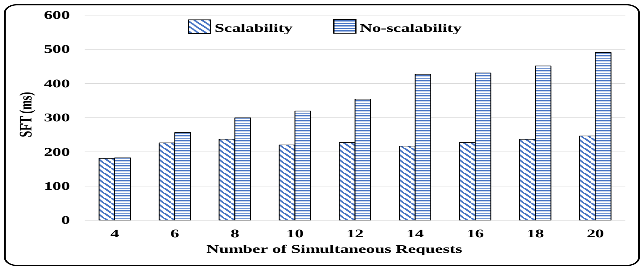

Figure 6: Analysis of master components' scalability

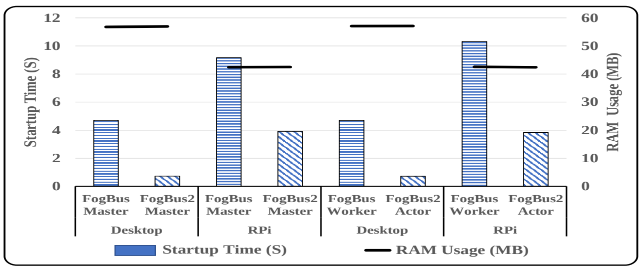

Figure 7: Startup time and RAM usage analysis

time of FogBus2 is roughly 80% and 60% faster in comparison to FogBus on Desktop and RPi, which makes it a suitable option for fast deployment of any type of IoT-enabled systems.

5 CONCLUSIONS AND FUTURE WORK

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. FogBus2 offers fast and low-overhead deployments of applications using containerization. Also, it offers scheduling, scalability, resource discovery, and dynamic profiling mechanisms, assisting IoT developers to define and deploy their targeted IoT applications on FogBus2. Moreover, it integrates several scheduling, scalability, and resource discovery policies. Besides, 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.

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 extended by (1) integrating dynamic clustering mechanisms and policies to cluster resources either horizontally or vertically, (2) integrating container-orchestration techniques to automate the management of application deployments and scaling, (3) mobility-support in different layers of edge/fog computing environment, i.e., mobility support for IoT users and edge/fog servers, (4) privacy-preservation-support for the users' private information and edge/fog servers, (5) integrating machine learning techniques to analyze the current state of edge/fog computing environment, (6) integrating lightweight security mechanisms to ensure data confidentiality and integrity.

REFERENCES

[1] P. Hu, S. Dhelim, H. Ning, and T. Qiu, "Survey on fog computing: architecture, key technologies, applications and open issues," *Journal of Network and Computer Applications*, vol. 98, pp. 27–42, 2017.

[2] S. Deng, Z. Xiang, J. Taheri, K. A. Mohammad, J. Yin, A. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed edges," *IEEE Transactions on Mobile Computing*, 2020.

[3] M. Goudarzi, Z. Movahedi, and M. Nazari, "Mobile cloud computing: a multisite computation offloading," in *2016 8th International Symposium on Telecommunications (IST)*. IEEE, 2016, pp. 660–665.

[4] X. Xu, Q. Liu, Y. Luo, K. Peng, X. Zhang, S. Meng, and L. Qi, "A computation offloading method over big data for iot-enabled cloud-edge computing," *Future Generation Computer Systems*, vol. 95, pp. 522–533, 2019.

[5] M. Goudarzi, H. Wu, M. S. Palaniswami, and R. Buyya, "An application placement technique for concurrent iot applications in edge and fog computing environments," *IEEE Transactions on Mobile Computing*, vol. 20, no. 4, pp. 1298 – 1311, 2021.

[6] M. Goudarzi, M. Palaniswami, and R. Buyya, "A fog-driven dynamic resource allocation technique in ultra dense femtocell networks," *Journal of Network and Computer Applications*, vol. 145, p. 102407, 2019.

[7] G. Merlino, R. Dautov, S. Distefano, and D. Bruneo, "Enabling workload engineering in edge, fog, and cloud computing through openstack-based middleware," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, pp. 1–22, 2019.

[8] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "Fogbus: A blockchain-based lightweight framework for edge and fog computing," *Journal of Systems and Software*, vol. 154, pp. 22–36, 2019.

[9] J. An, W. Li, F. Le Gall, E. Kovac, J. Kim, T. Taleb, and J. Song, "Eif: Toward an elastic iot fog framework for ai services," *IEEE Communications Magazine*, vol. 57, no. 5, pp. 28–33, 2019.

[10] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H. C. Cankaya, Q. Zhang, W. Xie, and J. P. Jue, "Fogplan: a lightweight qos-aware dynamic fog service provisioning framework," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 5080–5096, 2019.

[11] D. T. Nguyen, L. B. Le, and V. K. Bhargava, "A market-based framework for multi-resource allocation in fog computing," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1151–1164, 2019.

[12] S. Ghosh, A. Mukherjee, S. K. Ghosh, and R. Buyya, "Mobi-iost: mobility-aware cloud-fog-edge-iot collaborative framework for time-critical applications," *IEEE Transactions on Network Science and Engineering*, 2019.

[13] D. Borthakur, H. Dubey, N. Constant, L. Mahler, and K. 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)*. IEEE, 2017, pp. 472–476.

[14] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, "Foggy: A framework for continuous automated iot application deployment in fog computing," in *2017 IEEE International Conference on AI & Mobile Services*. IEEE, 2017, pp. 38–45.

[15] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over raspberrypi," in *Proceedings of the 18th international conference on distributed computing and networking*, 2017, pp. 1–10.

[16] A. J. Ferrer, J. M. Marques, and J. Jorba, "Ad-hoc edge cloud: A framework for dynamic creation of edge computing infrastructures," in *2019 28th International Conference on Computer Communication and Networks*. IEEE, 2019, pp. 1–7.

[17] S. Noor, B. Koehler, A. Steenson, J. Caballero, D. Ellenberger, and L. Heilman, "Iotdoc: A docker-container based architecture of iot-enabled cloud system," in *3rd IEEE/ACIS International Conference on Big Data, Cloud Computing, and Data Science Engineering*. Springer, 2019, pp. 51–68.

[18] K. Deb and H. 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*, vol. 18, no. 4, pp. 577–601, 2013.

[19] P. Rendell, "Turing universality of the game of life," in *Collision-based computing*. Springer, 2002, pp. 513–539.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.