An Integrated and Standards-based Fog Computing Federation Framework

A Dissertation Submitted to the University of Hyderabad in Partial Fulfillment of the Degree of

Master of Technology

in

Information Technology

by

Suvam Basak

20MCMB08

School of Computer and Information Science

University of Hyderabad

Gachibowli, Hyderabad - 500 046

Telangana, India

June 11, 2022

CERTIFICATE

This is to certify that the dissertation titled, **"An Integrated and Standards-based Fog Computing Federation Framework"** submitted by **Suvam Basak**, bearing Registration number: **20MCMB08**, in partial fulfillment of the requirements for the award of Master of Technology in Information Technology is a bonafide work carried out by him under my supervision and guidance.

The dissertation has not been submitted previously in part or in full to this or any other University or Institution for the award of any degree or diploma.

Dr. Satish Narayana Srirama	**Prof. Chakravarthy Bhagvati**
(Project Supervisor)	(Dean)
School of Computer and Information Sciences,	School of Computer and Information Sciences,
University of Hyderabad	**University of Hyderabad**

DECLARATION

I, Suvam Basak, hereby declare that this dissertation titled, **"An Integrated and Standards-based Fog Computing Federation Framework"**, submitted by **me** under the guidance and supervision of **Dr. Satish Narayana Srirama** is a bonafide work which is also free from plagiarism. I also declare that it has not been submitted previously in part or in full to this University or other University or Institution for the award of any degree or diploma. I hereby agree that my dissertation can be deposited in Shodganga/INFLIBNET.

A report on plagiarism statistics from the University Librarian is enclosed.

Date:

Name: Suvam Basak

Registration No.: 20MCMB08

//Countersigned//

Signature of the Supervisor(s):

(Dr. Satish Narayana Srirama)

Acknowledgements

It gives me great pleasure to thank all the people who have helped me make this project a real learning experience. First and foremost, I would like to thank and express my gratitude to **Dr. Satish Narayana Srirama** for his valuable guidance, providing a research-friendly atmosphere, and support while dealing with some setbacks. Without his excellent direction, this project would not have been possible. It was a real privilege to work with him and grow in leaps and bounds. I would also like to thank the **School of Computer and Information Sciences**, the **University of Hyderabad**, for providing excellent lab facilities.

I also acknowledge **Science and Engineering Research Board**, India, for financial support, through grant **CRG/2021/003888**. I also thank financial support to **University of Hyderabad - Institute of Eminence** by **Ministry of Education, (formerly the Ministry of Human Resource Development), India (F11/9/2019-U3(A))**.

Suvam Basak

Abstract

The conventional cloud-centric IoT application does not meet the requirement of the quick reaction of time-critical applications. The idea of edge and fog computing arrived to overcome the limitation of cloud computing in the domain of IoT applications. In fog computing, the workloads get distributed across the fog devices, and the placement of services in the nearest fog devices drastically reduces the network delay, connectivity, and reliability issues and delivers real-time capabilities as an extension, it reduces energy consumption and network overhead in the case of large sensor networks. However, achieving seamless interoperability, platform independence, and automatic deployment of services becomes the major challenge over heterogeneous fog devices. This work proposes an adaptation of the OASIS - Topology and Orchestration Specification for Cloud Applications (TOSCA) for service deployment for fog computing. With TOSCA, we build an integrated and standards-based fog computing federation framework that abstracts all the heterogeneity and complexities and offers a user-friendly paradigm to model and dynamically deploy fog services, on-demand, on the fly, from a remote system. The framework uses (i) standard TOSCA Service Template for application modeling. (ii) Docker Containers (OS-level virtualization) for platform independence, and (iii) Docker Swarm (container orchestration tool) manages seamless coordination and cooperation across heterogeneous fog devices. Moreover, the fog application is dynamically deployed by a lightweight TOSCA-compliant orchestrator on a set of out-of-the-box fog devices. The case study on the framework indicates convincing resource consumption by the orchestration process on these resource-constrained fog devices. The framework realizes various fog applications through the dynamic deployment of custom or user-developed services on the fly.

Contents

List of Figures

List of Tables

List of acronyms

IoT	Internet of Things
TOSCA	Topology and Orchestration Specification for Cloud Applications
IIoT	Industrial Internet of Things
IC	Integrated Circuit
I/O	Inputs/Output
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
USB	Universal Serial Bus
ALU	Arithmetic Logical Unit
CU	Control Unit
PC	Program Counte
SP	Stack Pointer
ISA	Instruction Set Architecture
PWM	Pulse-Width Modulation
NFC	Near-Field Communication
GPIO	General Purpose Input Output
RAM	Random Access Memory
SSH	Secure Shell
ISP	Internet Service Provider
IP	Internet Protocol
NAT	Network Address Translation
DNS	Domain Name System
M2M	Machine to Machine

Table 1: List of acronyms

Chapter 1

Introduction

The beginning of the era of Internet of Things (IoT) applications was cloud-centric. All the diverse data collected through the sensors from the different fields are stored in the cloud. The cloud has virtually infinite computational power for processing that sensor data, generating an actuation signal to send back to the actuators on that site. This architecture has some drawbacks. First, the cloud is physically placed thousands of kilometers away from the sensors and actuators in most cases, which adds a significant network delay between sensing and actuation. That is undesirable for some real-time applications. Second, this architecture needs stable connectivity to the cloud for continuously streaming the data. However, the IoT devices may have to handle unstable connectivity in the real world, leading to the reliability issues of the IoT applications. Third, with the exponential increase of IoT devices data generation rate has also increased rapidly. It is becoming an overhead for the network to move vast volumes of data to the cloud. Therefore, this cloud-centric architecture fails to meet the several requirements of real-time applications criteria of low latency and high reliability [7]. Therefore, the concept of Edge and Fog computing comes into the picture to address these problems. The solution to this issue is to move this processing component from the cloud platform to the local hybrid distributed processing architecture known as Fog computing [8, 9]. As an extension, this solution reduces the overall network load as well. This fog federation allows seamless coordination and cooperation between heterogeneous devices (gateway devices such as Raspberry Pi, Drones, Network Switches, and Routers). In Fog computing, the utilization of gateway devices acts as an alternative or supportive computational equipment of the cloud [36]. The biggest challenge in this architecture is having reliability, portability, interoperability, and platform independence on top of a set of heterogeneous devices [6, 10].

The cloud community also faced a similar problem with a lack of standardization and vendor lock-in or portability of composite cloud applications across the different cloud service providers [27]. OASIS - Topology and Orchestration Specification for Cloud Applications (TOSCA) addressed the problem [5]. TOSCA automates the entire life cycle of a cloud application with operations like 'create', 'configure', 'start', 'stop', and 'delete' [4]. In a nutshell, TOSCA is a modeling language used to create the blueprint of composite cloud applications. Developers can model their applications in a YAML[1] file, known as TOSCA Service Template, and deploy them on a cloud platform through a TOSCA- compliant orchestrator. To implement the TOSCA standard, a TOSCA-compliant orchestrator uses implementation scripts (Shell script, Python script[2], or any IT Automation tool like Ansible[3], Chef[4], Puppet[5]).

This thesis proposes an extension of the TOSCA standard for fog computing framework, **FogDEFT**[6] (**F**og computing out of the box: **D**ynamic d**E**ployment of **F**og service containers with **T**OSCA), to provide user-friendly development and deployment paradigms for fog applications. Therefore, with the standard TOSCA Service Template, an application developer will describe the fog service's conceptual structure (or blueprint). A lightweight orchestrator tool will take TOSCA Service Templates and dynamically deploys these services on a set of heterogeneous fog devices. This deployment means starting one or more Docker Containers in

[1]https://yaml.org
[2]https://www.python.org
[3]https://www.ansible.com
[4]https://www.chef.io
[5]https://puppet.com
[6]https://github.com/cloud-and-smart-labs/fog-service-orchestration

swarm mode or could be in standalone mode. These containers can talk to each other like microservice architecture and exchange sensor data and actuation signals seamlessly, irrespective of their hardware architecture and placement inside a fog federation.

1.1 Motivation

The number of connected devices on the internet is a few billion. This number increases rapidly with the Industrial Internet of Things (IIoT) or Industry 4.0. Interestingly, each device connected to the internet comes with some computational power. The core idea of Fog computing is to accumulate and utilize these devices' computational power at the network's edge. However, the adoption rate of Fog computing is not in proportion with the performance it proposes because resource constraints, heterogeneity, and lack of standardization, require application-specific proprietary solutions.

Adopting Fog computing and deploying the services of an IoT application on fog nodes opens a nontrivial area of research. Several automation tools have been developed with the rise of Infrastructure as Code (IaC) to manage, configure, and provision servers and data centers. None of them primarily targets the deployment of fog services. Unlike conventional computing systems, fog devices are heterogeneous and have a completely different hardware architecture that may run on different software and probably optimized to perform specific tasks. However, the main requirement of a fog federation is to have seamless cooperation and interoperability across all the fog devices inside the network. Therefore, the dynamic deployment of a service on fog devices imposes a challenge of handling platform independence, interoperability, and portability with resource constraints. Therefore, some technology is supposed to be developed to unlock this computational power, and any fog application can deploy services in a distributed fashion. With this motivation in this thesis, we propose a standardized framework for fog service deployment called 'FogDEFT'.

1.2 Contributions

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

1. Creating node and relationship types for describing fog services in TOSCA.
2. Relevant actuation scripts for the created node and relationship types.
3. Dynamic deployment of services on top of fog nodes on the fly with a lightweight orchestrator.
4. Demonstrate deployment of fog application across multiple networks.
5. System design and modeling (creating TOSCA Service Template) of a domain-specific fog application followed by deployment (a case study of the framework).

1.3 Summary

The conventional cloud-centric IoT model has the limitation of latency and creates network overhead. In Fog computing, on-premise placement on this processing component eliminates the issue of latency, connectivity, reliability, and network overhead. The challenges in this Fog computing are (i) Automatic and dynamic deployment of the services, (ii) Platform independence, and (iii) Interoperability across heterogeneous devices. The cloud communities also faced a similar problem due to a lack of standardization. OASIS TOSCA addressed the issue with standard application modeling technology. The hypothesis of the project is: (i) Adopt the standard TOSCA Service Template for modeling fog application, (ii) Use containerization for platform independence, and (iii) Container orchestration for interoperability. Further, automatically deploy the services on the fly with a lightweight orchestrator using standard TOSCA Service Templates.

1.4 Outline

This chapter introduced the scope of the project and illustrated the problem description. The rest of the thesis is organized like this. Chapter 2 discusses all sorts of related work in this domain up to this date. Chapter 3 briefly describes the fog and other hardware devices used in experiments and case studies. Chapter 4 discusses the FogDEFT framework internal details. Chapter 5 illustrates the on-demand deployment of service on heterogeneous fog devices. Chapter 6 includes two case studies on the framework. Finally, Chapter 7 concludes with future potential applications of TOSCA in IoT.

Chapter 2

Related Work

Fog computing resolves many problems of cloud-centric IoT applications. However, fog computing comes with challenges that significantly slow down the adoption rate of fog computing. On-demand service deployment on the fog node is one of these challenges discussed in Chapter 1. Therefore, application/service deployment for fog computing has been studied extensively. This chapter discusses all sorts of related application deployment and evaluation works already being done in different domains.

2.1 Performance evaluation of container orchestration on fog devices

In [3], P Bellavista et al. conducted a thorough feasibility study on fog computing deployment on Raspberry Pi with performance overhead of containerization technologies (LXC, Docker, CRIU), container orchestration tools (Docker Swarm, Kubernetes, Apache Mesos), filesystems (AUFS, OverlayFS). They gave an impressive comparison between natively running code and container execution.

P Kayal in [23] reviewed container orchestration service Kubernetes and its architecture and implementation from the point of view of IoT-driven application and fog computing. Furthermore, point outs that (i) required functionality is achievable without significant penalty and (ii) some shortcomings of Kubernetes in distributed deployment on fog infrastructure.

2.2 Dynamic deployment of applications

In [20], N Ferry et al. carried out a case study of GeneSIS in smart buildings. They showed that GeneSIS is secure by design from the development and deployment to the operation of IoT-based systems and keeps adapting to the evolving security threats to maintain their trustworthiness.

In [21], HA Hassan and RP Qasha propose a new idea to build a deployment model for the IoT based distributed systems on a user-friendly and straightforward description of the intelligent devices' installation, configuration, and their computation and communication with the IoT based system parts with Ansible-based YAML description. The work minimizes the efforts to deploy the IoT-driven distributed applications on various infrastructures consisting of fog and cloud.

In [34], O Tomarchio et al. proposed a TOSCA-based framework, 'TORCH,' for deploying and orchestrating classical and containerized cloud applications on multiple cloud providers. The main benefit of the framework is the option or adaptability to add support to the cloud service provider platform at minimal modification effort, and it provides web based tool to manage all deployments.

In [14], R Dautov et al. proposed software updates provisioning architecture (hierarchical) that pushes the update from the cloud to terminal edge devices through the edge gateways. These edge gateways run some agents as microservice containers connecting edge devices to the centralized cloud platform and installing the firmware update at the edge devices in a targeted manner.

In [32], H Song et al. describe joint research on an industrial use case Smart Healthcare application provider that uses a model-based approach to provide a fleet of edge devices or gateways for automatic deployment. That fleet selection process uses a set of constraints (hard and soft constraints) to pick the correct and balanced distribution of software.

2.3 Deployment of fog applications

B Donassolo et al. in [18] proposed another orchestration framework called 'FI-TOR,' an automated deployment and microservice migration solution for IoT applications. The framework uses ”Optimized Fog Service Provisioning” (O-FSP) based on a greedy approach that outperforms other relevant strategies in terms of (i) CPU usage, (ii) acceptance rate, and (iii) provisioning cost.

In [19], N Ferry et al. developed a framework for continuous deployment across a set of heterogeneous IoT, edge devices, and cloud platforms for decentralized computing called ”Generation and Deployment of Smart IoT Systems (GeneSIS).” GeneSIS provides (i) an execution engine to support automatic deployment across a set of IoT, edge devices, and cloud resources and (ii) a domain-specific modeling language for modeling and the deployment and orchestration of Smart IoT-driven systems.

In [37], S Venticinque and A Amato proposed a new fog service placement methodology. The methodology’s effectiveness is demonstrated in the energy domain with smart grid.

In [15], G Davoli et al. developed a modular orchestration system called ‘FORCH.’ The orchestrator is aware of different service models (SaaS/PaaS/IaaS) and dynamically deploys services and manages resources on the fog nodes.

In [29], H Sami and A Mourad proposed a new framework for deploying fog service on-demand on the fly with the existence of volunteering nodes (devices). The framework is based on Kubeadm and Docker. Moreover, the framework optimizes the container placement problem with an ”Evolutionary Memetic Algorithm” (MA) that uses heuristics to make decisions.

In [30], H Sami et al. proposed a context-aware and resource-efficient approach for deploying microservices container on-demand named ’Vehicular-OBUs-As-On-Demand-Fogs.’ The scheme embeds flexible networking architecture combining cellular technologies and ad-hoc wireless network (802.11p) used in vehicles and a Kubeadm-based clustering approach with docker container-based microservices deployment. It provides an on-demand service placement technology for fog and vehicles based on an ’Evolutionary Memetic Algorithm’.

2.4 Dynamic deployment of fog applications

In [22], S Hoque et al. have carried out a technical evaluation of docker container and container orchestration tools, their capability, limitations, and how containerization can impact application performance. The result shows that significant adjustments are required to meet the fog environment needs, and they have proposed a framework based on the docker swarm to address issues with the help of ‘OpenloTFog’ toolkits.

In 2013 F Li et al. put the first effort to use TOSCA, the new cloud standard, for IoT applications and demonstrated the feasibility of modeling IoT components gateways and drivers for building Air Handling Unit (AHU) with the first edition of TOSCA [24].

ACF da Silva et al. in [12] automatically deployed an IoT application with OpenTOSCA based on Mosquitto Message Broker running on the cloud, and the publishers and subscribers were running in two different raspberry pis. Later, in [13], they automatically deployed an IoT application out of the box where a python script on raspberry pi pushes the data to the message broker on a cloud, and another virtual machine is hosting a web-based dashboard to present the sensor data. They validated this deployment with three case studies of emerging middleware (i) OpenMTC, (ii) Eclipse Mosquitto, and (iii) FIWARE Orion Context Broker.

In [35], A Tsagkaropoulos et al. presented TOSCA extensions for modeling applications relying on any combination of technologies and discussed semantic enhancements, optimization aspects, and methodology that should be followed for edge and fog deployment support. Furthermore, added a comparison with other cloud application deployment approaches.

In [31], HE Solayman and RP Qasha, used TOSCA for deploying Docker containers of IoT applications for Intensive Care Unit (ICU). The demonstration shows automation of IoT application provisioning in heterogeneous environments consisting of hardware components and cloud instance message broker for network communication between components containerized with docker containers.

In this work we propose the idea of “fog out of the box” where the applications services are dynamically deployed on the fly without any cloud involvement (except the cloning of the code or container images from registry/repositories), as J Delsing et al. argued in [17] about open internet automation limitations and discussed the idea of local cloud in IoT automation. Since such automation of IoT is geographically and physically local, hence local cloud meets requirements of (1) interoperability issue of a wide range of IoT and some legacy devices, (2) real-time capabilities as latency guarantee required for automation system, (3) scalability of enormous scale, (4) security fence from the external network of automation system and (5) ease of application engineering with integrity, and agility. Therefore, in this work the fog federation is dynamically created with a standardize TOSCA Service Template and container orchestration tool, Docker Swarm. The orchestration process will either run on the fog node itself or any PC/laptop/workstation that remotely orchestrates services on the fog nodes.

In recent years, significant work has been published regarding smart IoT-based solutions for greenhouse production and farming [28, 33, 2, 26]. And some work in Cold Storage as well [1]. The proposed framework FogDEFT can realize such use cases to achieve the highest level of convenience and can change the climate of greenhouse or cold storage for specific crops and products, respectively. The case study of such application with this framework is demonstrated in Chapter 6.

2.5 Summary

All discussed related work and their focus in this chapter are summarized in Table 2.1. In contrast, our proposed fog federation framework uses the TOSCA standard for modeling the fog application and gives a user-friendly way to design and dynamically deploy fog service across fog nodes on demand through a single command.

Work	Architecture Framework	Performance	Virtualization	Agent	Deployment/Placement /Cost Algorithm	Service Placement	TOSCA
P Bellavista et al. [3]	✓	✓	✓				
P Kayal [23]			✓				
N Ferry et al. [20]	✓		✓	✓			
HA Hassan and RP Qasha [21]	✓	✓					
O Tomarchio et al. [34]	✓	✓	✓				✓
R Dautov et al. [14]			✓	✓			
H Song et al. [32]	✓	✓	✓	✓			
B Donassolo et al. [18]	✓	✓			✓	✓	
N Ferry et al. [19]	✓		✓	✓			
S Venticinque and A Amato [37]						✓	
G Davoli et al. [15]	✓						
H Sami and A Mourad. [29]			✓		✓	✓	
H Sami et al. [30]	✓		✓		✓	✓	
S Hoque et al. [22]	✓	✓	✓				
F Li et al.[24]							✓
ACF da Silva et al. [12]							✓
ACF da Silva et al. [13]							✓
A Tsagkaropoulos et al. [35]							✓
HE Solayman and RP Qasha [31]			✓				✓
FogDEFT	✓	✓	✓				✓

Table 2.1: A summary of related work and their focus

Chapter 3

Fog devices and hardware overview

This Chapter briefly discusses IoT hardware. The experiments and case studies used two different IoT hardware devices based on microcontrollers and microprocessors. Only the microprocessors are considered fog nodes, where fog services get deployed. The microcontroller primarily provides an interface to the analog sensors and actuators, whereas digital sensors can directly interface with the microprocessor-based devices. Therefore, two different types of sensors and actuators are analog and digital. The following sections give a brief overview of microcontroller and microprocessor-based systems (used in our experiments and case studies), their capability and limitations, and digital and analog sensors and actuators. Later, Chapter 6, with case studies, shows the generic system design for deploying different fog services with these hardware devices.

3.1 Microcontrollers based devices

A microcontroller is an integrated circuit (IC) designed to perform an embedded system's specific operation (single task). The main elements of the microprocessor are the processor (could be 4-bit, 8-bit, 16-bit, 32-bit, 64-bit processor) that performs basic arithmetic, logic, and I/O operations. Program memory non-volatile memory stores programs or instructions to be executed. Data memory cloud be

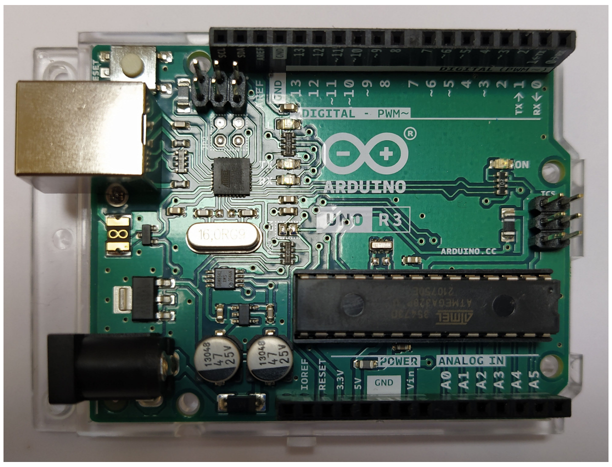


Figure 3.1: Arduino Uno

volatile memory for temporary data storage. Other supporting elements (highly relevant to this IoT context) Analog to Digital Converter (ADC) that converts analog signals to digital signals, Digital to Analog Converter (DAC) that converts digital signals to analog signals, Serial port is an I/O port to connect other devices (e.g., USB). Therefore, microcontrollers are used for various purposes, including automation, manufacturing, robotics, automotive, and IoT applications.

3.1.1 Arduino Uno

Arduino Uno[1], shown in Figure 3.1, is (open-source hardware) a microcontroller board based on the **ATmega328P** microcontroller. When a program is loaded, it executes the same program if connected to power. It has fourteen digital I/O pins (six-pin PWM output capable) and six analog input pins with 10-bit ADC. Therefore, Arduino can interact with both analog and digital sensors and actuators. The board has a serial port to interface with other devices and exchange data across the device.

3.1.2 Arduino Nano 33 BLE Sense

Arduino Nano 33 BLE Sense[2], shown in Figure 3.2, similar to Arduino Uno, is also the smallest form factor microcontroller board. This board is powered by a 32-

[1]https://docs.arduino.cc/hardware/uno-rev3
[2]https://docs.arduino.cc/hardware/nano-33-ble-sense

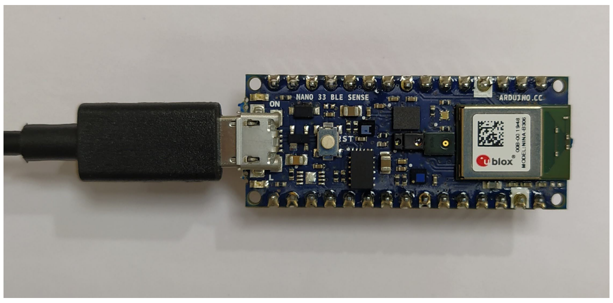


Figure 3.2: Arduino Nano 33 BLE Sense

bit ARM based Cortex-M4 CPU and 1MB of program memory and has onboard Bluetooth pairing via NFC and ultra-low power consumption modes, a micro-USB connector, and a bunch of onboard sensors:

- Nine axes inertial sensor
- Humidity and temperature sensor
- Barometric sensor
- Microphone
- Gesture, proximity, light color, and light intensity sensor

3.2 Microprocessors based devices

A microprocessor is an integrated circuit (IC) in the computer we use daily. A microprocessor consists of an Arithmetic Logical Unit (ALU) that performs arithmetic and logical operation on data, a Control Unit (CU) control the flow of data/instructions inside the system memory and bus, and an array of registers with some special register like Program Counter (PC), Stack Pointer (SP), etc. A microprocessor implements an Instruction Set Architecture (IAS) and executes these instructions to perform a particular task. The regular personal computer and smartphones we use consist of a microprocessor of some architecture (x86 or ARM).

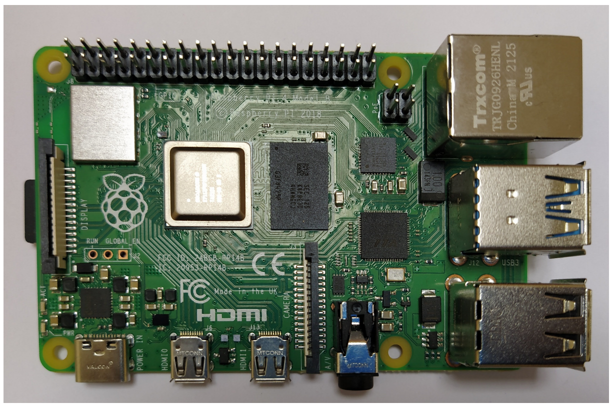


Figure 3.3: Raspberry Pi 4 Model B

3.2.1 Raspberry Pi

Raspberry Pi[3] is a single-board credit-card-sized computer that runs a Debian-based Linux distribution called Raspberry Pi OS (can run other operating systems as well, e.g. Ubuntu, Ubuntu core, Windows 10 IoT Core). It has Serial ports (USB), a display port (more than once in the current version), and 40 GPIO pins to interface sensors and actuators. From an IoT point of view, Raspberry Pi is a gateway device. As the Raspberry Pi is a computer with an ARM-based microprocessor. It can not interface with analog sensors and actuators. Raspberry Pi works with digital sensors and actuators only.

The Raspberry Pi, 4 Model B[4], is shown in Figure 3.3. It has Gigabit Ethernet, onboard wireless networking, and Bluetooth and is powered by Broadcom chip (BCM2711), ARM v8 based Quad-core Cortex-A72 64-bit 1.5GHz processor with 4GB RAM. Our experiments and case studies include an older version of Raspberry Pi boards like Raspberry Pi 3 Model B[5].

3.2.2 Personal Computer

Personal computers and workstations (Intel x86 based) are available on-premises and can be used in fog infrastructure (just like a local cloud). In some experiments

[3]https://www.raspberrypi.org/
[4]https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
[5]https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

Table 3.1: IoT device list

Device Name	**Processor Architecture**	**Memory Size**	**Operating System**
Arduino Uno	Microcontroller ATmega328P	32 KB	-
Arduino Nano 33 BLE Sense	ARM Cortex-M4	1 MB	-
Raspberry Pi 4	ARMv7l 32-Bit	4 GB	Raspberry Pi OS 10
Raspberry Pi 4	ARMv8 64-Bit	4 GB	Raspberry Pi OS 11
PC	Intel x86-64	4-8 GB	Ubuntu 20.04 Debian 11
Virtual Machines	Intel x86-64	2-4 GB	Ubuntu 20.04 Debian 11
Workstation	Intel Xeon W-2145 3.70GHz $\times$ 16	32 GB	Ubuntu 20.04

and case studies, personal computers, workstations, and virtual machines are also included in the fog federation.

3.3 Other hardware

Apart from microcontrollers and microprocessors handful of sensors and actuators are also used for experiments, testing, and case studies of the framework. The sensors include **DHT11**[6], digital temperature, and humidity sensors. The actuators include **LED** and **Servomotor**[7].

3.4 Summary

All the devices used in the experiment and the FogDEFT framework case studies are summarized and listed in Table 3.1, and sensors and actuators are listed in Table 3.2.

[6]https://www.adafruit.com/product/386
[7]https://www.towerpro.com.tw/product/sg90-7/

Table 3.2: Sensors and Actuators

Sensors	**Actuators**
DHT11	LED
	Servomotor

Chapter 4

The framework FogDEFT

The **FogDEFT** (**Fog** computing out of the box: **D**ynamic d**E**ployment of **F**og service containers with **T**OSCA) framework is a fog federation framework built on the extension of the TOSCA standard, which is the de-facto standard for modeling cloud applications. The framework enables seamless cooperation and coordination between fog devices in the network hides the heterogeneity, offers a user-friendly development paradigm for the custom or user-developed fog application, and realizes dynamic deployment of the fog services on the fly on demand. The FogDEFT framework maintains three layers of abstraction. In this Chapter, in subsequent sections, we discuss the problem and challenges followed by how this three-layer of abstraction ensures platform independence, interoperability, and portability of fog services across heterogeneous fog devices.

4.1 Problem description

The development of a framework for deploying standardized user-developed services out of the box throws a few challenges. These challenges can be summarized into two categories:

4.1.1 Platform independence and interoperability over heterogeneous hardware

The idea of a fog federation is to include all devices available locally on-premises for computational purposes. Then the local processing task is handled with the accumulated computational capability of these devices. The locally available devices primarily include network and gateway devices and some on-premises core computational devices (e.g., from private clouds). These devices are built by different manufacturers, based on specific hardware architecture, optimized for particular tasks, and could run on individual software. For example, Table 3.1 from the previous Chapter shows the collection of heterogeneous devices we used for our experiments.

These are all diverse types of hardware commonly we come across in typical scenarios. Each of them has a different type of features, capabilities, and performance. The Raspberry Pi has 40 GPIO pins to interact with sensors and actuators. However, personal computers do not have that capability. Personal computers have much more processing power than Raspberry Pi. Even comparing Raspberry Pi 3 and Raspberry Pi 4, there are also significant differences in the processing power. Therefore, a significant heterogeneity over hardware and software can be envisioned as relevant to the fog infrastructure. So, while orchestrating the services on such fog devices, we must be able to place these services across the nodes (heterogeneous devices) based on their capabilities and requirements of services. Therefore these services must serve their purpose independent of their platform. Furthermore, these services should be able to talk to each other like a conventional microservice architecture irrespective of the placement of the services over the fog devices, which will ensure interoperability across the fog federation.

4.1.2 Modeling of user-developed services with TOSCA

TOSCA's development acknowledges the issue of standardization and portability of cloud applications. TOSCA and relevant basics are discussed in Section 4.4. EU

H2020 RADON Project[1] extended TOSCA and provides reusable TOSCA types of application runtimes, computing resources, and Function as a Service (FaaS) platforms in the form of abstract and deployable modeling entities [11]. Another extension of TOSCA called TOSCAData focuses on modeling data pipeline-based cloud applications [16]. Because TOSCA is platform agnostic, the language provides a mechanism to extend the definitions with additional domain-specific or vendor-specific information. We can extend/adapt the TOSCA to model user-developed fog applications.

Therefore, supporting fog service with TOSCA requires the creation of appropriate Nodes and Relationship types. These Nodes and Relationship types will represent the components of the fog services. Then a developer can write a TOSCA Service Template with Node and Relationship types for user-developed fog applications. A lightweight TOSCA-complaint orchestrator (discussed in Chapter 5) can take the TOSCA Service Template and deploy the services on out-of-the-box fog nodes.

4.2 Platform independence

All fog devices are mostly network equipment and gateway devices (Routers, Network Switches, Drones) or conventional computational devices (e.g., Local cloud). These devices consist of different hardware architectures and operating systems. Therefore, the **first layer of the abstraction of the fog federation framework is to handle platform independence through virtualization.** However, hardware virtualization is costly and resource-intensive. Therefore, it is not a feasible solution for resource-constrained fog devices. However, all these fog devices are network devices, and all network devices run on some form of Linux system. Therefore, these fog devices are running Linux kernels, making containerization or OS-level virtualization a feasible solution since containers take kernel support from the host machine and run in isolation without interfering with the

[1] https://radon-h2020.eu

host system or other containers. The **FogDEFT framework** uses **Docker containers**[2] to deploy the services on the fog nodes.

4.2.1 Addressing heterogeneity with Docker

Before diving into fog devices, let us understand the problem containerization technology is trying to solve in the industry.

- In the IT industry, before developing any application, the developers need to carry out a compatibility check of all the components and services of the application with underlying hardware, operating systems, libraries, and dependencies.
- If any component and service need to be modified or updated, then the developers have to go through the same process of the compatibility check; otherwise, that change may break the system.
- Any application running in an environment does not guarantee that the application will run in different environments. E.g. *"The application works on the developer's machine but not in production."*

The containerization of the applications or components of the application removes such problems [25]. These containers are minimally packed with specific software or source code with dependencies and libraries. These containers run in isolation on the Docker Engine in a system shown in Figure 4.1. Containers have processes, services, network interfaces, and mounts almost like virtual machines (feels like), except they use the same host machine Operating System's Kernel. So now, each application component is possible to change or modify independently without affecting the host Operating System or any other application components. These containers are lightweight and significantly small (mostly a few MegaBytes). They consume minimal resources while running, so starting a container in a few seconds is possible. On the other hand, virtual machines take a few GigaBytes of storage (run a full-fledged Operating System), consume a massive amount of resources,

[2] https://www.docker.com

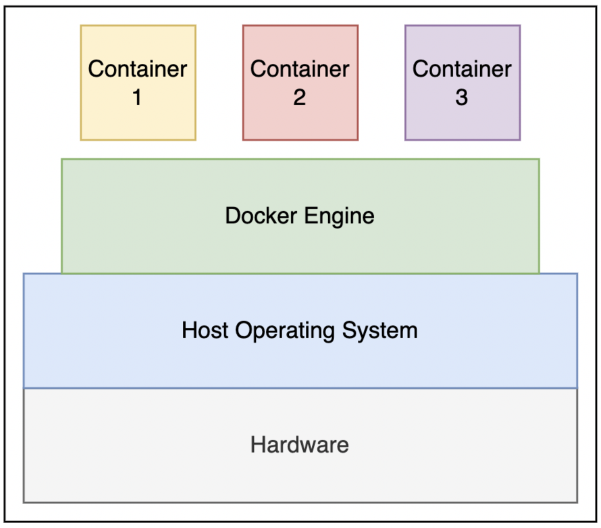


Figure 4.1: System architecture of a Docker host

and take a few minutes to start.

This containerization technology uses kernel features **control groups** or **cgroups** (limits and accounts for the system resources) and **namespace** (logical isolation from the host system with scope) to create logical isolation inside a host system. These containers are based on decade-old technology such as LXC, LXD, and LX-CFS. Docker uses LXC[3]. However, setting up such containers is difficult as they are at a deficient level. Here Docker comes into the picture and provides a user-friendly high-level tool with many functionalities to make setting up containers easier for the end-users like us.

We have to recollect some basic Operating System concepts to understand how containerization works on these systems. Broadly an Operating System has two components. At the bottom, the Kernel interacts with hardware, and a set of software (file manager, word processor, compiler, user interface) on top takes the services from the Kernel via System Calls. The underlying Kernel is the same if we look at different Linux distributions. The set of software on top of that makes the different Operating Systems (known as Linux distributions). We mentioned earlier that the Docker container shares the underlying kernel. So if we manage to

[3]https://linuxcontainers.org

have Docker Engine on a Linux system, then any flavor of Linux operating system containers can run given the container based on the same Linux Kernel.

Interestingly, all the containers of applications we build their base image are some flavor of Linux distributions (like Debian, Ubuntu, Alpine). If we look at fog devices, they are mostly network equipment such as Gateway devices, Network switches, and Routers. All those devices run some flavors of the Linux Operating System. So if we can have a Docker Engine on fog devices, we can run a Docker container. Given containers use the same Linux Kernel.

4.2.2 Compatibility over heterogeneous CPU architectures with Buildx

The conventional way to shift a docker container is as a Docker image. A docker image is created at the developer end and pushed to the Docker Registry. Then we can start the container by pulling the Docker image from the Docker Registry at the deployment end. Usually, the development and deployment sides used to have the same or different chips based on the same architecture (e.g., Intel Core at developer and Intel Xeon at production). However, fog devices are heterogeneous and consist of different processor architectures. An image built on one processor architecture will not work on another processor architecture. A straightforward example is Intel Core at the developer end and an ARM-based processor on a Raspberry Pi.

The solution to this problem is through Docker Multi-architecture Manifests. Creating Docker images of different CPU architectures of the same application component to support multiple hardware architectures, Docker automatically pulls the compatible Docker images from the Docker Registry during deployment. There are two ways to build such types of images.

1. The old way is to build the image on each architecture natively and then create a combined manifest file and push it to the Docker registry.

2. With the new approach, Docker introduces a CLI tool Buildx[4] to build multi-architecture images, combine them in a manifest file and push them to the Docker registry with a single command. Buildx uses QEMU[5] emulation support from the Linux Kernel to emulate multiple processor architectures and build multi-architecture images parallelly.

However, there could be some issues, like not every base image supports multi-architecture. Buildx may fail for many reasons for some uncommon applications, such as the unavailability of dependency. In that case, the older way is the only way to go.

4.3 Interoperability

The second essential requirement of fog computing is interoperability which ensures seamless cooperation and coordination between services running across fog nodes. Docker provides the native container orchestration tool called Docker Swarm[6]. Since the fog services are running in Docker containers, anything that runs well in standalone containers runs equally well in swarm mode. Therefore, this **second layer of abstraction handles interoperability with the Docker swarm and enables seamless coordination and cooperation in the fog federation.** In the following subsections, we illustrate the networking aspect of standalone Docker containers on a single host, then the networking of swarm mode operation across multiple host machines.

4.3.1 Docker networking

Let us assume all our fog devices are installed with Docker Engine so we can treat them as Docker Host. A Bridge Network[7] (`172.17.0.0/16`) will be created on each Docker Host by default. After starting, all containers get attached to this Bridge Network. In Figure 4.2, Container 1 and Container 2 get the IP address

[4] https://docs.docker.com/buildx/working-with-buildx/
[5] https://www.qemu.org
[6] https://docs.docker.com/engine/swarm/
[7] https://docs.docker.com/network/bridge/

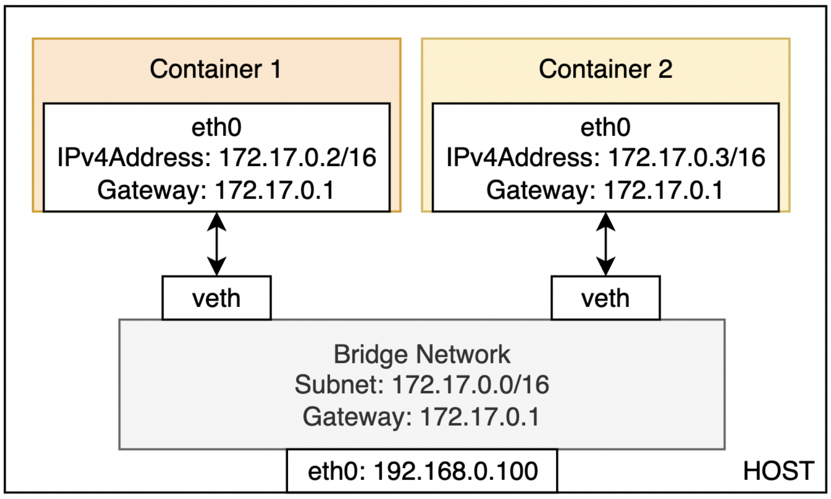


Figure 4.2: Networking inside a Docker host

`172.17.0.2` and `172.17.0.3`, respectively. On a Docker Host, these containers can talk to each other through their IP addresses of each other. If the Bridge Network is user-defined, that is, by default DNS enabled, the container names also can be used to communicate with each other.

However, we are supposed to use many fog devices in fog computing. Then how will the communication occur between the containers running on different Docker Hosts? That is where Overlay Network[8] comes into the picture. It is a shared network over all the Docker Hosts. Therefore, we used Container Orchestration Tool called Docker swarm.

4.3.2 Interoperability over heterogeneous devices with swarm

Docker Swarm is a native clustering engine for or by Docker that creates a pool of Docker Hosts that virtually acts like a single machine from an external view, as shown in Figure 4.3. Any service in a standalone Docker container can equally run well in swarm mode (there is some limitation to be discussed in the upcoming Subsection 4.3.3). Docker swarm, by default, creates an Ingress Network[9]. This Ingress Network is one type of Overlay Network spread across all Docker Hosts joined in the swarm shown in Figure 4.4. It has an inbuilt load balancer and routing mesh. The load balancer distributes the traffic across the multiple replicas of

[8]`https://docs.docker.com/network/overlay/`
[9]`https://docs.docker.com/engine/swarm/ingress/`

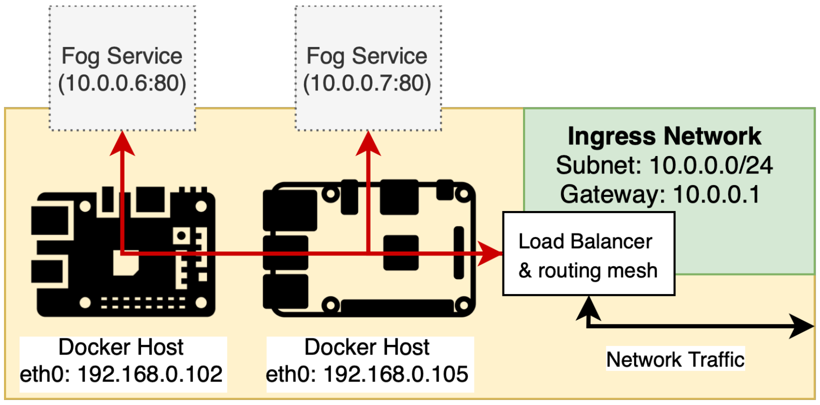


Figure 4.3: Interoperability with Docker swarm

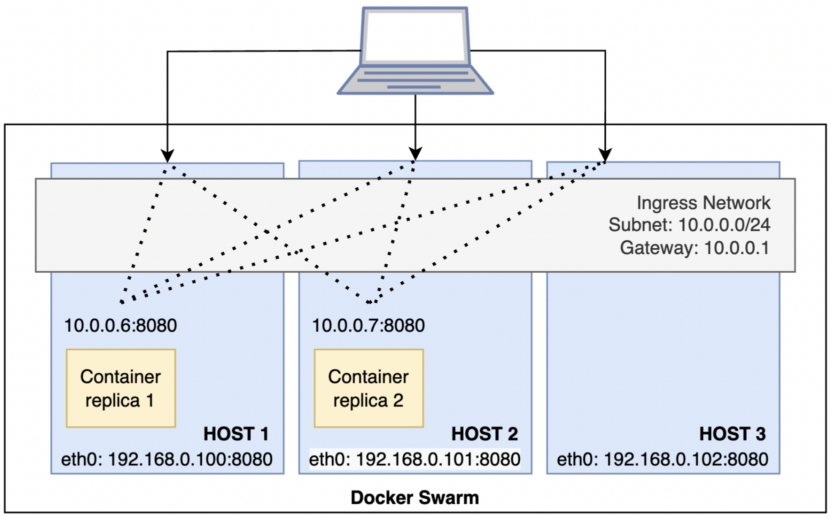


Figure 4.4: Networking across multiple Docker hosts in (Swarm mode)

a service running in different Docker Hosts. The routing mesh automatically redirects the traffic to that specific container where the service is running. Therefore, the service becomes available through all the IP addresses of fog devices. So all the fog devices can use any service running in the swarm of fog nodes without knowing the actual device where that container is running. This communication is private traffic inside swarm. With port mapping service made available to the external devices through all the IP addresses of fog nodes. The port mapping is similar to the bind-mount in the file system. The container's internal port is mapped with one host port. Any traffic coming to that Docker Host port is redirected into that mapped container's internal port. In Figure 4.4, HOST 1, 2, and 3's port 8080 of Docker Host are mapped with the docker service port `8080`. Any traffic coming to ports `8080` of HOST 1, 2, and 3 is redirected to containers running inside HOST 1 and HOST 2. The HOST 3 is not running the container, but still, service is available through `192.168.0.102:8080` because of the routing mesh.

4.3.3 Addressing different capabilities over heterogeneous hardware

Here capability is not about the processing, but about the capability of sensing and actuation. All fog devices included in the fog federation are not having the hardware support to interface a sensor and actuators. Even this issue is not about the placement of the container. Docker swarm offers functionality (labels and constraints) to specify the suitable Docker Host to deploy a container. The problem is with containerizing the sensing and actuation components. The container is OS-level virtualization technology to create logical isolation from the platform. However, sensing and actuation required direct platform and hardware interaction. Therefore, it is a conflict of interest. For the processing and communication (web server or message broker) components of an application, platform independence is achievable. However, we are still heavily dependent on the hardware platform for sensing and actuation.

Here are three issues we came across while experimenting:

1. Secure computing mode is a Kernel feature that restricts the actions inside a container. By default, out of 300+ System Calls, 44 System Calls are disabled[10].

2. We have to interact with the hardware directly for sensing and actuation, but the container is an isolated environment.

3. Installation of packages such as ‘`RPi.GPIO`’ inside a container fails because the base image is regular Linux distributions (e.g. ubuntu:20.04, debian:buster-slim). So, the platform is unsupported.

We figured out these two ways to dynamically deploy a sensing and actuation service on a Raspberry Pi.

[10]https://docs.docker.com/engine/security/seccomp/

Privileged container with bind mounts

These three issues of containerizing sensing and actuation are possible to address as follows.

1. All Docker containers are unprivileged by default, limiting the host machine's Kernel features and System Calls. Docker enables access to entire Kernel features on a host for a privileged container. A privileged container can load a Kernel module and function as same as a process natively running in a host machine.

2. Linux treats everything as a file. The path to the Raspberry Pi's GPIO pin is `‘‘/sys/class/gpio’’`. So we bind-mounted[11] the `‘/sys’` directory of the container to the `‘/sys’` directory of the Raspberry Pi. Bind mount means the container will mount one directory from the host machine, so here containers `‘/sys’` directory is mapped with Raspberry Pi's `‘/sys’` directory.

3. Similarly, All the libraries and packages are installed on the Raspberry Pi natively. That directory is bind-mounted inside the container (`‘/usr’` directory is bind-mounted with the container's `‘/usr’` directory).

After these three steps, executing a script inside a container for sensing and actuation becomes possible shown in Figure 4.5.

However, there are some limitations in Docker swarm. We get all the features in a standalone container; a few are not available in Swarm mode. The privileged container is one such feature that is not available in Docker swarm mode yet. So, the sensor and actuation service deployment must be as a standalone container shown as green boxes inside HOST 1 and 2 in Figure 4.5. These standalone containers can communicate with other services through the host IP address or any IP address of the fog node inside Docker swarm. The routing mesh will redirect the traffic to the correct container. This communication is acting as

[11] `https://docs.docker.com/storage/bind-mounts/`

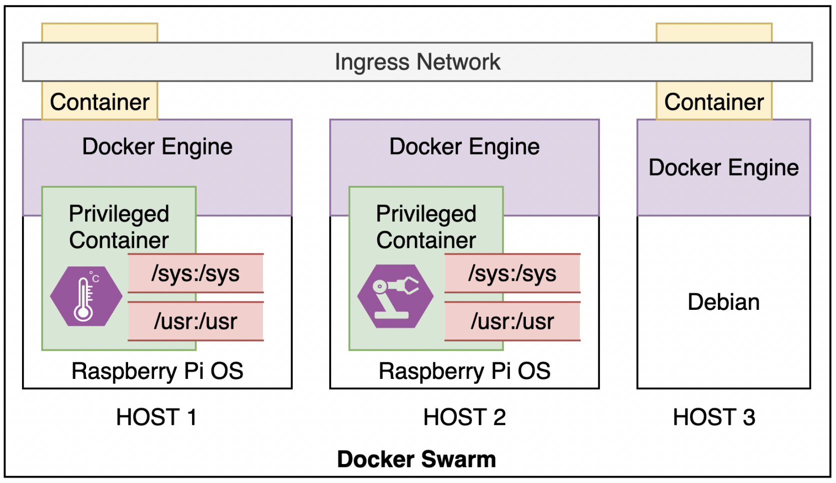


Figure 4.5: Privileged containers with bind-mount on a Docker host

an external device communicating from outside with the services running in the swarm. Otherwise, we can create another Overlay Network and attach the Docker services running and standalone container.

Native background service

In the containerization approach, the container is just providing the source code. Almost everything else is entirely dependent on the platform. This is unnecessarily adding up overhead on the fog devices. Therefore, it is better to execute natively on the host machine as a background service. Therefore, instead of starting a container, a background service of sensing or actuation will be created and started on the host machine Operating System during deployment shown in Figure 4.6. This communication with the services running in the swarm is possible through any IP address of the hosts in the swarm cluster exactly like the previous method. Our experiment tells us that this mechanism is much more robust and resource-efficient than the containerization process discussed in the previous Subsection 4.3.3 (Privileged container with bind mounts).

4.4 Standardization

To enable the portability of a fog application from one fog federation to another requires some standardization. It is a similar type of problem faced by the cloud

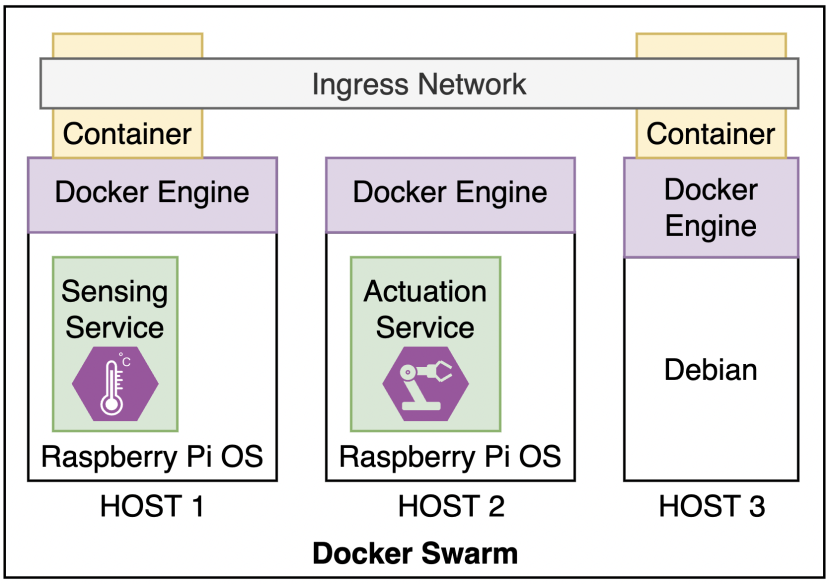


Figure 4.6: Background service for sensing and actuation

communities while porting a composite cloud application from one cloud service provider to another one. OASIS addressed the problem with **Topology and Orchestration Specification for Cloud Applications (TOSCA)**, which standardizes a composite cloud application description. This **third layer of abstraction provides a TOSCA extension for modeling fog applications.** Therefore, this section briefly discusses TOSCA backgrounds, followed by the extension (TOSCA new Nodes and Relationship types) for describing fog applications.

4.4.1 TOSCA background

TOSCA is a standardized specification to describe software applications to run in the cloud. TOSCA describes not only the application, but also describes the dependency and supporting infrastructure of an application in the cloud. These end-to-end descriptions (from base infrastructure, networks, and software to the running composite application) make TOSCA vendor-independent, and interoperability and portability enabled service definitions for the cloud applications.

TOSCA has two basic building blocks: Nodes and Relationships. Nodes are the infrastructure (e.g., servers, networks, virtual machines) or software components (e.g., services and runtime environments). Relationships describe the relationship between nodes (e.g., a virtual machine **hosts** a web server). In a TOSCA Service Template (typically a YAML file), we create a Topology Template. The Topology Template consists of Node Templates and Relationship Templates describing the composite application's blueprint.

Each Node and Relationship Templates has a Node Type and Relationship Type. Suppose we relate TOSCA with object-oriented programming concepts. In that case, Node Types or Relationship Types are the classes, and Node and Relationships are the objects. TOSCA Simple Profile v1.3[12] comes with some normative Node Types (e.g., Compute, SoftwareComponent, WebServer, WebApplication, DBMS, Database, ObjectStorage), Relationship Types (e.g., HostedOn, ConnectsTo, DependsOn, AttachesTo, and RoutesTo), Capabilities Types, Data Types (e.g., string, integer, list, dictionary). For modeling, user-developed applications custom Node Types, Relationship Types, and Capability Types are created by extending available Normative Types. Each Node and Relationship Type can have a set of attributes, properties, requirements, capabilities, and implementation. In the implementation, Nodes interface with the system and execute implementation scripts (Shell scripts, Python scripts, or Ansible Playbook scripts), based on the orchestration platform[13]. A Node's lifecycle is maintained with associated implementation scripts with interface operations: `create`, `configure`, `start`, `stop`, and `delete`.

It is important to note that **TOSCA is just a standard**. The only thing TOSCA gives is a string for the implementation, and it is up to the orchestrator to interpret that string and make sense of it.

[12]`https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html`

[13]`https://wiki.oasis-open.org/tosca/TOSCA-implementations`

Table 4.1: List of node types for fog applications

Nodes Types	Description
`docker_containers`	Pull a `docker-compose.yaml` file from the given URL and deploy/undeploy on the host fog node
`docker_services`	Pull a `docker-compose.yaml` file from the given URL and deploy/undeploy on the Docker swarm from Swarm Leader node
`swarm_leader`	Initiates Docker swarm on the host node and the node becomes Swarm manager
`swarm_worker`	Host node joins the Docker swarm as worker node
`system_service`	Pulls scripts and configuration files from the given URL and creates a background service with `systemctl` command

Table 4.2: List of relationship types for fog applications

Relationship Types	Description
`token_transfer`	Relationship (dependency) between Swarm Leader and Swarm Worker

4.4.2 TOSCA extension for fog application

The **Radon project**[14] and **xOpera examples**[15] repository (open-source) consist of a list of TOSCA node types that are publicly available. However, These two projects and the development of TOSCA are for the cloud platforms. Therefore, these node types are for the deployment of pure cloud applications. Hence, significant modifications are required to create relevant node types for deployment of IoT applications on fog nodes. In general, there will be five generic Nodes[16] and one Relationship[17] type required for user-defined applications. These Nodes and Relationship types are listed in the Table 4.1 and table 4.2 respectively.

[14]https://github.com/radon-h2020

[15]https://github.com/xlab-si/xopera-examples.git

[16]https://github.com/cloud-and-smart-labs/fog-service-orchestration/tree/swarm/orchestrator/tosca/nodetypes

[17]https://github.com/cloud-and-smart-labs/fog-service-orchestration/tree/swarm/orchestrator/tosca/relationshiptypes

Swarm Leader

The node initiates the Docker swarm and works as the swarm manager. The attributes (Listing 4.1, line 4-13) specify the joining token and advertise address of the swarm. Swarm Leader has the requirement of type Compute, so it must be hosted on a Compute type node (line 15). Furthermore, the node can host any positive number (line 21) of Docker services (theoretically). In interfaces, two operations (line 34-35), `create` and `delete`, are implemented with Ansible Playbooks. The orchestrator will invoke these Ansible Playbook scrips during deployment and undeployment, respectively. Inside the inputs section (line 26-32), the host node's IP address is concatenated with port `‘2377’` will be available as an input inside Ansible Playbooks. Now, it is up to tasks written into Ansible Playbooks to make the deployment and undeployment happen on the host node.

Swarm Worker

The node joins the Docker swarm as a worker node. Must be hosted on the Compute type node and depend on the Swarm Leader type node specified by the Token Transfer relationship type (Listing 4.2).

Swarm Leader and Swarm Worker nodes have a similar capability to host Docker Services. In interfaces, `create` and `delete` operations are implemented with three inputs: `worker_join_token` (to join the Docker swarm), `ip_address` (Host IP address), and `join_addr_port` (Docker swarm joining address from Swarm Leader) (Listing 4.3).

Docker Service

Docker Container

System Service

These three node types are more or less the same. Docker Service and Docker Container pull a `’docker-compose.yaml’` file from the online repository. The system service pulls the scripts and service configuration files for background ser-

```yaml
node_types:
 fog.docker.SwarmLeader:
  derived_from: tosca.nodes.SoftwareComponent
  attributes:
   manager_token:
    type: string
    default: undefined
   worker_token:
    type: string
    default: undefined
   advertise_addr:
    type: string
    default: undefined
  requirements:
   - host:
      capability: tosca.capabilities.Compute
      relationship: tosca.relationships.HostedOn
  capabilities:
   host:
    type: tosca.capabilities.Container
    occurrences: [0, UNBOUNDED]
    valid_source_types: [fog.docker.Services]
  interfaces:
   Standard:
    type: tosca.interfaces.node.lifecycle.Standard
    inputs:
     advertise_addr:
      value:
       concat:
        - get_attribute:[SELF, host, private_address]
        - ":2377"
      type: string
    operations:
     create: playbooks/create.yaml
     delete: playbooks/delete.yaml
```

Listing 4.1: Swarm Leader node type

```yaml
requirements:
 - leader:
    capability: tosca.capabilities.Node
    relationship: fog.docker.swarm.relationships.TokenTransfer
 - host:
    capability: tosca.capabilities.Compute
    relationship: tosca.relationships.HostedOn
```

Listing 4.2: Requirements of Swarm Worker node types

```
interfaces:
 Standard:
  type: tosca.interfaces.node.lifecycle.Standard
  inputs:
   worker_join_token:
    value:
     get_attribute: [SELF, leader, worker_token]
    type: string
   ip_addr:
    value:
     get_attribute: [SELF, host, private_address]
    type: string
   join_addr_port:
    value:
     get_attribute: [SELF, leader, advertise_addr]
    type: string
```

Listing 4.3: Interfaces of Swarm Worker node types

```
1 requirements:
2  - host:
3     capability: tosca.capabilities.Container
4     relationship: tosca.relationships.HostedOn
5     occurrences: [1, 1]
6  - dependency:
7     capability: tosca.capabilities.Container
8     relationship: tosca.relationships.DependsOn
9     occurrences: [0, UNBOUNDED]
```

Listing 4.4: Requirements of Docker Service node type

vice creation. Then again, it is up to the implementation in the Ansible Playbook to make the container or service creation happen on the host system. The only difference is the requirement. Docker Service node type deploys the service in swarm mode, so has the dependency on the Swarm Leader nodes and all Swarm Worker nodes inside the swarm (Listing 4.4).

Whereas background services and standalone containers run on a single node, the Compute node is the only requirement. Therefore, in Listing 4.4, the requirement of a 'host' is sufficient (line 2-5). The dependency section is not required for Docker Container and System Service.

```
1  operations:
2   pre_configure_source:
3    inputs:
4     manager_token:
5      value: { get_attribute: [TARGET, manager_token] }
6      type: string
7     worker_token:
8      value: { get_attribute: [TARGET, worker_token] }
9      type: string
10    join_addr_port:
11     value: {get_attribute: [TARGET, advertise_addr]}
12     type: string
13   implementation: playbooks/pre_configure_source.yaml
```

Listing 4.5: Interfaces of Token transfer relationship type

Token transfer

The relationship type specifies the relationship between Swarm Leader and Swarm Worker (Listing 4.5). Which is derived from normative `'dependOn'` Relationships type. The relationship sets the attributes joining tokens and advertised address of Swarm Leader node with `'pre_configure_source'` interface operation (line 13).

These are the generic and common types of TOSCA node types. Now, based on the different IoT Application requirements, other types of components may arrive. In that case, these node and relationship types have to be created like the way these five nodes and one relationship type are created. With these Nodes and Relationship types, fog applications can be designed and deployed by a TOSCA-compliant orchestrator. It is important to note that all these operations, including creating Docker Swarm, joining all the fog devices to the swarm cluster, deploying the services, and starting privileged containers or background services, will be **wholly automatic and handled by the FogDEFT framework without any human intervention**. Deployment or undeployment of fog services will take place by **single command or a button click** to provide the highest level of convenience. Chapter 6 will demonstrate such deployment with two different case studies of fog applications.

4.5 Summary

The FogDEFT framework uses three-layer abstractions to deal with platform independence, interoperability, and portability. The first layer of abstraction incorporates the OS-level virtualization to run the services on the fog nodes. Since hardware virtualization is resource-intensive, Docker containers are critical for deploying services on resource-constrained fog nodes. The second layer of abstraction uses Docker swarm, a container orchestration tool by Docker, to establish the interoperability of services through the Overlay Network (a kind of shared network across multiple fog nodes). The third and last layer of abstraction extends TOSCA standards and offers generic Nodes and Relationship types to describe the blueprint of a fog application through a TOSCA Service Template. The service becomes portable because the orchestrator can deploy these services to any fog infrastructure given a TOSCA Service Template.

Chapter 5

Dynamic deployment on fog

The deployment of these services on the fog nodes requires TOSCA Compliant orchestrators. TOSCA Compliant Orchestrator consists of a TOSCA processor, an engine or tool capable of parsing, validating, and interpreting the Topology Template. Therefore, TOSCA Compliant Orchestrator interprets a TOSCA Service Template to instantiate, deploy, and manage the application. TOSCA Compliant Orchestrator is also called an orchestration engine. The FogDEFT Framework adopted the **xOpera**[1] orchestrator to deploy the services on the fog nodes.

5.1 xOpera Orchestrator

The xOpera is a lightweight OASIS TOSCA compliant orchestrator (currently with TOSCA Simple Profile in YAML v1.3) that fits resource-constrained fog devices. It uses the Ansible Automation Tool to implement the TOSCA standard. Hence, all the interface operations are performed with Ansible Playbooks. Therefore, all the Nodes and Relationship types listed in Tables 4.1 and 4.2 have associated Ansible Playbook scripts (`create.yaml`, `delete.yaml`) for interface operations, as shown in Listing 5.6 (line 14-15). The xOpera understands TOSCA Service Template and executes Ansible Playbooks in a particular order based on the node's dependency to make deployment and undeployment happen. These Ansible Playbooks execute on the orchestrator (can be a remote machine) and

[1]https://xlab-si.github.io/xopera-docs/02-cli.html

```
interfaces:
  Standard:
    type: tosca.interfaces.node.lifecycle.Standard

    inputs:
      name:
        value: { get_property: [SELF, name] }
        type: string
      url:
        value: { get_property: [SELF, url] }
        type: string

    operations:
      create: playbooks/create.yaml
      delete: playbooks/delete.yaml
```

Listing 5.6: TOSCA node's interface operations

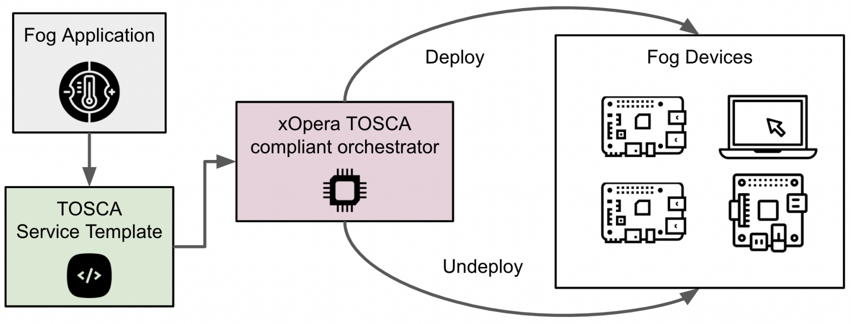


Figure 5.1: Fog service orchestration process

connect fog nodes to push small programs called ansible modules (Figure 5.1). This connection is over Secure Shell (SSH) by default. Ansible then executes these modules and then removes them after completion. Hence, the service deployment functions out of the box on the fly. The xOpera orchestrator and Ansible both heavily rely on SSH infrastructure. The machine where the orchestrator is running must be able to access all the fog devices through the SSH.

5.2 Dynamic service deployment

The fog service deployment with the FogDEFT framework requires a two-step procedure modeling the application followed by orchestration.

5.2.1 Modeling

Before deployment, the fog service's TOSCA Service Template needs to be created using the Nodes (Table 4.1) and Relationship (Table 4.2) Types provided by the FogDEFT framework discussed in the previous chapter. The file structure of the entire package is shown in Figure 5.2. In that file structure `'service.yaml'` file is the TOSCA Service Template for fog application consisting of the topology of all the Node Templates describing the end-to-end architecture of the application. Furthermore, the `'inputs.yaml'` file contains the list of the IP address of fog nodes (just changing these IP addresses will allow the deployment on some other fog nodes inside another network), URI or repository links of `'docker-compose.yaml'` files supposed to be deployed on specific fog nodes or in swarm mode. The basic thumb rule is that hardware-independent services like message broker and webserver should be deployed in swarm mode, and hardware-dependent services should be deployed in standalone mode or as system services. The next chapter will illustrate TOSCA Service Template creation in detail with case studies.

5.2.2 Orchestration

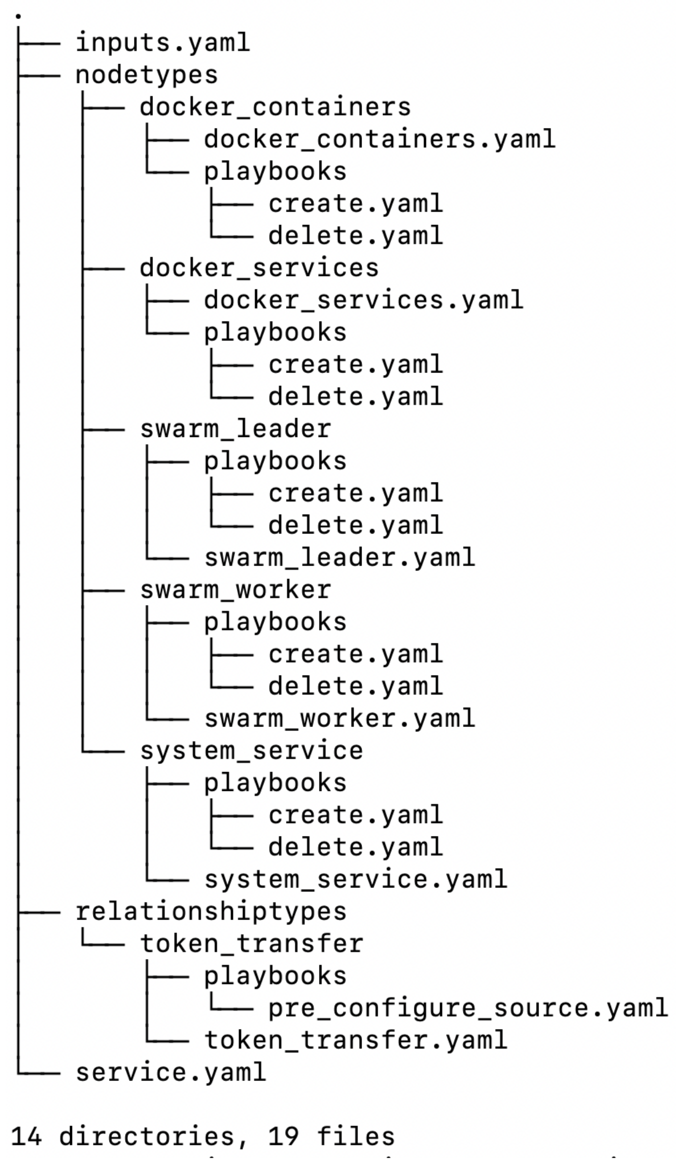
The FogDEFT framework adopted the xOpera orchestrator to deploy the services on the fog nodes discussed in Section 5.1. The xOpera Orchestration tool is available/distributed as a Python package[2]. Assuming the organization of the fog nodes like Figure 5.3, where the personal computer in the middle is running the orchestrator and remotely deploying fog service to all other nodes in that network. To deploy the service with xOpera following steps[3] need to be followed.

Environment Setup

The xOpera and Ansible both use SSH for the deployment of service. Therefore the orchestrator machine must be able to login into all the fog devices through

[2]https://pypi.org/project/opera/

[3]https://github.com/cloud-and-smart-labs/fog-service-orchestration/blob/swarm/README.md



```
.
├── inputs.yaml
├── nodetypes
│   ├── docker_containers
│   │   ├── docker_containers.yaml
│   │   └── playbooks
│   │       ├── create.yaml
│   │       └── delete.yaml
│   ├── docker_services
│   │   ├── docker_services.yaml
│   │   └── playbooks
│   │       ├── create.yaml
│   │       └── delete.yaml
│   ├── swarm_leader
│   │   ├── playbooks
│   │   │   ├── create.yaml
│   │   │   └── delete.yaml
│   │   └── swarm_leader.yaml
│   ├── swarm_worker
│   │   ├── playbooks
│   │   │   ├── create.yaml
│   │   │   └── delete.yaml
│   │   └── swarm_worker.yaml
│   └── system_service
│       ├── playbooks
│       │   ├── create.yaml
│       │   └── delete.yaml
│       └── system_service.yaml
├── relationshiptypes
│   └── token_transfer
│       ├── playbooks
│       │   └── pre_configure_source.yaml
│       └── token_transfer.yaml
└── service.yaml

14 directories, 19 files
```

Figure 5.2: Files organization of the framework

SSH without a password[4].

1. Generate SSH key pair.

```
$ ssh-keygen
```

2. Copy the public key to all fog nodes.

```
$ ssh-copy-id root@192.168.0.XXX
```

3. Install xOpera though package installer for Python (PIP).

[4]Otherwise, this will ask for the password for each fog node for each service to be deployed.

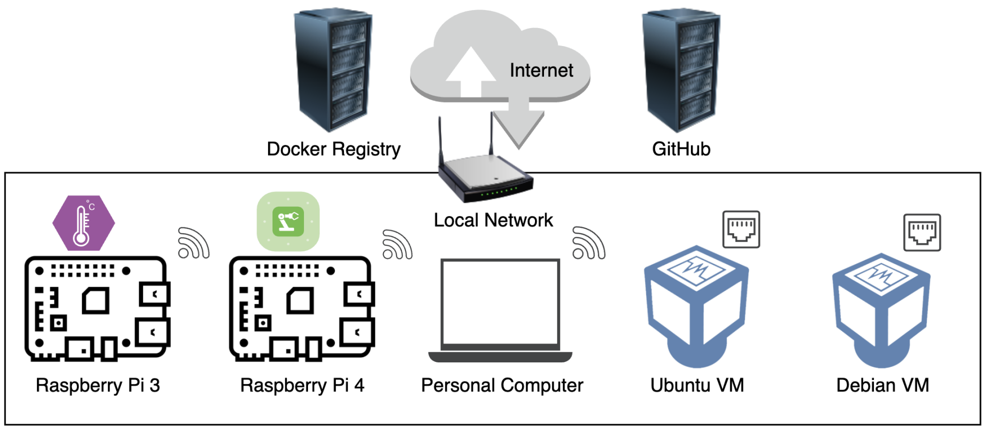


Figure 5.3: Fog devices organization in local area network

```
$ pip install opera==0.6.8
```

4. By default xOpera login as `'centos'` username. To change the login username set the `'OPERA_SSH_USER'` environment variable to the fog nodes username.

```
$ export OPERA_SSH_USER=root
```

Service deployment/undeployment

The commands that need to be executed to **validate, deploy, and undeploy** service through the xOpera orchestration is listed below.

- **Validate** TOSCA Service Templates before deployment.

  ```
  $ opera validate -e -i inputs.yaml service.yaml
  ```

  `-e`: executors (e.g. Ansible Playbooks) behind them
  `-i`: Input file path

- **Deploy** TOSCA Service Templates.

  ```
  $ opera deploy -w 2 -i inputs.yaml service.yaml
  ```

  -i: Input file path
  -w: Number of concurrent threads (default 1)

At the time of deployment, the xOpera orchestrator executes Ansible Playbook scripts in a specific order to make this deployment happen discussed in the previous section. In the case of a single deployment thread (`-w 1` or empty), this specific order is one of the topological sorts of the applications services topology defined in the topology template inside TOSCA Service Templates. That order depends entirely on the node templates dependency for multiple concurrent deployment threads.

- **Undeploy** TOSCA Service Templates.

```
$ opera undeploy -w 2
```

`-w`: Number of concurrent threads (default 1)
Similarly, for the undeployment, the order follows the transpose graph of the application topology in the case of a single undeployment thread.

Resource utilization and performance

The development of TOSCA standards and Orchestrators is targeted for cloud instances. The cloud gives an illusion of an infinite amount of resources. However, we are using it to orchestrate services on fog devices. Fog devices are resource-constrained devices. So resource consumption could be an issue for service deployment on fog devices. For resource utilization and performance analysis, the fog service orchestration with xOpera orchestrator of the TOSCA Service Template consists of a ten node template deployed on three fog nodes (two Raspberry Pi 4 and one Raspberry Pi 3). The resource consumption (CPU and memory usage) and time taken for the deployment followed by undeployment are observed under the different number of concurrent deployments threads (Workers) and listed in Table 5.1.

Importantly, this data is a pure orchestrator's resource utilization on a typical personal computer (laptop). While successively deploying and then undeploying the service. This resource consumption does not include any other processes or applications we deploy. We ensure this through the custom Python script[5] that explicitly picks the resource consumption of the processes responsible for the orchestration. Each row in the Table shows deployment followed by deployment time, CPU, and memory utilization corresponding to the number of concurrent deployment threads employed. Therefore, in the case of a single thread, CPU and memory usage is significantly low but takes the longest time because the deployment order of nodes is sequential (order of topological sort). However, when more concurrent deployment threads are employed, deployment and undeployment times decrease significantly, and resource consumption increases drastically (peaks in a CPU and memory usage graph show that), mostly while deploying independent nodes parallelly.

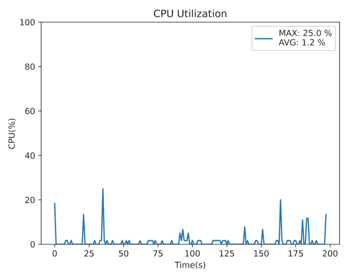
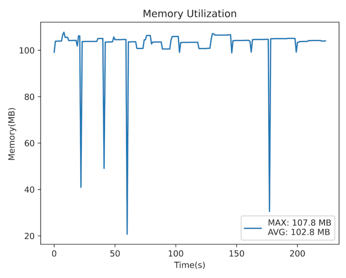
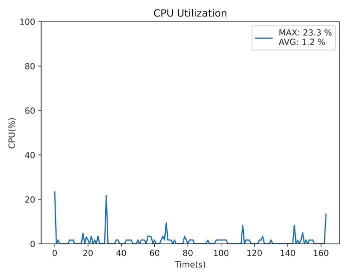
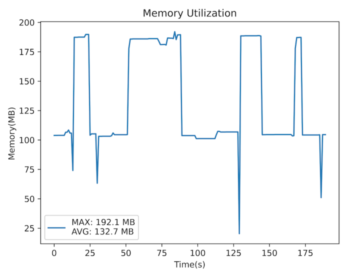
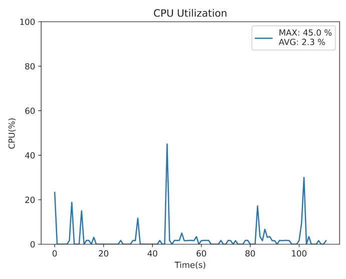
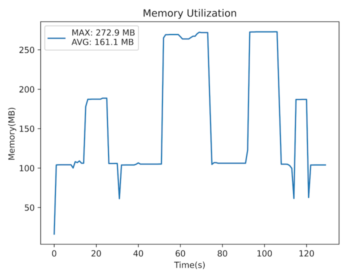
Deployment of the service for the first time can take more time for image pulling from the Docker registry and cloning the code from the repository. The second deployment of the same service will take significantly less time because of cached images on the fog devices.

5.3 Orchestration behind NAT

So far, we have discussed service deployment on-the-fly up to the previous subsection. The core mechanism is that all the fog nodes are running SSH servers. The Orchestrator connects to each node through the SSH. However, the "Out of the Box" core idea is that fog devices are not using exceptional configurations/services. Therefore, standard broadband service connects these devices from the organization/home network to the internet. There are high chances that the Internet Service Provider (ISP) has kept behind the Network Address Translation (NAT) due to the effective use of public IP addresses. It could be multiple layers of NAT in the case of a big organization or urban area. Hence the dynamic deployment is limited to the local network, shown in the organization of the fog federation Figure 5.3. With this, the scope of work gets stuck into a local area, probably at

[5]https://github.com/cloud-and-smart-labs/system-setup-util/tree/main/sys-monitor

Table 5.1: Orchestrator resource utilization and performance

Number of Workers	Time (s)	CPU (%)	Memory (MB)
1	Deploy real 139.90 usr 6.32 sys 1.60 Undeploy real 67.90 user 3.62 sys 0.90		
2	Deploy real 106.24 us 6.75 sys 1.77 Undeploy real 57.74 usr 4.10 sys 0.91		
3	Deploy real 91.57 user 7.56 sys 1.88 Undeploy real 37.73 user 4.43 sys 1.00		

most one site.

An external coordinator and a modified orchestration tool can bridge this gap. To support this type of deployment, we developed

Orchestration Manager (external coordinator)

This module is an interactive prompt like a command-line interface running on a static/public IP address. All the orchestrators connect to this manager and act as slaves/workers. The Orchestration Manager broadcasts the commands and configuration. The configuration consists of the name of TOSCA Node types and docker-compose files URI.

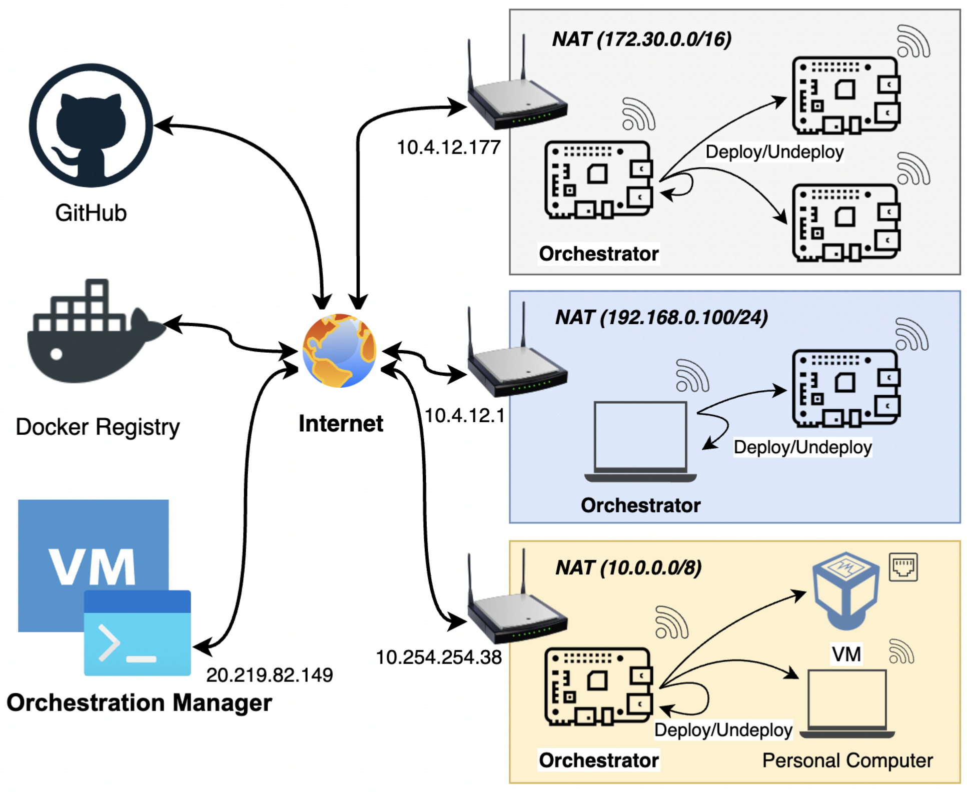


Figure 5.4: Deployment of services behind the NAT

Orchestrator (modified orchestration tool)

This Orchestrator is a package of

- xOpera orchestrator
- TOSCA Node types
- An agent to connect the Orchestration Manager and TOSCA Service Template generator

The Orchestrator can also work in standalone mode like a typical xOpera orchestrator. Furthermore, it acts as a slave/worker when connected to the Orchestration Manager. In connected mode when it receives a configuration from Orchestration Manager, it dynamically generates a TOSCA Service Template for that local network behind the NAT.

In our experiment, one of the fog devices runs the Orchestrator as a Docker container behind each NAT. The Orchestration manager runs on a Microsoft Azure[6] cloud instance as Docker container. When the respective configuration/command is received, all the Orchestrators dynamically generate the TOSCA Service Template based on the number of fog devices available there and deploy/undeploy on all the fog devices behind that NAT shown in Figure 5.4.

5.4 Summary

The FogDEFT framework adopts the xOpera orchestrator. xOpera orchestrator uses Ansible Automation Tool for the implementation of the TOSCA standard. Therefore, all the implementation scripts are written in Ansible Playbooks for all Node and Relationship types. At the time of deployment/undeployment, these scripts get executed in a specific order based on node templates dependency, and the deployment of these services takes place on fog nodes. For deployment of any user-defined fog services, first, the TOSCA Service Template needs to be created, and then the Service Temple will be given to xOpera orchestration, which will deploy/undeploy these services on a set of fog nodes with a single command. xOpera being a lightweight orchestrator, uses minimal resources in a single-threaded deployment. However, it gives the option to employ multiple concurrent threads to deploy independent nodes parallelly to speed up the deployment and undeployment process with the cost of higher resource consumption. If the fog nodes are behind the NAT, then Orchestration Manager and modified Orchestrator can be used to deploy services behind the NAT.

[6]https://azure.microsoft.com/en-in/services/virtual-machines/

Chapter 6

Case studies on FogDEFT

The previous two chapters discussed the abstraction layer, technology, and service deployment with the FogDEFT framework. This chapter will take two applications, Remote LED service and Climate Control system, as a case study to evaluate the FogDEFT framework. We first design each application's hardware and software architecture and then create a TOSCA Service Template to deploy the services dynamically. Furthermore, we will monitor these fog devices' performance and resource utilization while the services are getting deployed.

6.1 Remote LED

This section demonstrates the dynamic deployment with a simple IoT application with the FogDEFT framework. This application serves a webpage. With a button click from a web page, some LEDs get on or off.

6.1.1 System design

The design of an IoT-driven system consists of two parts: hardware and software.

Hardware design

The hardware setup for this application uses two Raspberry Pi 4 and one Raspberry Pi 3. Each of them has one LED connected through GPIO pin 18.

Software design

Five different Node Templates are creating the blueprint of this application:

- Compute Node: all the fog devices
- Swarm Leader: Docker Swarm manager
- Swarm Worker: Docker Swarm workers
- Docker Services: Hosts an Nginx web server, and serves the webpage and Python WebSocket server from where the actuation signal is forwarded to the actuators.
- Docker Containers/System Service: LED actuator: turn LED on/off

The visual blueprint of the structure of this application's TOSCA Service Template would be like Figure 6.1. The Fog nodes are the fog devices. On top of that, Swarm Leader and Swarm Workers are hosted. Swarm Workers depend on the Swarm Leader. Docker Service is hosted on the Swarm Leader with dependency on all Swarm Workers. Two Docker Services, Python Websocket Server and Nginx web server run as Docker Stack on the Docker Swarm (Swarm Leader and Swarm Workers). However, the Nginx web server depends on the Python WebSocket server mentioned in the '`docker-compose.yaml`' file. So Nginx webserver service will be deployed after deploying the Python WebSocket server. Therefore, LED actuators are hosted on each fog node but depend on Docker Services. So LED Actuators will be deployed after the Docker Services is ready.

6.1.2 TOSCA Service Template

The TOSCA Service Template[1] of the application contains the Topology Template given in Listing 6.7, that describes the application's structure given in the previous section Topology model shown in Figure 6.1. The Node Template section defines (line 2-48) all application components and their properties and dependencies.

[1]`https://github.com/cloud-and-smart-labs/fog-service-orchestration/blob/swarm/orchestrator/tosca/service-2.yaml`

```
topology_template:
 node_templates:
  fog-node-1:
   type: tosca.nodes.Compute
   attributes:
    private_address: 192.168.0.166
    public_address: 192.168.0.166
  fog-node-2:
    ...
  fog-node-3:
    ...

  docker-swarm-leader:
   type: fog.docker.SwarmLeader
   requirements:
    - host: fog-node-1

  docker-swarm-worker-1:
   type: fog.docker.SwarmWorker
   requirements:
    - host: fog-node-2
    - leader: docker-swarm-leader
  docker-swarm-worker-2:
    ...

  docker-service-1:
   type: fog.docker.Services
   properties:
    name: { get_input: docker_service_name }
    url: { get_input: docker_service_url }
   requirements:
    - host: docker-swarm-leader
    - dependency: docker-swarm-worker-1
    - dependency: docker-swarm-worker-2

  privileged_container-1:
   type: fog.docker.Containers
   properties:
    name: { get_input: docker_compose_name }
    url: { get_input: docker_compose_url }
    packages: { get_input: packages }
   requirements:
    - host: fog-node-1
    - dependency: docker-service-1
  privileged_container-2:
    ...
  privileged_container-3:
    ...
```

Listing 6.7: TOSCA Service Template of the application

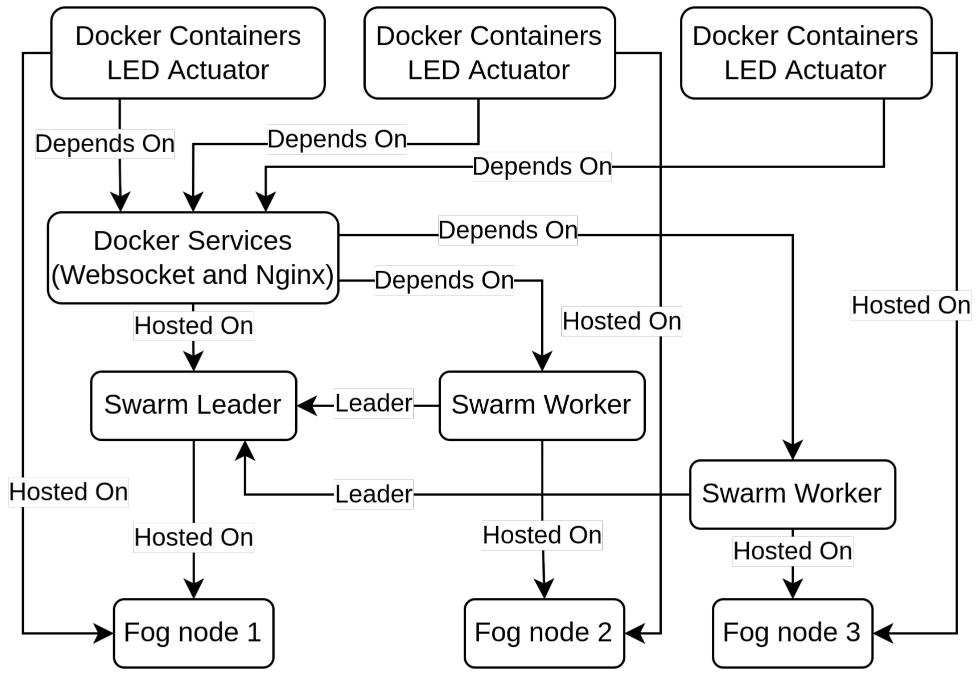


Figure 6.1: Topology or Blueprint of the application

```
system-service-1:
 type: fog.system.Service
 properties:
  name: { get_input: system_service_name }
  script_url: {get_input: system_service_script_url}
  service_url: {get_input: system_service_service_url}
  packages: { get_input: packages }
 requirements:
  - host: fog-node-1
  - dependency: docker-service-1
```

Listing 6.8: System Service node template

The Privileged Containers Node Templates need to be replaced by the System Service Node Templates to deploy the actuation as background services instead of standalone containers shown in Listing 6.8.

For the inputs, we have a separate file (‘`inputs.yaml`’) to add the required inputs repository or nodes IP addresses. The file structure of the entire package (node, relationship types, service template, inputs, and Ansible Playbooks) of an application is shown in previous Chapter Figure 5.2.

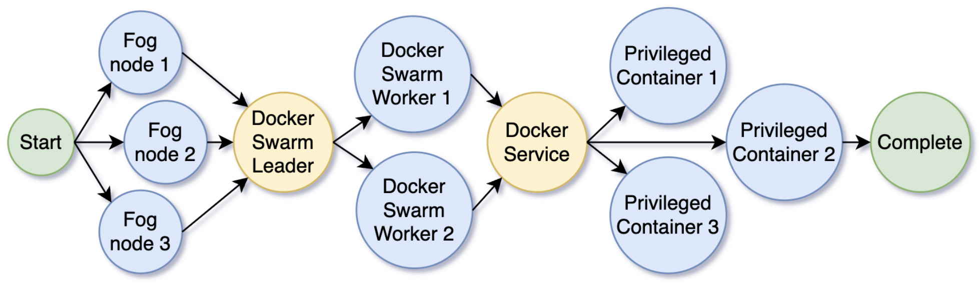


Figure 6.2: Deployment sequence with multiple workers (parallel threads)

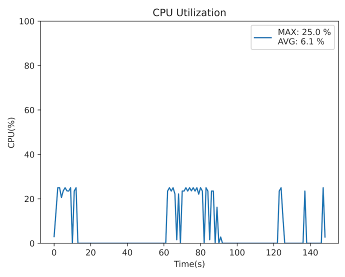
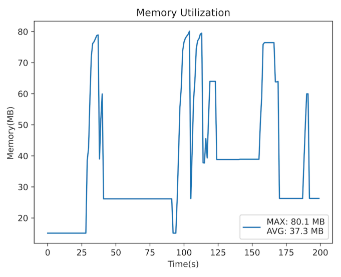
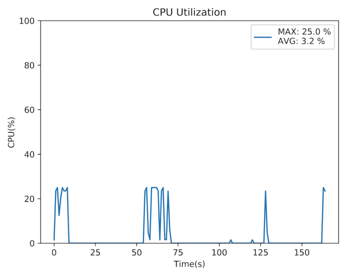
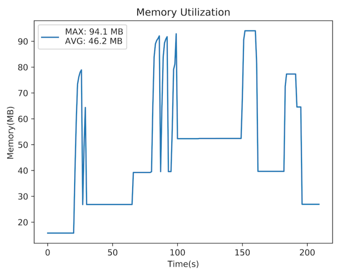
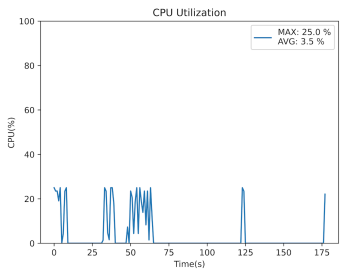
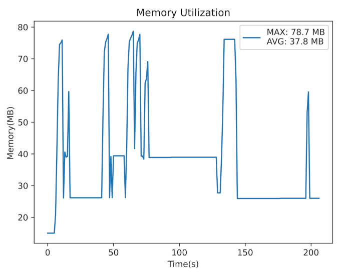
6.1.3 Results and discussion

Three concurrent deployment threads were employed during the orchestration as the experiment was going on three fog nodes as illustrated in section Topology model Figure 6.1. The orchestrator, based on the dependencies, parallelly deploys the nodes on the fog devices as shown in Figure 6.2. The fog nodes are first created parallelly in the deployment sequence, followed by the Docker Swarm Leader. All other stuff waited until the completion of the Docker Swarm Leader node, because all other nodes directly or indirectly had some dependency on the Docker Swarm Leader node. Then two Swarm Workers are created parallelly, followed by Docker Services that had the dependency on the entire Docker Swarm. After deploying the Docker Service, the orchestrator simultaneously deploys all three Privileged Containers.

Similarly, for the undeployment of the service, the orchestrator undeployment sequence is precisely the reverse order of the node's dependency in the Topology Template.

The resource utilization details of the fog nodes where the application services are getting deployed are shown in Table 6.1. These results show that irrespective of the devices and their role (Leader/worker) in fog federation, CPU and memory usage are more or less the same and favorable to entitle as a lightweight fog deployment framework.

Table 6.1: Fog nodes resource utilization and performance (remote-LED)

Device	CPU (%)	Memory (MB)
Raspberry Pi 3 Model B (As Worker node)		
Raspberry Pi 4 (As Worker node)		
Raspberry Pi 4 (As Leader node)		

6.2 Climate Control system

This section demonstrates the dynamic deployment by realizing a case study of the climate control system of the convention center. A convention center inside a city usually hosts diverse events like conferences, exhibitions, and cultural events. Probably a storage area for off times or a hospital isolation ward in times of pandemic, and the past two years made it clear. Therefore, all these events in different seasons require different climate conditions inside a convention center. For example, a cultural event needs different lighting requirements and intensity than an international conference. Even those requirements will differ from daytime to nighttime as well. The weather and season will play a significant role in climate control. A summer event requires lower temperature, a winter event higher temperature, and a monsoon needs lower humidity and temperature. The number of

Table 6.2: IoT device list for climate control system

Device Name	Processor Architecture	Memory Size	Operating System
Arduino Uno	Microcontroller ATmega328P	32 KB	-
Arduino Nano 33 BLE Sense	ARM Cortex-M4	1 MB	-
Raspberry Pi 4	ARMv7l 32-Bit	4 GB	Raspberry Pi OS 10
Raspberry Pi 4	ARMv8 64-Bit	4GB	Raspberry Pi OS 11
Workstation	Intel Xeon W-2145 3.70GHz $\times$ 16	32 GB	Ubuntu 20.04

guests is also a factor in controlling temperature and humidity. Altogether, the automation of these climate control systems is one of the ideal scenarios for the dynamic deployment of fog services (A similar idea can be used in greenhouse farming as well).

6.2.1 System design

The design of an IoT-driven system consists of two parts: hardware and software. The section described the requirement of a generic hardware platform with sensors and actuators capable of hosting some fog services. The following subsections illustrate the hardware prototype followed by the software architecture with FogDEFT framework on the hardware prototype, creating a fog federation for deploying the case study services

Hardware design

The prototype for the demonstration of dynamic deployment of fog services uses a handful of IoT devices listed in Table 6.2.

Figure 6.3 shows the illustration of the hardware design of the prototype. Two Arduino Nano 33 BLE Sense boards are placed outside the convention center and connected to a Raspberry Pi 4 through serial ports (USB). These two Arduino boards have inbuilt sensors (temperature, humidity, light, barometric pressure,

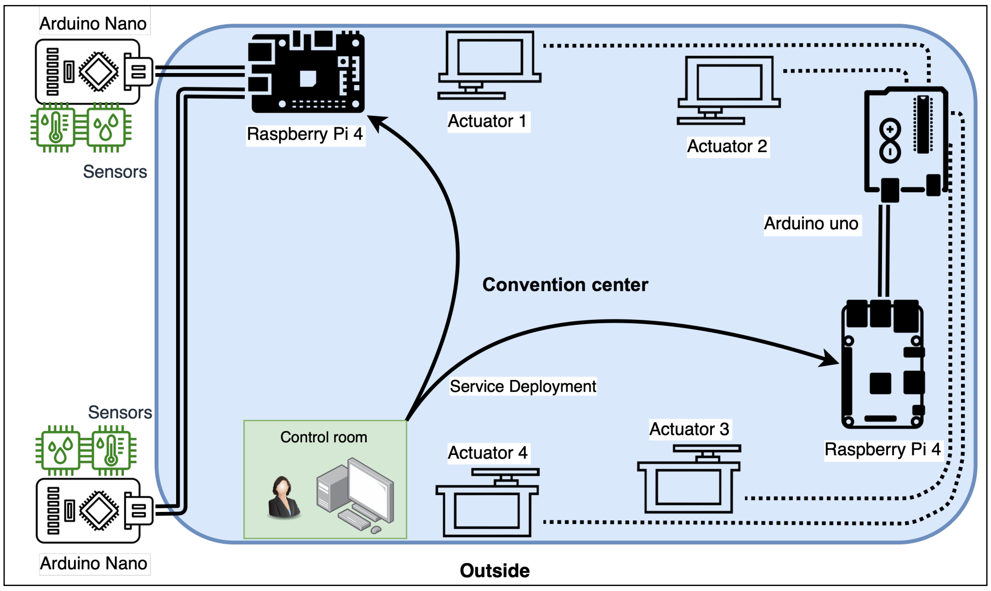


Figure 6.3: Hardware design

proximity, and microphone) to sense the outside environment. Another Arduino Uno is connected to four actuators (servo motors) through Pulse Width Modulation (PWM) pins and one Raspberry Pi 4 through a serial port. All the Raspberry Pis and the Workstation inside the control room are connected to the network and have internet connectivity.

Software design

After designing the generic hardware platform, it is up to the job of the software to create the platform for fog federation with these on-premises gateway devices (Raspberry Pis). The devices listed in Table 6.3 come under two different categories.

First, Arduinos come under the category of microcontrollers. We have three Arduinos here of two different categories: Nano 33 BLE Sense and Uno. The first one is for sensing the outside environments. Therefore, these two Arduino were programmed to collect the sensor dataset and send it through the serial port of that connected Raspberry Pi 4. The second one is for actuation to control the climate inside the convention center. Therefore, this is programmed to perform actuation on connected actuators. These actuation parameters are retrieved from

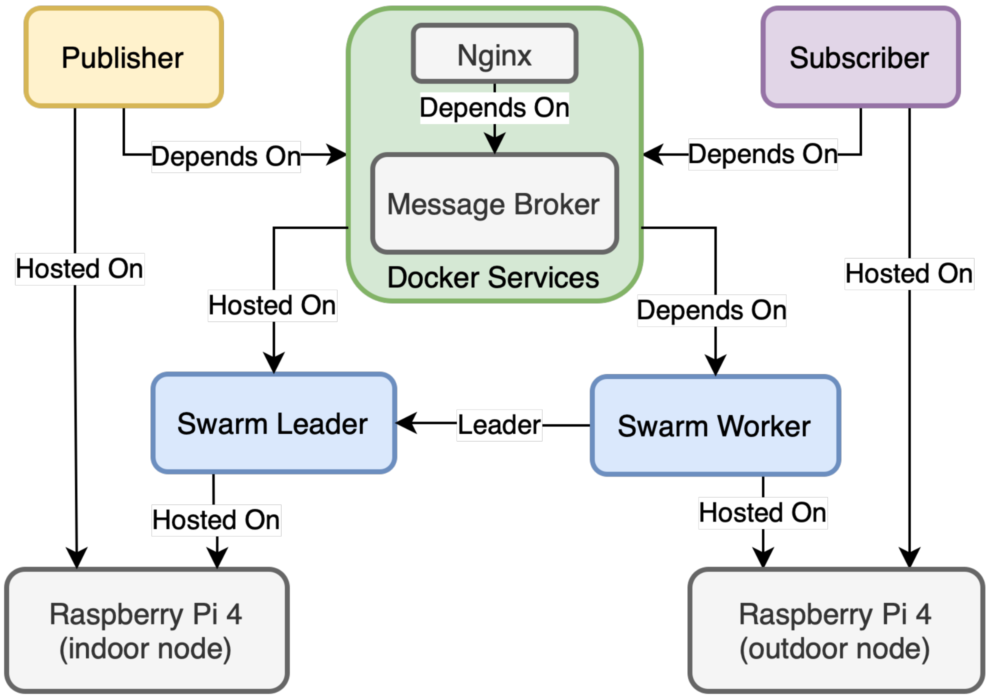


Figure 6.4: Service design blueprint

the serial port connected to the Raspberry Pi 4.

Second, Raspberry Pis and the system inside the control room come under the microprocessor category. Unlike Arduinos, these are typical computers with Operating Systems that can load programs and execute processes and services. Therefore, these devices will be treated as fog nodes. Here, the FogDEFT framework comes into the picture to ease the development and deployment of fog services on the fog nodes.

The design of the prototype in Figure 6.3 consists of two fog nodes (Raspberry Pi 4). The fog services to maintain the climate will be deployed on these two fog nodes. Here, Figure 6.4 illustrates the topology design of fog services to be deployed on the fog nodes inside convention centers to IoT-driven climate control systems. In this illustration, boxes are nodes, and directed edges are relationships between nodes, as discussed in Section 4.4.

The bottom two gray nodes represent the fog nodes. The green node in the middle represents Docker Services (Message broker and web server to show the sensor data and state) running in the swarm mode (interservice dependencies are mentioned in the docker-compose file). This Docker Service node is hosted and depends on two blue nodes, Swarm Leader and Swarm Worker, respectively. The edge between two blue nodes is their relationship, and both are hosted on one fog node, respectively. The remaining two yellow and purple nodes are Publisher and Subscriber services, respectively. The Publisher node pushes the sensor data to the message broker. Therefore this node is a standalone container hosted on the fog node connected to the Arduino outside. Similarly, the Subscriber node receives the broadcast of each update from the message broker and makes adjustments to actuators. Therefore, this node is also a standalone container hosted on the fog node connected to the Arduino wired with actuators.

This service blueprint of the design indicates that at least four different microservices (Message broker, web viewer, sensor data publisher, and Subscriber with climate controller) are required. Interestingly, this design illustrated in Figure 6.4 requires only one URI change inside the purple node named Subscriber. That could dynamically change to utterly different climate conditions, probably requiring one from a specific event. Hence, this fog federation framework provides a versatile platform to deploy services on demand on the fly.

6.2.2 TOSCA Service Template

With these Node and Relationship types provided by the FogDEFT framework the TOSCA Service Template[2] of the blueprint given in Figure 6.4 is shown in Listing 6.9.

6.2.3 Results and discussion

We deployed the Service Template given in Listing 6.9 on the prototype Figure 6.3 from a remote system (corresponds to the control room) with the xOpera orches-

[2] `https://github.com/cloud-and-smart-labs/climate-control/blob/main/tosca/service.yaml`

```yaml
topology_template:
  node_templates:
    outdoor-node:
      type: tosca.nodes.Compute
      attributes:
        private_address: 192.168.0.103
        public_address: 192.168.0.103

    indoor-node:
      type: tosca.nodes.Compute
      attributes:
        private_address: 192.168.0.105
        public_address: 192.168.0.105

    docker-swarm-leader:
      type: fog.docker.SwarmLeader
      requirements:
        - host: indoor-node

    docker-swarm-worker:
      type: fog.docker.SwarmWorker
      requirements:
        - host: outdoor-node
        - leader: docker-swarm-leader

    broker-service:
      type: fog.docker.Services
      properties:
        name: broker
        url: https://repo/brokr/docker-compose.yaml
      requirements:
        - host: docker-swarm-leader
        - dependency: docker-swarm-worker

    sensor-data-publisher:
      type: fog.docker.Containers
      properties:
        name: publisher
        url: https://repo/pub/docker-compose.yaml
      requirements:
        - host: outdoor-node
        - dependency: broker-service

    actuator-data-subscriber:
      type: fog.docker.Containers
      properties:
        name: subscriber
        url: https://repo/subs/docker-compose.yaml
      requirements:
        - host: indoor-node
        - dependency: broker-service
```

Listing 6.9: TOSCA Service Template

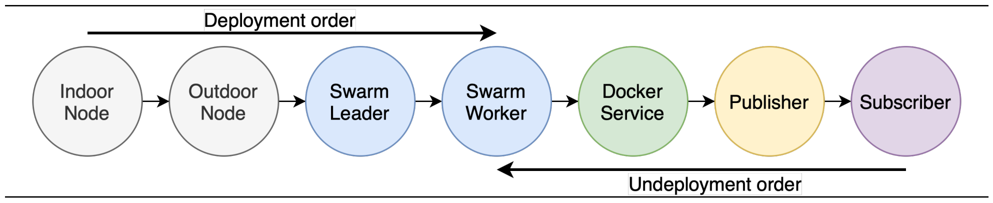


Figure 6.5: One of the valid orders for deployment/undeployment

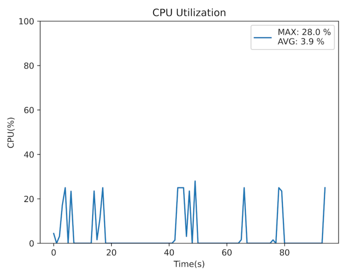
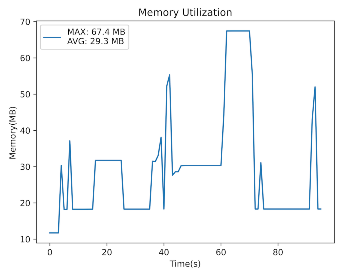
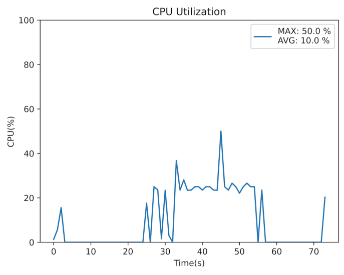
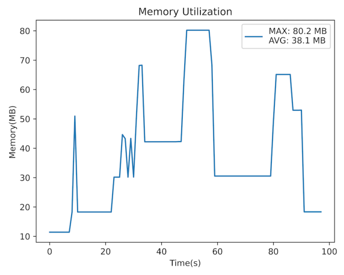
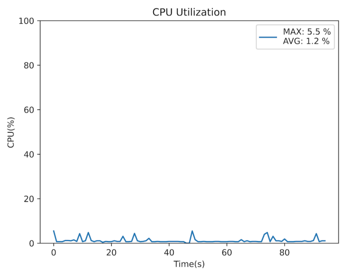
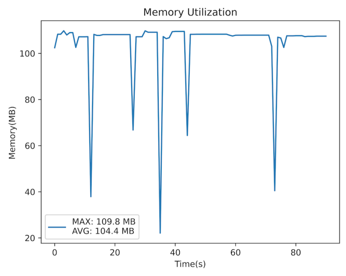
trator. The deployment and undeployment order of the nodes follows topological sort of application topology shown in Figure 6.4. Therefore, one of the valid deployment and undeployment orders is given in Figure 6.5.

The resource utilization of each node (Indoor, Outdoor, and Workstation) is given in Table 6.3. This resource usage only includes the resource consumption of these processes responsible for the orchestrations of the fog services. Resource consumption of fog services is out of the framework's scope. In our experiment, the deployment and undeployment took around 121.05s and 96.90s, respectively, from the workstation with a single thread.

6.3 Summary

The FogDEFT framework is evaluated with two case studies Remote LED and Climate control system. In Remote LED, services are designed and deployed on three Raspberry Pi connected to LED via GPIO pin and gives a LED toggle button through a webpage. In the Climate control system, the hardware is designed with two Raspberry Pi and three Arduino boards (two Nano and one Uno). The fog services are deployed on Raspberry Pi through TOSCA Service Templates, and sensing and actuation are provided through the Arduino boards. With a change of one Node Template inside the TOSCA Service Template, utterly different climate conditions are achievable.

Table 6.3: Resource utilization and performance (climate control)

Device	CPU (%)	Memory (MB)
Raspberry Pi 4 Indoor Node		
Raspberry Pi 4 Outdoor Node		
Workstation Orchestrator		

Chapter 7

Conclusion and Future Work

We have created TOSCA node types and relationship types for modeling a fog application using TOSCA to build a fog federation framework: FogDEFT that allows fog service on-demand deployment through a single command. The framework abstracts the heterogeneity of fog devices and provides a standardized platform for deploying custom or user-developed applications on the fly demonstrated in two case studies. The xOpera Orchestrator and ansible automation tool uses Secure Shell (SSH) infrastructure to push ansible modules for the deployment of the service. That makes the framework secure and agentless. Therefore, any custom/user-developed application deployment is possible out of the box on the fly with the given TOSCA Service Template.

Moreover, Docker containers have become de-facto standards for deploying services with support across various platform and hardware architectures. The fog devices have become more potent with the advancement of technologies, so deploying Docker containers with negligible overhead has become a reality. However, some devices will be left out of Docker support. For these, options for native deployment are always available with the System Service node type.

7.1 Future scope

The development of TOSCA primarily targets the standardization of cloud applications. However, this work demonstrates the potential of TOSCA in standardizing fog services for IoT-driven applications. The orchestrator used in this work was developed for the cloud application deployment in virtual machines. In IoT, non-IP-based networking is prevalent for M2M communication. Therefore, creating a lightweight IoT- focused orchestrator with non-IP-based network support can unlock huge possibilities like M2M or drone to drone on-demand service deployment. Therefore, it opens plenty of future research areas.

7.1.1 Security issue

Deployment of the Docker container in privileged mode opens access to the entire Kernel of the device. Then bind-mount with the file system and system services opens access to the file system of the host devices. A most straightforward way to handle this security issue is by creating a different user with limited privileges for deploying containers, but it blocks the option to install the required packages on the fly while deploying the service.

7.1.2 Connectivities

When a service is deployed behind the NAT, it limits the interoperability inside each NAT. Communication between two and more NATs can be established through a cloud with message passing. However, this will also add to the delay, which is a conflict of interest in fog computing.

7.1.3 Stateless and stateful applications

In the case studies, we carried out, most of them are stateless applications. However, stateful applications will store some data on the fog nodes or hold any state of an application. Therefore in these cases, it is crucial to add some features to the framework to migrate the application's state with standardization while moving the application from one fog setup to another one.

Bibliography

[1] Hina Afreen and Imran Sarwar Bajwa. An iot-based real-time intelligent monitoring and notification system of cold storage. *IEEE Access*, 9:38236–38253, 2021.

[2] Muhammad Azizi Mohd Ariffin, Muhammad Izzad Ramli, Mohd Nazrul Mohd Amin, Marina Ismail, Zarina Zainol, Nor Diana Ahmad, and Nursuriati Jamil. Automatic climate control for mushroom cultivation using iot approach. In *2020 IEEE 10th International Conference on System Engineering and Technology (ICSET)*, pages 123–128, 2020.

[3] 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*, ICDCN '17, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Tosca: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

[5] Tobias Binz, Gerd Breiter, Frank Leyman, and Thomas Spatzier. Portable cloud services using tosca. *IEEE Internet Computing*, 16(3):80–85, 2012.

[6] Abdur Rahim Biswas and Raffaele Giaffreda. Iot and cloud convergence: Opportunities and challenges. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 375–376, 2014.

[7] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, 3-4:134–155, 2018.

[8] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, page 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Rajkumar Buyya and Satish Narayana Srirama. *Fog and edge computing: principles and paradigms*. John Wiley & Sons, 2019.

[10] Rajkumar Buyya and Satish Narayana Srirama. *Internet of Things (IoT) and New Computing Paradigms*, pages 1–23. 2019.

[11] Giuliano Casale, Matej Artač, W-J Van Den Heuvel, André van Hoorn, Pelle Jakovits, Frank Leymann, Mike Long, Vasilis Papanikolaou, Domenico Presenza, Alessandra Russo, et al. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87, 2020.

[12] Ana C. Franco da Silva, Uwe Breitenbücher, Kálmán Képes, Oliver Kopp, and Frank Leymann. Opentosca for iot: Automating the deployment of iot applications based on the mosquitto message broker. In *Proceedings of the 6th International Conference on the Internet of Things*, IoT'16, page 181–182, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Ana Cristina Franco da Silva, Uwe Breitenbücher, Pascal Hirmer, Kálmán Képes, Oliver Kopp, Frank Leymann, Bernhard Mitschang, and Ronald Steinke. Internet of things out of the box: Using tosca for automating the deployment of iot environments. In *CLOSER*, pages 330–339, 2017.

[14] Rustem Dautov, Hui Song, and Nicolas Ferry. Towards a sustainable iot with last-mile software deployment. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2021.

[15] Gianluca Davoli, Davide Borsatti, Daniele Tarchi, and Walter Cerroni. Forch: An orchestrator for fog computing service deployment. In *2020 IFIP Networking Conference (Networking)*, pages 677–678, 2020.

[16] Chinmaya Kumar Dehury, Pelle Jakovits, Satish Narayana Srirama, Giorgos Giotis, and Gaurav Garg. Toscadata: Modeling data pipeline applications in tosca. *Journal of Systems and Software*, 186:111164, 2022.

[17] Jerker Delsing, Jens Eliasson, Jan van Deventer, Hasan Derhamy, and Pal Varga. Enabling iot automation using local clouds. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 502–507, 2016.

[18] Bruno Donassolo, Ilhem Fajjari, Arnaud Legrand, and Panayotis Mertikopoulos. Fog based framework for iot service provisioning. In *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6, 2019.

[19] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. Genesis: Continuous orchestration and deployment of smart iot systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875, 2019.

[20] Nicolas Ferry, Phu H Nguyen, Hui Song, Erkuden Rios, Eider Iturbe, Satur Martinez, Angel Rego, et al. Continuous deployment of trustworthy smart iot systems. *The Journal of Object Technology*, 2020.

[21] Habeeb Ahmed Hassan and Rawaa Putros Qasha. Toward the generation of deployable distributed IoT system on the cloud. *IOP Conference Series: Materials Science and Engineering*, 1088(1):012078, feb 2021.

[22] Saiful Hoque, Mathias Santos De Brito, Alexander Willner, Oliver Keil, and Thomas Magedanz. Towards container orchestration in fog computing infrastructures. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299, 2017.

[23] Paridhika Kayal. Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope : Invited paper. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, pages 1–6, 2020.

[24] Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Towards automated iot application deployment by a cloud-based approach. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 61–68, 2013.

[25] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[26] M. Muñoz, J. L. Guzmán, J. A. Sánchez-Molina, F. Rodríguez, M. Torres, and M. Berenguel. A new iot-based platform for greenhouse crop production. *IEEE Internet of Things Journal*, 9(9):6325–6334, 2022.

[27] Justice Opara-Martins, Reza Sahandi, and Feng Tian. Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective. *J. Cloud Comput.*, 5(1), dec 2016.

[28] Rakiba Rayhana, Gaozhi Xiao, and Zheng Liu. Internet of things empowered smart greenhouse farming. *IEEE Journal of Radio Frequency Identification*, 4(3):195–211, 2020.

[29] Hani Sami and Azzam Mourad. Dynamic on-demand fog formation offering on-the-fly iot service deployment. *IEEE Transactions on Network and Service Management*, 17(2):1026–1039, 2020.

[30] Hani Sami, Azzam Mourad, and Wassim El-Hajj. Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services. *IEEE/ACM Transactions on Networking*, 28(2):778–790, 2020.

[31] Haleema Essa Solayman and Rawaa Putros Qasha. Portable modeling for icu iot-based application using tosca on the edge and cloud. In *2022 International Conference on Computer Science and Software Engineering (CSASE)*, pages 301–305, 2022.

[32] Hui Song, Rustem Dautov, Nicolas Ferry, Arnor Solberg, and Franck Fleurey. Model-based fleet deployment in the iot–edge–cloud continuum. *Software and Systems Modeling*, pages 1–26, 2022.

[33] Ahmad F. Subahi and Kheir Eddine Bouazza. An intelligent iot-based system design for controlling and monitoring greenhouse temperature. *IEEE Access*, 8:125488–125500, 2020.

[34] Orazio Tomarchio, Domenico Calcaterra, Giuseppe Di Modica, and Pietro Mazzaglia. Torch: a tosca-based orchestrator of multi-cloud containerised applications. *Journal of Grid Computing*, 19(1):1–25, 2021.

[35] Andreas Tsagkaropoulos, Yiannis Verginadis, Maxime Compastié, Dimitris Apostolou, and Gregoris Mentzas. Extending tosca for edge and fog deployment support. *Electronics*, 10(6), 2021.

[36] Luis M. Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, oct 2014.

[37] Salvatore Venticinque and Alba Amato. A methodology for deployment of iot application in fog. *Journal of Ambient Intelligence and Humanized Computing*, 10(5):1955–1976, 2019.