****

DEPLOYING MICROSERVICES FOR IOT USING DOCKER AND EDGE COMPUTING

**Graduation Project**

**DEPLOYING MICROSERVICES FOR IOT USING DOCKER AND EDGE COMPUTING**

**Submitted By**

**OLCAY DAĞIDIR 1700005401**

**SADIK DORUK ALİOĞLU 1501020032**

Project Advisor

Assoc. Prof. Dr. Akhan AKBULUT

Department of Computer Engineering

İstanbul Kültür University

2020-Spring

****

May

2020

**Graduation Project**

**DEPLOYING MICROSERVICES FOR IOT USING DOCKER AND EDGE COMPUTING**

**Submitted By**

**OLCAY DAĞIDIR 1700005401**

**SADIK DORUK ALİOĞLU 1501020032**

Advisor

Assoc. Prof. Dr. Akhan AKBULUT

THE EXAMINATION COMMITTEE

Jury Member Signature

1. Assoc. Prof. Dr. Akhan AKBULUT ……………………….

# ABSTRACT

Single-board computers play an important role in the IoT in today's world. Different software is used in the preparation of these cards. In this project, the Docker platform with virtualization technology infrastructure is used to solve problems such as single-board computers that do not have the remote version update feature. Docker is a virtualization platform for programmers and developers. With Docker, virtual containers can be run on Windows, Linux, macOS X operating systems. Running, updating and testing of programs can be done quickly in the Docker system. In this study, different usage methods of the Armbian Debian operating system and Docker architecture are tested in Orange Pi Zero, which is a single-board computer. Seven different sensors are connected to the Orange Pi Zero and the codes written for the sensors are used in Docker containers. As a result of the tests, it is seen which type of usage provides more performance and advantage. As a Method A, sensor codes are operated directly on the Orange Pi Zero device. In Method B, separate containers are operated using a separate Docker image for each sensor. In the last method, all sensor codes are collected in a single Docker image, only that container is operated and performance tests are compared.

# ÖZET

Tek kartlı bilgisayarlar bugünün dünyasında Nesnelerin İnterneti'nde önemli bir rol oynamaktadır. Bu kartların hazırlanmasında farklı yazılımlar kullanılmaktadır. Bu projede, sanallaştırma teknolojisi altyapısı olan Docker platformu, uzaktan sürüm güncellemesi özelliği olmayan tek kartlı bilgisayarlar gibi sorunları çözmek için kullanılmaktadır. Docker, programcılar ve geliştiriciler için bir sanallaştırma platformudur. Docker ile sanal kapsayıcılar Windows, Linux, macOS X işletim sistemlerinde çalıştırılabilir. Programların çalıştırılması, güncellenmesi ve test edilmesi Docker sisteminde hızlı bir şekilde yapılabilmektedir. Bu çalışmada, tek kartlı bir bilgisayar olan Orange Pi Zero'da Armbian Debian işletim sistemi ile Docker mimarisinin farklı kullanım yöntemleri test edilmektedir. Orange Pi Zero cihazına yedi farklı sensör bağlanılmakta ve sensörler için yazılan kodlar Docker kaplarında kullanılmaktadır. Testler sonucunda hangi kullanım türünün daha fazla performans ve avantaj sağladığı görülmektedir. İlk yöntem olarak sensör kodları doğrudan Orange Pi Zero cihazında çalıştırılmaktadır. İkinci yöntemde, ayrı kaplar her sensör için ayrı bir Docker görüntüsü kullanılarak çalıştırılmaktadır. Son yöntemde, tüm sensör kodları tek bir Docker görüntüsünde toplanıp sadece o konteyner çalıştırılmakta ve performans testleri karşılaştırılmaktadır.

# ACKNOWLEDGEMENTS

First, we would like to thank Prof. Akhan AKBULUT, our project consultant, for all his support and for transferring his knowledge to us to complete the project. We would like to thank all our lecturers and teaching assistants who brought us to this level for us to do this work. We would like to thank Professor Murat TAYLI, our head of the department and the instructor of Operating Systems, for the information about the virtualization technology he gave in the course of Operating Systems. Finally, we are very grateful to our families for their beliefs and support.

# TABLE OF CONTENTS

[ABSTRACT III](#_Toc42612528)

[ÖZET IV](#_Toc42612529)

[ACKNOWLEDGEMENTS V](#_Toc42612530)

[TABLE OF CONTENTS VI](#_Toc42612531)

[LIST OF TABLES VIII](#_Toc42612532)

[LIST OF FIGURES I](#_Toc42612533)

[SYMBOLS & ABBREVIATIONS II](#_Toc42612534)

[1. INTRODUCTION 1](#_Toc42612535)

[1.1. Problem Statement 2](#_Toc42612536)

[1.2. Project Purpose 2](#_Toc42612537)

[1.3. Project Scope 2](#_Toc42612538)

[1.4. Objectives and Success Criteria of the Project 3](#_Toc42612539)

[1.5. Report Outline 3](#_Toc42612540)

[2. RELATED WORK 4](#_Toc42612541)

[2.1. Existing Systems 11](#_Toc42612542)

[2.2. Overall Problems of Existing Systems 11](#_Toc42612543)

[2.3. Comparison Between Existing and Proposed Method 11](#_Toc42612544)

[3. METHODOLOGY 13](#_Toc42612545)

[3.1. Design Overview 13](#_Toc42612546)

[3.1.1. Orange Pi 13](#_Toc42612547)

[3.1.2. Types of Sensors 15](#_Toc42612548)

[3.1.2.1. Flame Sensor 15](#_Toc42612549)

[3.1.2.2. Gas Detection Sensor 15](#_Toc42612550)

[3.1.2.3. Rain Sensor 15](#_Toc42612551)

[3.1.2.4. Motion Sensor 16](#_Toc42612552)

[3.1.2.5. Temperature Sensor 16](#_Toc42612553)

[3.1.2.6. Humidity Sensor 17](#_Toc42612554)

[3.1.2.7. Buzzer Sensor 17](#_Toc42612555)

[3.2. System Architecture 18](#_Toc42612556)

[3.2.1. Traditional Usage 18](#_Toc42612557)

[3.2.2. Individual Container Usage 19](#_Toc42612558)

[3.2.3. All-in-One Container 21](#_Toc42612559)

[3.3. System Software 24](#_Toc42612560)

[3.3.1. Docker 24](#_Toc42612561)

[3.3.2. Debian 25](#_Toc42612562)

[3.3.3. Python 26](#_Toc42612563)

[3.3.4. Putty 26](#_Toc42612564)

[4. EXPERIMENTAL RESULTS 29](#_Toc42612565)

[4.1. CPU Utilization 30](#_Toc42612566)

[4.2. CPU Speed 31](#_Toc42612567)

[4.3. Network Bandwidth 32](#_Toc42612568)

[4.4. CPU Temperature 32](#_Toc42612569)

[5. DISCUSSION 34](#_Toc42612570)

[6. CONCLUSIONS 38](#_Toc42612571)

[REFERENCES 39](#_Toc42612572)

[APPENDIX 41](#_Toc42612573)

[APPENDIX A – Method A 41](#_Toc42612574)

[APPENDIX B – Method B 54](#_Toc42612575)

[APPENDIX C – Method C 70](#_Toc42612576)

# LIST OF TABLES

[Table 2. 1: Method Comparison 12](#_Toc42607240)

[Table 3. 1: Comparison of Orange Pi Models 14](#_Toc42607416)

[Table 5. 1: Advantages and Disadvantages of Methods 33](#_Toc42607435)

# LIST OF FIGURES

[Figure 2. 1: Distributed IoTaaS Example [11] 6](#_Toc42608937)

[Figure 3. 1: Orange Pi Zero 13](#_Toc42608945)

[Figure 3. 2: Flame Sensor 15](#_Toc42608946)

[Figure 3. 3: Rain Sensor 16](#_Toc42608947)

[Figure 3. 4: Motion Sensor 16](#_Toc42608948)

[Figure 3. 5: Temperature Sensor 17](#_Toc42608949)

[Figure 3. 6: Buzzer Sensor 17](#_Toc42608950)

[Figure 3. 7: Architecture of Method A 19](#_Toc42608951)

[Figure 3. 8: Example of Conceptual Design of Docker over Raspberry Pi 20](#_Toc42608952)

[Figure 3. 9: Architecture of Method B 21](#_Toc42608953)

[Figure 3. 10: Conceptual design of Docker [21] 22](#_Toc42608954)

[Figure 3. 11: Architecture of Method C 23](#_Toc42608955)

[Figure 3. 12: Docker Infrastructure 24](#_Toc42608956)

[Figure 3. 13: Putty 27](#_Toc42608957)

[Figure 3. 14: Connection over Putty 28](#_Toc42608958)

[Figure 4. 1: Example of -top command 29](#_Toc42608959)

[Figure 4. 2: Example of armbianmotinor 30](#_Toc42608960)

[Figure 4. 3: Comparison with the proposed methods (CPU Utilization) 31](#_Toc42608961)

[Figure 4. 4: Comparison with the proposed methods (CPU Speed MHz) 31](#_Toc42608962)

[Figure 4. 5: Overall Network Bandwidth (Kbps) 32](#_Toc42608963)

[Figure 4. 6: CPU Temperature (°C) 33](#_Toc42608964)

# SYMBOLS & ABBREVIATIONS

VM: Virtual Machine

IoT: Internet of Things

CPU: Central Processing Unit

PC: Personal Computer

TCP: Transfer Control Protocol

IP: Internet Protocol

RAM: Random Access Memory

DPWS: Devices Profile for Web Services

REST: Representational State Transfer

FEFO: First Expired First Out

MCI: Mild Cognitive Impairment

UHF: Ultra High Frequency

IoTaaS: IoT as a Service

PaaS: Platform as a Service

IT: Information Technology

API: Application programming interface

SBC: Single Board Computer

UDP: User Datagram Protocol

CoAP: Constrained Application Protocol

MQTT: Message Queuing Telemetry Transport

SDN: Software-Defined Network

TCP: Transfer Control Protocol

HTTP: Hypertext Transfer Protocol

VPN: Virtual Private Network

SoC: System on a Chip

GPIO: General Purpose Input Output

SSH: Secure Shell

WOT : Web of Things

MCI: Multiplicative Competitive Interaction

ARM: Advanced Reduced Instruction Set Computing Machine

# INTRODUCTION

Virtualization technology, which has been popular in recent years and used mostly on servers, has many disadvantages. Therefore, Docker technology [1], a new type of virtualization, helps solve many problems such as a bunch of people working on the same application. Docker is an open-source virtualization platform developed for software developers and systematics. With Docker, you can run Linux and Windows virtual containers on Linux, Windows, and MacOSX. Docker is built on the LXC virtualization mechanism. A Docker image is operated in units called containers. Each container uses a process. Depending on their power, thousands of Docker containers can work on one machine. Container images share common system files. Therefore, disk space is saved. Docker takes the image of the installed final versions of the software and makes them available again. You can create these images once and send them to other servers. Also, you can create different images for each server. Every server can remake the image by looking at the instruction document called Dockerfile. In this way, no manual intervention is required. VMs have a full operating system for each employee instance. Docker, on the other hand, uses images that are reduced in size instead of the full operating system and uses host operating system libraries as shared. However, this makes Docker the system resource-friendly and lowers the isolation level. Both virtualization approaches have advantages and disadvantages compared to each other. However, it would be correct to say that some advantages are very critical from the perspective of Docker.

One of the advantages of Docker is easily operable. Docker Hypervisor runs in seconds because it does not use a fully loaded operating system and the host is running close to the system. Another is its predisposition to versioning. One of the most striking features of Docker is its versioning feature. Docker allows us to record different versions of the operating system images it uses. This situation opens the door for sharing the prepared images among users. If another feature is talked about, this is shareability. Operating system images prepared by users or distributors can be sent to central servers and can also be obtained by other users from these central servers. Working on a single operating system brings the problem of security to mind. Docker brought software solutions to this issue. Applications running in the container cannot see and affect the application in another container unless stated otherwise, in other words, they are isolated. Orange Pi is a new generation minicomputer with leading features. Orange Pi is the number one choice of developers with its unrivaled features that can be accessed all over the world with catch-phrase of “new generation mini PC”. Orange Pi minicomputer has hardware and software support that can be used from prototype to production. You can develop large projects and use them commercially with Orange Pi. Orange Pi has extensive equipment support compatible with your projects. Orange Pi is the leading developer mini developer computer for developers. IoT is the most important feature that enables it to be a developer product and to be in an assertive position in this field.

## Problem Statement

To widely use smart developer cards such as Orange Pi and Raspberry Pi, it is necessary to connect to the device each time to update the code blocks or change some settings. In this study, this is called the traditional method. For such cases, version update or code update can’t possible because you need to connect the device with SSH in the same network. Therefore, to solve this problem, you just need to connect to the device once during the first installation and prepare for the Docker container. This was done to ensure that the version is automatically updated without connecting directly to the device in every update. Version updates can be performed easily thanks to the microservice that can be created with Docker.

## Project Purpose

The purpose of this project is microservices, utilize IoT devices in terms of management and security for various fields. Having light supplies services instead of embedded software provides energy efficiency and stability. In this project, it is aimed to use internal services as a microservice to run an IoT device sensors to increase its operational performance. The focus of the project is to deal with interoperability challenges between devices and IoT device systems.

## Project Scope

The scope of this project is to make a study comparing these methods which are Traditional Method as Method A, Individual Method as Method B, and All in One Method as Method C, by using three different development environments in Orange Pi Zero device. The tools used in this project are as follows.

* Docker platform
* Orange Pi Zero
* Flame sensor
* Gas sensor
* Rain sensor
* Motion sensor
* Temperature sensor
* Humidity sensor
* Buzzer sensor

## Objectives and Success Criteria of the Project

The project aims to show the advantages and disadvantages of Docker container usage on Orange Pi. It is examined if there is a decrease in device performance by using the remote version update method of IoT device with Docker container. The success criteria are as follows.

* Providing high performance and ease of use for developers and users with container usage.
* Developing the traditional method by using the advantages of remote control and version update of the container structure.

## Report Outline

A summary of this study, when an IoT device was designed, the ability to remotely control the updates and changes, and performance comparisons with other uses were made after the code and application setups were made to the device via the docker only after connecting to the device once.

The following topics are defined in this report.

* Installation of the Armbian Debian operating system and its necessary environments.
* Installation of the virtualization technology of Docker Container.
* Configuration of Dockerfile and how to use it.

# RELATED WORK

Different architectures need to be used to transition from the IoT to built-in systems to a low power network. For this, cloud-based restful architecture should be used. With the advent of light TCP / IP packets with the web, files on the web can be directly accessed [2]. The web vision of objects is simple and recommends the use of well-defined restful interfaces. The study offers a system that includes restricted devices that directly runs a web server. The basis of all device code is an embedded operating system that can be used for resource-constrained platforms. An example of this operating system is Contiki3[3]. When the software needs to be developed, the developers continue by making changes to the network. These can be full image change on the network, incremental update, or dynamic link. Mate[4] is one of the first VMs for sensor networks. It is specific to the mobile coded area for a specific VM. General-purpose virtual machines were developed for resource-constrained platforms. WSN is particularly suitable for programming as the functionality of each node in the network is based on a simple set of actions: periodic detection, alarm triggering, and execution. This approach is application-independent, focused, and interactive functionality as testing increases productivity by making it easy. DPWS[5] can add standards to low-end devices for interoperability. This is a standard Web application standard. Web resources can provide functionality regardless of certain applications. It does this using REST's uniform interfaces. You can provide the sensor to receive value or a post-run task. Thanks to this, the devices do not need to be reprogrammed. Also, multiple applications can benefit from devices at the same time. The application server extends the WOT approach with Actinium[6]. Not only the relaxing interfaces of the devices but also the runtime container is RESTful. This advocates a true end-to-end RESTful approach.

One-third of the products produced for the consumption of people is wasted. There is the FEFO rule in the food supply chain. In this rule, the first to get on the shelf first comes out. It gives information about the deterioration time by following the temperatures during the transportation of food or during the shelf [7]. Thus, losses can be reduced with this system. As a solution to this project, it is a system that provides information about the shelf life of the products by using the smart container by integrating the internet of objects. Sensor nodes in the container collect information in the environment and calculate the life of products that can be destroyed by the battery-powered computer in the container. It starts by sending a Node ID. Sensor values are packed by MCI. Then, UHF Receiver-Transmitter is sent.

Hyper-manager virtualization technology is the process of hosting one or more machines through a single physical machine. The basis of cloud computing is virtualization. Creates an abstraction layer, hiding basic hardware and software complexity. Thus, cloud providers can adapt to flexibility, scalability, reliability, isolation, resource optimization, etc. can be trusted. So, it can offer guaranteed services. The Linux Container Virtualization concept for IoT makes sense. This concept is light virtualization that allows different applications to run. There are two types of containers. These are applications and system containers. Only one application runs on the container. In the system containers, an example of a user area is booted. IoT Clouds are not dependent [8]. IoT devices interact with a far Cloud system, which is responsible for collecting and evenly detecting data from heterogeneous IoT devices. Usually, IoT devices run customized software written with a specific programming language and development framework. IoT devices can perform minimum processing and storage jobs. However, it is necessary to flexibly measure/reduce processing and storage capabilities. That's why big data processing tasks using virtualization technologies are performed in the Cloud system. According to the multiple clouds or cloudy burst model, IoT Cloud 1 creates its own IoTaaS 1, which takes advantage of the IoTaaS 2-4 presented by IoT Cloud 2-4, it is shown in Figure 2.1 as an example. It is very similar to the model of PaaS and SaaS business models adopted by this. Dropbox built its business using IaaS provided by Amazon [9]. That means the system was distributed to Amazon's VMs. In this scenario, we emphasize that there is no low-level interaction between Amazon and Dropbox. Private IoT clouds host various virtual machines and containers to provide virtual services to their customers. These are their virtualization infrastructure. The "cross-cloud federation" scenario can increment the number of resources each particle Cloud operator requires more storage, computing, detection, and activation virtualization. The term "federation" refers to a system organization characterized by a merger. In this federation, the IoT Cloud federation is defined as a network of interconnected IoT Cloud providers to enable activation. An environment where everything is everywhere, driven by restrictions and agreements on infrastructure. Consider IoTaaS [10], which combines containers built from different federated clouds with different IoT devices. In this respect, it is possible to apply location-sensitive software strategies. Despite obvious advantages, IoT clouds are more complex than traditional systems. The implementation of the IoT Cloud Federation scenario is not trivial, as current federation models are not considered appropriate.

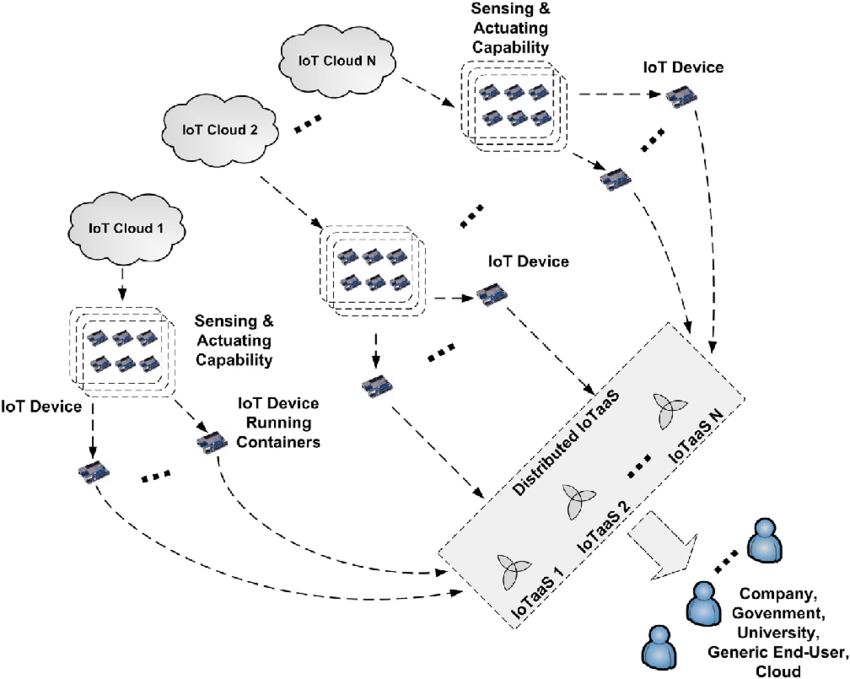


Figure 2. 1: Distributed IoTaaS Example [11]

Cloud computing and the internet of objects cause changes in information and communication systems. In general, the number of IoT devices was determined by a new smart distributed system. For example, Linux-based operating system, remote cloud infrastructure, platform, or single-board computers to show the object in the cloud. In this context, container-based virtualization technology is a lightweight alternative to the hypervisor-based approach that can be adopted on smart objects to improve cloud service delivery. It allows IoT Cloud providers to deploy and customize smart objects on a flexible piece of software. This paragraph explores IoT Cloud scenarios that analyze container-based virtualization, its advantages, and its performance on smart objects.

Cloud computing encouraged the use of outsourcing, paving the way for the reduction of maintenance costs of computing infrastructures. Container technology has come as an alternative solution to virtual machines today. Although successful, cloud computing remains insufficient for the IoT. Applications running on the VM are heavy, one of them is boot time [12]. For this reason, the use of virtualization technology and docker has gained popularity. When the containers became known as light virtualization, they started to be used by many IT companies. Google Kubernetes, Apache Marathon1, Docker Swarm2 are open-source container tools used in recent years. It introduced some limitations, such as Cloud Computing, which concluded the proposal of technologies such as Fog and Edge Computing, which ensure the advantages of a virtualized circle for IoT applications. The container is more suitable for fog computing, which consists of less resource-constrained nodes because it is lighter virtualization than the VM. When using IoT applications, the docker evaluates the effects on the container performances. IoT needs fog computing because fog computing transfers the data to nearby devices and the data travels less than the cloud, thereby reducing latency. Kubernetes is another open-source Container Editing tool to automate the distribution, scaling, and management of Container-based applications. Three requirements have been identified, these are in a cluster fog infrastructure, a fog node, and Orange Pi. If a fog system is used in a factory, if a node is connected with a manufacturing machine, the data from the machine must work on the fog node to get better efficiency with each other. Therefore, there are scenarios in which a Sis node must be physically connected to devices and machines to collect data. Examples of serial ports are so that applications must have access to resources other than CPU, memory, and storage. A use case can be added to a ZigBee-Coordinator (such as Xbeestick4) to the serial USB port of a fog node to collect data from ZigBee sensors [13]. Relevant Containerized ZigBee applications must access the USB port of the fog node to which the coordinator is connected. If a Docker Container runs this application, the fog node's USB port should have the option to provide access to the Container. OpenMTC was used for performance evaluation and worked through two scenarios. It was run without a container in Scenario 1. How long access to the device was measured. In Scenario two, the application was run in a dockworker container and measurements were made. It was observed that Docker containers added an average of approximately 0.036 cargoes. Besides, 95% confidence interval was seen for all scenarios. The Linux "stress" test tool was used, where-c, - m, - io are CPU, memory, I / O workloads respectively. Again, each scenario was repeated 20 times to get the measurements accurate. It is important to note that this delay is only observed when the device is first accessed. Marathon is a container editing tool that works in the Mesos cluster. The marathon depends on Mesos and ZooKeeper for clustering. To join the ZooKeeper cluster, the Mesos on each host must be set up separately.

Cloud services such as load balancers, message-oriented middleware, NoSQL storage, and streaming data processing frameworks are designed to accept data flow and workloads from the IoT, they lack the capabilities to coordinate with IoT processing decisively. For example, such cloud services monitor static data loaded from IoT and adjust performance behaviors, but rarely transfer it back to IoT elements to maintain the load generated by IoT. Dustdar deploys such a service with preconfigured sensor data that read speeds and activates only a subset of sensors in the IoT section, which forms infrastructures and IoT elements alongside objects, on the edge of the cloud but not in the Data Center [14]. The enabled sensors send tracking data to cloud infrastructure services that process flow data to control behavior data. To achieve high accuracy analysis from the cloud, Dustdar activates more sensors and instantly increases the reading speeds of sensors. However, if its action is not coordinated with the control of cloud services, these services may not react quickly to deal with the sudden volume of incoming data. Network protocols, communication protocols, interfaces, etc. Together with cloud network hardware that acts as a software-defined router, which can be programmed to control.

Container-based virtualization can be implemented to consider a slight alternative to hypervisor-based virtualization. Containers isolate processes at the operating system level of the host, thus avoiding overhead due to virtualized hardware and virtual device drivers [15]. A container engine can be considered a small, isolated virtual environment with a set of dependencies required to run a particular application. The concept of inclusive is nothing new in the virtualization world, but with the advent of the Docker platform it has recently taken on more relevance and real-world cohesion. Docker offers an underlying container engine, along with a functional API that allows you to easily create, manage, and remove a container application. Due to small overhead production, multiple containers can operate even on devices with limited resource requirements, such as single-board computer platforms. Given the potential benefits that containers bring, Morabito aims to provide a detailed performance analysis by evaluating the use of Docker containers in restricted environments, along with the significant increase in usage cases.

In Single-Board Computers, Morabito tested Raspberry Pi 2 Model B, Raspberry Pi 3, Odroid C1+, Odroid C2, and Odroid XU4 [16]. The most relevant differences between the boards occur in terms of their CPU, flash storage, and Ethernet capabilities. From a software perspective, for the Raspberry Pi an image provided by Hypriot running Raspbian Jessie with the Linux kernel 4.4.10 used as a base OS. This image provides a lightweight environment optimized for executing Docker container technologies on top of Raspberry Pi devices. For the Odroid platforms: Ubuntu version 14.04 for OC1+, Ubuntu version 16.04 for OC2, and Ubuntu 15.10 for OXU4 used. The power consumption of the SBCs was measured using an external voltage meter characterized by a 16-bit resolution used. From a power consumption perspective Morabito found out that the consumption of the Raspberry Pi boards, OC1+, and OC2 vary, ranging from 1.88W (RPi2) to 3.26W (OC2). This variation range highlights the high energy efficiency of these devices. As Morabito's results, some tests were conducted on the hardware. For the CPU, there is a proven current difference between the local enclosure and the Docker enclosure. For memory and Input / Output testing, Morabito used the Unix command mbw5 to test memory performance, which determines the available memory bandwidth by copying large data sequences in memory and performing three different tests. Similarly, for CPU state, natural and cap performance is comparable with a maximum margin of 6.04 percent during memory testing. On the other hand, for Disk Input / Output, Morabito is the sysbench used to evaluate disk performance. The comparison is set to perform a random read/write operation. The result of this assessment shows a decrease in Docker's performance compared to the local situation. This difference is roughly around 10%.

On the software side, MySQL and Apache2 were tested [17]. There is no significant difference in performance between the native state and the Docker configuration, especially for MySQL, for a large number of concurrent threads. Besides, the Docker produces lower power consumption than the local enclosure when operating the Apache2.

As a result, Raspberry Pi cards were found to be highly efficient when handling low volume network traffic, especially UDP traffic. This result can be particularly useful when creating effective IoT gateway designs specifically designed for the execution of lightweight IoT applications such as restricted application protocol CoAP and MQTT protocols [18]. One of the main concerns was the lack of namespace isolation, which made placed applications more vulnerable. The most recently released versions of Docker include a variety of security enhancements to deal with these issues. Further efforts to provide better security for systems with Docker, Docker with the Center for internet safety issues that are represented in the collaboration between the various security benchmark, which is a developer tool that can control security led to the launch of Docker within virtualized applications.

The use of virtualization technologies for different topics such as cloud environments, the IoT, and SDN has increased rapidly in recent years. Container-based solutions for deploying lightweight applications have their features surrounded by these virtualization technologies. Containers implement isolation of operations at the operating system level to avoid overhead due to virtualized hardware and virtual drives. In this case, a container is known as an isolated and small virtual environment that needs several dependents to run a particular application. The concept of containerization is nothing new in the virtual world, but with the arrival of Docker, it has gained more attention and Real-World Acceptance. Docker represents an underlying container engine with its functional API that allows creating, managing, and removing container applications. Previously, multiple containers could operate even on devices with limited resources, such as SBC platforms, due to small overhead production. Given the benefits of containers, Morabito uses the Raspberry Pi 2 device as an SBC. Morabito used the Docker version 1.8.0 and the Raspbian Jessie operating system with Linux 3.18.11.

To test the environment, Morabito starts with the Synthetic Benchmark test by using sysbench respectively CPU, Memory I/O, Disk I/O, and finally Network I/O. The author proved that an existing difference between native platform and the Docker platform cases. However, Docker has 2.67% more CPU performance. For the results of Memory I/O, native and container performances are similar, the maximum percentage difference is 6.04%. To test Disk I/O, again sysbench used. The result of the test shows that Docker has a loss of performance with 10% difference. For the Network I/O, when tracking UDP and TCP network traffic, packet loss and power consumption mismatch are not observable. On the Application Benchmark, MySQL and Apache2 tested. The result of the tests, when the real database used, Docker, and native case configurations are not much different. When using Apache2, tested by using the Apache HTTP server benchmarking tool, the Docker platform produces less power consumption.

## Existing Systems

The following platforms are based on Single Board Computer infrastructure.

* Raspberry Pi
* Orange Pi
* Banana Pi
* ODROID

## Overall Problems of Existing Systems

The biggest problem when using Single Board Computer is that the developer or user should be on the same network to update the version. This situation leads to limited solutions and time lost during the problem. Also, VPN or network specifications must be configured to connect to the private network.

## Comparison Between Existing and Proposed Method

This section compares Method A, Method B, and Method C. The comparison is as seen in Table 2.3.1.

Table 2. 1: Method Comparison

| Method A  Traditional | Method B  Individual Container for Each Sensor | Method C  All in One |
| --- | --- | --- |
| Requires being on the same network to be managed | The environment is independent and can be managed remotely | The environment is independent and can be managed remotely |
| Performance change is not observed | Performance change can be observed | Performance change can be observed |
| It does not support the operation of different code files at the same time. | Any code file can be updated and changed without affecting other files | It does not support the operation of different code files at the same time |

Table 2.3.1 shows the comparison of Method A, Method B, and Method C.

# [METHODOLOGY](#_Toc470871184)

This section includes the container structure created on the IoT created using the advantages of virtualization technology and the feasibility structure of the version update.

## Design Overview

### Orange Pi

Orange Pi is the usage of the Advanced Reduced Instruction Set Computing Machine (ARM) technology. ARM technology is used at the board which reduces cost, heat, and power consumption. It is power powerful multi middle CPU implemented as System-On-Chip weighing 26gram and operates on 5V, 600mA energy rating [19]. This board is to be had in three models named Zero, PC, Lite.



Figure 3. 1: Orange Pi Zero

The Orange Pi Zero board is the contemporary version amongst them, and it runs on the ARM Cortex-A7 processor with 256MB RAM working at six-hundred MHz frequency. It has an SD card slot, that's used for booting the running structure like Armbian Debian Buster. It has one USB2.zero port to connect with the peripherals like mouse, keyboard, and Wi-Fi adapter, etc., making it a full-sized portable pocket computer. It additionally has an Ethernet port to connect with the network. GPIO ports are used to interface and control the LEDs, switches, sensors, and other devices. It does not have any display ports like HDMI port, all types of video display units like LCD screens, projectors, TVs cannot be connected. In this board, a few additional features like digital camera connector are to be had to interface digicam and an audio jack with using the shield of the Orange Pi Zero. With all these features, Orange Pi isn't just limited to single usage, it could be used in lots of applications.

Table 3. 1: Comparison of Orange Pi Models

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Board | Zero | One | Lite | Pc | Plus |
| SoC Vendor | Allwinner | Allwinner | Allwinner | Allwinner | Allwinner |
| SoC Chip | H2 | H3 | H3 | H3 | H3 |
| SoC Process | 40nm | 40nm | 40nm | 40nm | 40nm |
| CPU Cores | 4 | 4 | 4 | 4 | 4 |
| CPU Design | Cortex-A7 | Cortex-A7 | Cortex-A7 | Cortex-A7 | Cortex-A7 |
| CPU Freq | 1.3GHz | 1.2GHz | 1.2GHz | 1.3GHz | 1.3GHz |
| CPU Instruction | ARMv7 | ARMv7 | ARMv7 | ARMv7 | ARMv7 |
| GPU Vendor | ARM | ARM | ARM | ARM | ARM |
| GPU Design | Mali-400 MP2 | Mali-400 MP2 | Mali-400 MP2 | Mali-400 MP2 | Mali-400 MP2 |
| GPU Freq | 600MHz | 600MHz | 600MHz | 600MHz | 600MHz |
| Memory | 512MB DDR3 | 512 MB DDR3 | 512 MB DDR3 | 1GB DDR3 | 1GB DDR3 |
| Storage Expandable | MicroSD | MicroSD | MicroSD | MicroSD | MicroSD/USB SATA 2.0 |
| Storage Onboard | None | None | None | None | 8GB eMMC |
| USB 2.0 | 1 + 1 OTG | 1 + 1 OTG | 2 + 1 OTG | 3 + 1 OTG | 4 + 1 OTG |
| USB 3.0 | 0 | 0 | 0 | 0 | 0 |
| Ethernet | 100MB | 100MB | 100MB | 100MB | 1Gb Realtek RTL8211E |
| Wireless | XR819, IEEE 802.11 b/g/n | None | 802.11N RTL8189FTV | None | 802.11N RTL8189ETV |
| Bluetooth | None | None | None | None | None |
| IR | Yes | None | Yes | Yes | Yes |
| Microphone | Yes | None | Yes | Yes | Yes |
| HDMI | None | 4K30fps | 4K30fps | 4K30fps | 4K30fps |
| Power | 5V2A | 5V2A | 5V2A | 5V2A | 5V2A |

### Types of Sensors

### Flame Sensor

The flame detector sensor board is a board that used to detect fire. It contains an infrared receiver. It can be used as a flame detection sensor in fire extinguishing robots. Sensitivity can be adjusted with the controller on it. It can give both analog and digital output signals.

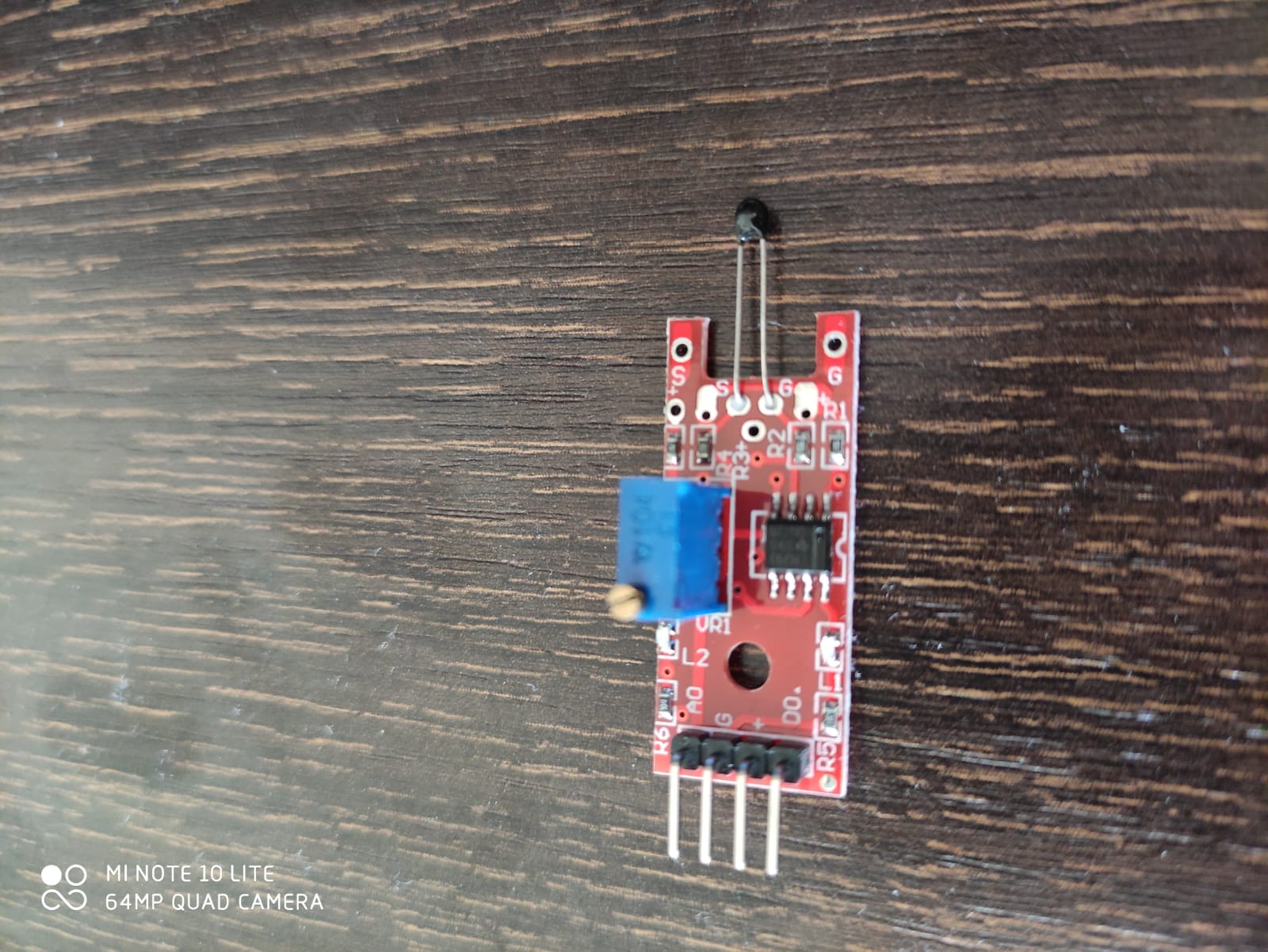


Figure 3. 2: Flame Sensor

### Gas Detection Sensor

The gas detection sensor is a sensor board that detects the presence of gases in the area where it is usually part of the security system. It is used to detect a gas leak or other emissions. It can enable any process to be closed automatically. For this, the sensor interface must be created in connection with the control system.

### Rain Sensor

The rain sensor, which can work with liquids with electrical conductivity, is mostly used to detect rain and water drops. Data are obtained as a result of the lines placed in parallel with the fluids that conduct electricity. Thus, analog output signals are received. It is used with many microcontroller platforms, especially Arduino.

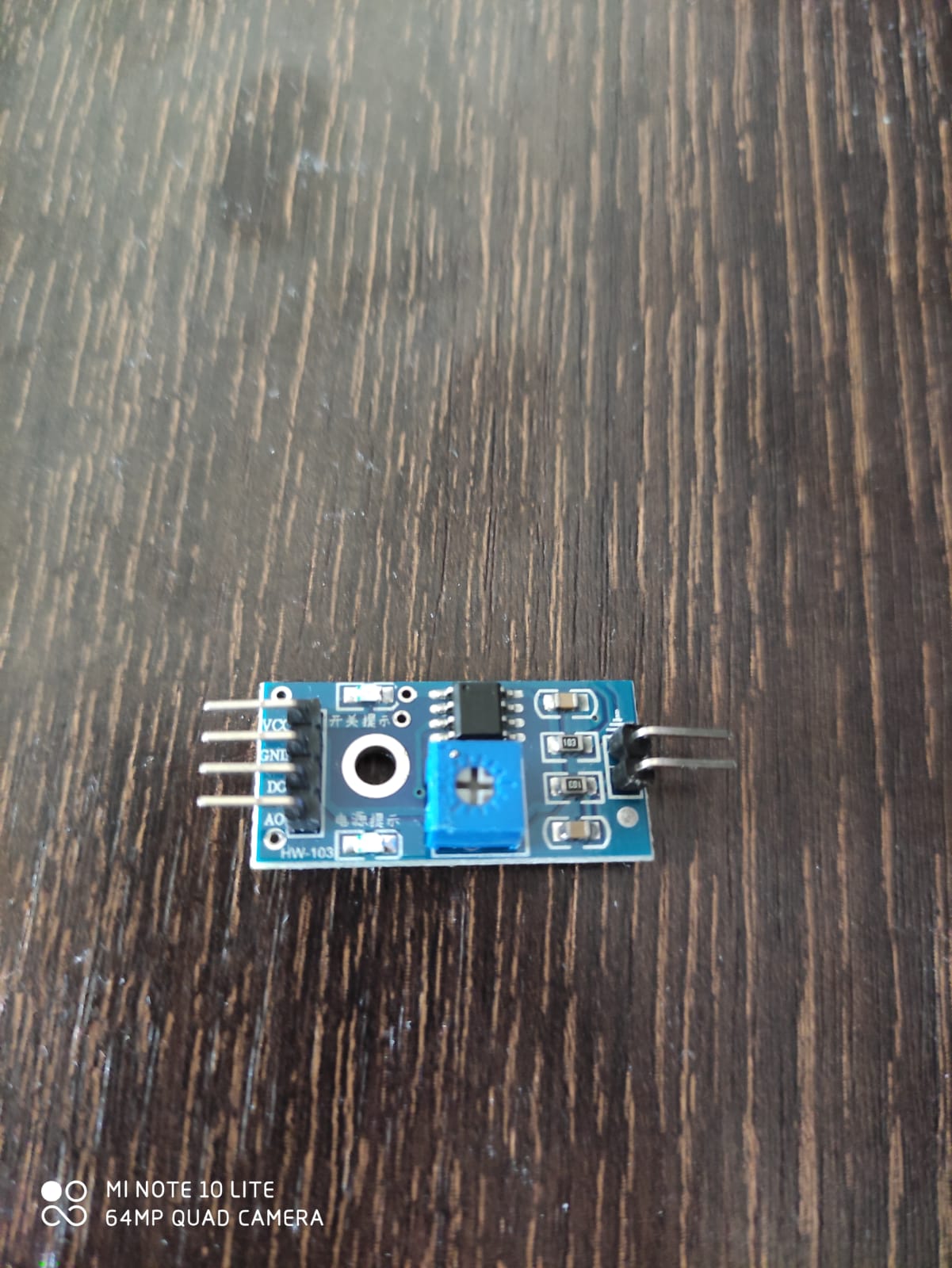


Figure 3. 3: Rain Sensor

### Motion Sensor

The motion sensor can detect motion in indoor and outdoor lighting and security systems by using passive infrared. It is also used in areas such as questioning the presence of materials, detecting objects, motion measurements, numbering objects, and vehicle parking sensors.



Figure 3. 4: Motion Sensor

### Temperature Sensor

The temperature sensor is a sensor board that measures the amount of heat energy or coldness produced by an object or system. It can give both output analog or digital output signals and detect any physical change in the environment. It is used in farming, environment monitoring applications, pandemic cases such as Covid-19, household appliances such as refrigerators and ovens.

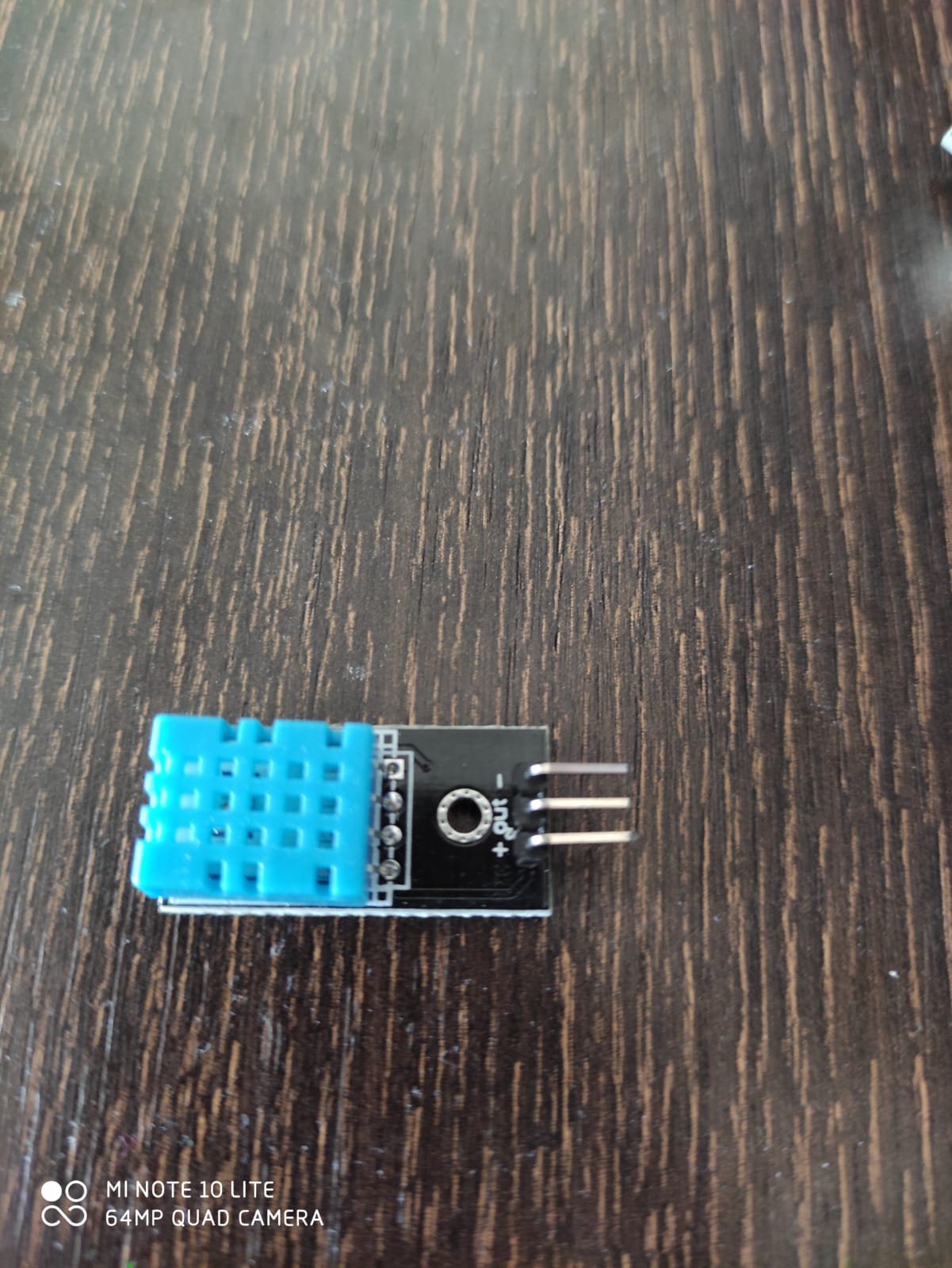


Figure 3. 5: Temperature Sensor

### Humidity Sensor

The amount of water vapor in the air is called humidity. Humidity measurement is used to improve production conditions and increase work efficiency. There are plates which are detecting moisture on it. The spaces between the plates are moistened and used as conductors. It is used in smart agriculture to increase crop quality, in food stores to prevent product degradation, in factories to operate machines efficiently, especially in the health sector.

### Buzzer Sensor

The sensor board called buzzer can be thought of as a small speaker. Although they do not produce loud and detailed sound as speakers, they emit a certain frequency. It is used in household appliances such as home cleaning robots, refrigerators, parking sensors in cars, simple toys, alarm clocks, keypads in computers, and elevators.

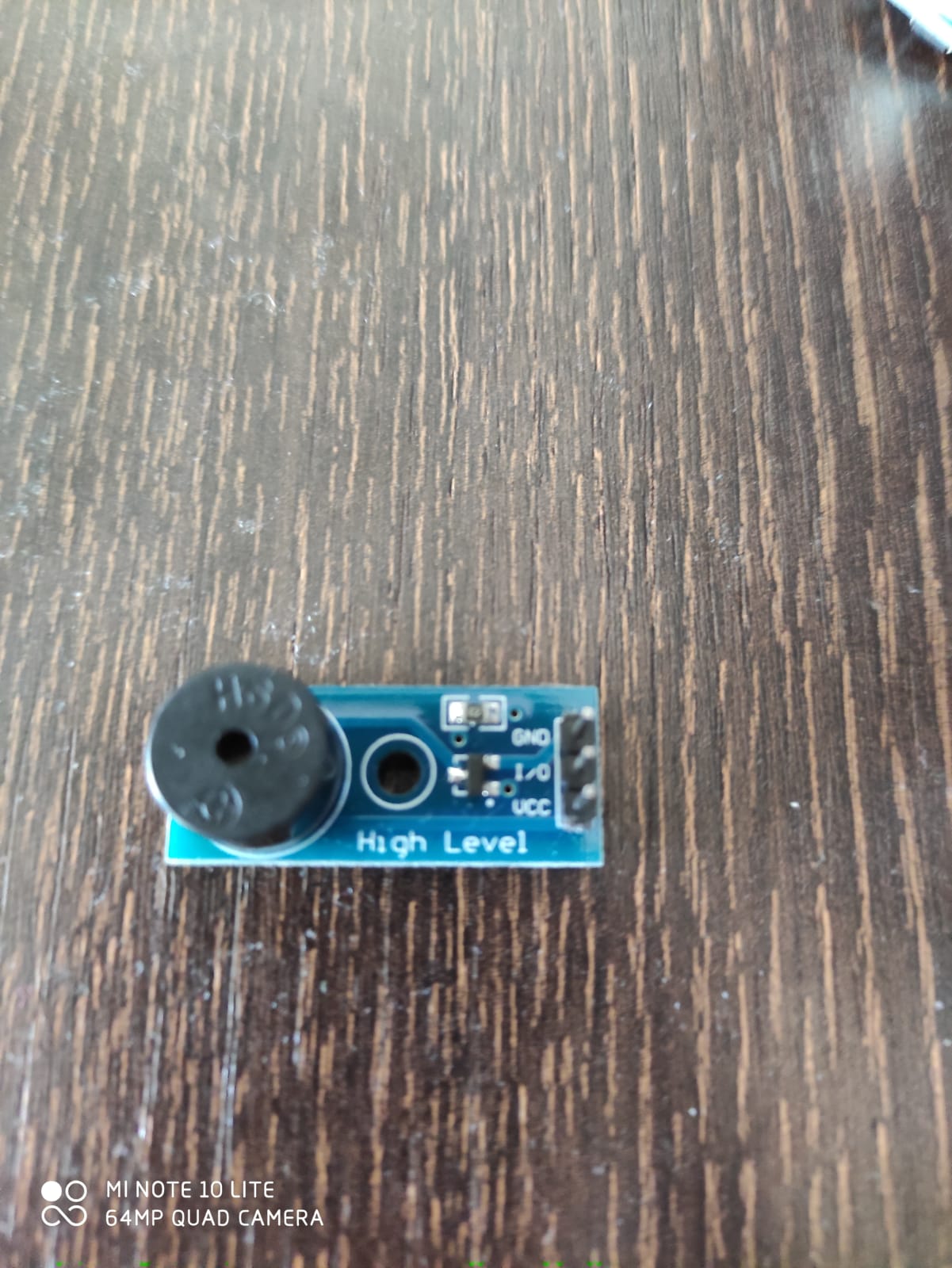


Figure 3. 6: Buzzer Sensor

## 3.2. System Architecture

There are 3 ways to use Orange Pi. These are used by transferring the application to the Orange Pi device, dividing the applications separately, using the container, and running the application in a single container.

### Traditional Usage

The traditional method is the easiest method for new users. It is not much different from using a normal computer. SBC cards such as Orange Pi have been used with the traditional method since the first day of its launch. It is one of the first devices that come to mind in different projects such as media environment, games, robot control, camera, web server, automation, a smart assistant that need GPIO structure. Having GPIO pins plays an important role in exchanging data and making the data meaningful. It is used even in artificial intelligence implementations with powerful hardware working on it. It is more advantageous than developer cards such as Arduino [20] thanks to its operating system. Arduino is a physical programming platform that contains an Input / Output board and the application of the processing language. Arduino can be used to create interactive objects that work alone, as well as software that runs on a computer. Arduino is an electronic development board that belongs to the open-source family with easy-to-use and flexible software/hardware architecture. In this use, which is called the traditional method in the project, it is necessary to connect the device by the user or developer for each operation to be performed. To connect to the device, it is necessary to be on the same network or to use specially configured network structures. This method can be considered as using the same computer for operations such as creating, editing, and deleting Text files on the computer. Traditional use is the first choice for developers using IoT devices. In this method, the developer and user are directly connected to the device. ARM-based operating system installation is required for the installation of the device. After the installation, Python and libraries must be loaded to operate the sensors.

The architecture of Method A as shown in Figure 3.7.

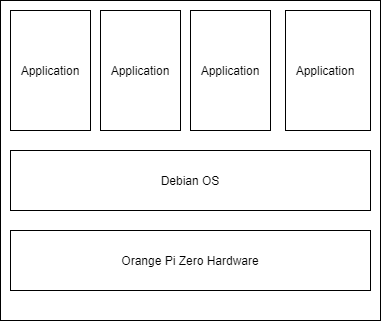


Figure 3. 7: Architecture of Method A

### Individual Container Usage

There are advantages and disadvantages to using a special Docker container for each sensor code. Docker helps deploy entire application dependencies without having to install an operating system. It also provides the ability to upgrade or downgrade. The lack of unnecessary files for developers and users of the operating system is incredibly important for embedded systems and IoT devices. In other words, the Docker infrastructure on ARM provides the SaaS distribution model of hardware products. The Docker infrastructure allows updating the differences of containers, thereby saving a lot of bandwidth. It also provides a great advantage for embedded systems that often have poor connectivity. Saving bandwidth doesn't just mean reducing costs. For a device with poor connectivity, installing small updates is faster and more reliable.

Without using the Docker infrastructure, updates can be done, but Docker containers need to be rebuilt or maintained. The isolated containers can be used on the device without interfering with each other. If this isolation is applied properly, these capabilities can be packed and operated without fear, this advantage can be used as an ecosystem of Dockerfiles. Each Dockerfile can add extra ability to IoT devices. Example infrastructure concept is as in Figure 3.8. However, running every container one by one causes large reductions in performance.

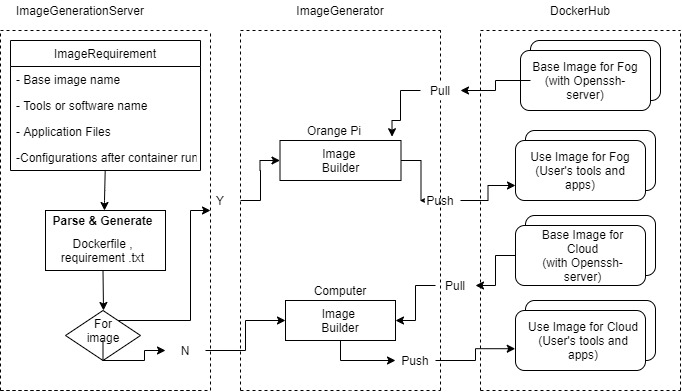


Figure 3. 8: Example of Conceptual Design of Docker over Raspberry Pi

If an app contains local dependencies, any app that includes deploying across multiple devices will take advantage of the Docker / SBC combination. Sometimes the use of Docker infrastructure is considered as an overhead in IoT devices. It is slower on average than x86 devices, but it has an incredibly powerful structure compared to Arduino. One of the benefits of Docker technology is, it is not a virtualized machine, so it does not overload SBC. If any application can be done in SBC without using Docker, it can also be done using Docker.

The architecture of Method B is as in Figure 3.9.

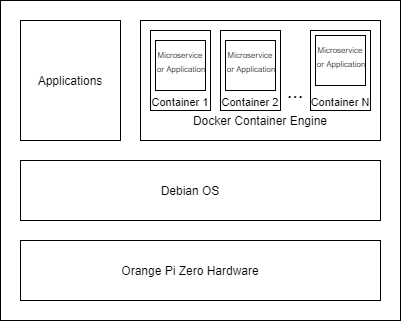


Figure 3. 9: Architecture of Method B

In this method, there is a Python file containing the code of each sensor, and a Docker image specific to this file. These image files are run in separate containers. Therefore, this container will be used for each sensor one by one. Creating different components of an application in different containers makes the maintenance easier for the application. Docker checks the differences between the version and whether there is a change in the Docker container, the basic applications in the container are updated.

### All-in-One Container

Docker enables any software to be run in an isolated environment called a container. The container is similar to the virtual machine but works with a completely different logic. Docker ensures consistency in projects and tests. For example, if a Web application is working correctly in the Docker container, it can work correctly anywhere. This feature makes it incredibly easy to manage project dependencies. It is not concerned with external modules and libraries called directly by the code. If a user has an open-source project they want to download and run, this can be easily accomplished by launching a container. It not only runs code that takes advantage of a reproducible environment, but it can also generate code in containers. It is used for the cross-compilation of SBCs. The container structure is quite clever. It is a completely independent operating system and runs on simulated hardware. However, containers natively share the kernel of the host. Example conceptual design of the virtualized infrastructure of Docker can be seen in Figure 3.10.

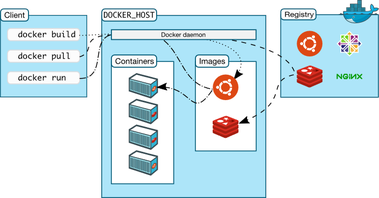


Figure 3. 10: Conceptual design of Docker [21]

This means that containers perform surprisingly better than virtual machines. When talking about a virtualization platform, the question of how many operating systems can be kept at the same time is asked. The question of how many Linux processes can run for the Docker platform should be asked. Also, running a virtual machine takes longer than running a Docker container. Because containers are more lightweight than virtual machines, all modules of an application can be run in different containers. These applications may be database servers or different containers. Structurally, Docker removes Linux domains, making their network interfaces look like a completely independent container, allowing containers to be separated while sharing the main kernel. Docker has very powerful tools that can run multiple containers and services, such as docker-compose, to automate such workflows.

The architecture of Method C is as in Figure 3.11.

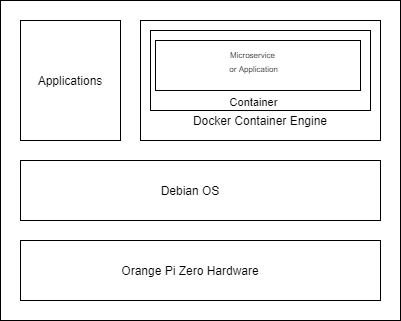


Figure 3. 11: Architecture of Method C

The software can be easily downloaded and run without requiring any installation. Docker can reproduce the code. It provides the opportunity to switch between platforms. Images can be run on any platform once installed.

In this use, a Docker container is created containing the python code of all sensors. Then Docker's image file is run in the container. Thus, all sensors can be operated quickly in a single container when the basic application container is started and stopped. Besides, it is more convenient to update all the codes using only one container.

## System Software

### Docker

Docker is a virtualization technology where you can run your applications and programs faster. Thanks to Docker, you can test and distribute your applications faster. When you make a change in your application on the Docker platform, instead of recompiling the whole program in general, it can only implement the change you made in the code and run your program. If there is an update or change in a version where you run your application, or if there is a change in the docker, it can quickly make the necessary updates and run your application.

Docker works by providing a regular way to run your code and is an operating system for containers. Similar to a virtual machine virtualizing the server hardware, containers virtualize the operating system of a server. Docker is installed on each server and provides simple commands that you can use to create, start, or stop containers.

In this project, the Docker Infrastructure 19.03.8 [22] version used.

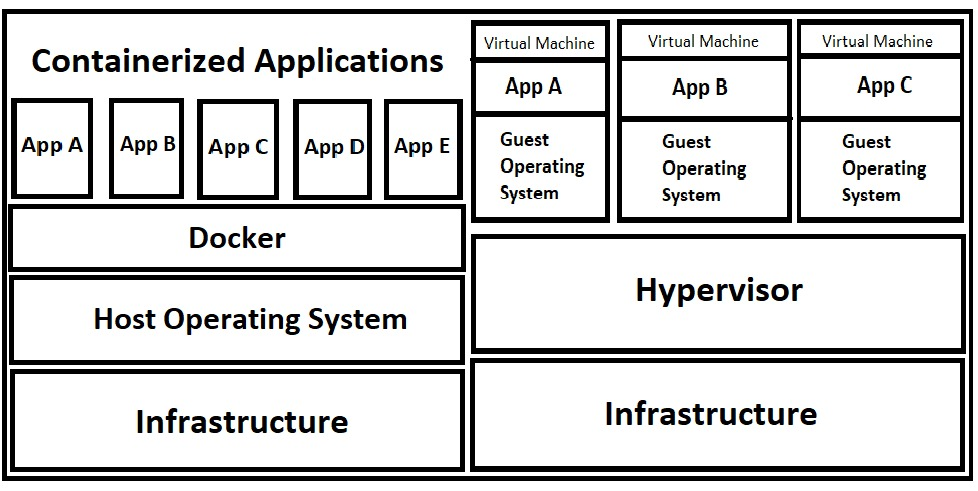


Figure 3. 12: Docker Infrastructure

Docker was installed on the device to use Docker containers [23]. Dockerfile was created for running Python files and use libraries required for the operation of the sensors. A simple Dockerfile example is as follows.

FROM Python:3.7

COPY test.py /

CMD python3 test.py

After editing the Dockerfile file, the image is built int the same folder directory with the required Python codes and libraries. In the directory, to build a Docker image, docker build . command used.

root@orangepizero: ~/docker docker build -t “ImageTag” .

The created and tagged image file is pushed to the Docker Hub platform for remote use.

root@orangepizero: ~/docker docker push “ImageTag” “DockerHubUser/Repository”

Images on the repository on the Docker Hub platform are ready to be downloaded and run.

root@orangepizero: ~/docker docker pull “DockerHubUser/Repository/ImageTag”

### Debian

Debian is an operating system that Linux distributions like Ubuntu, Linux based on Debian, also they are known as Debian Derivatives. Debian has a total of 3 versions: stable, testing, and unstable.

* An unstable version is a version that keeps changing and updated at a time. This version can be considered as a Beta version.
* Testing version is a more stable version than most Linux distributions on the market, where certain parts can be version upgraded and tested to see if these pars are stable on their own but are working properly together. In a way, it can be thought of as a candidate for release. Testing versions go into a freezing process after all the enhancements are completed, after serious periods of system use and testing continue, the testing version gets a stable version number and is no longer referred to as testing.
* Stable is a long-running version of Debian. The versions that you can install on a machine and forget that machine, which can work for years without any updates are marked as stable after long tests.

In this project, Armbian Debian Buster 20.02.1 [24] version was used as an operating system in Orange Pi. To set up Debian on Orange Pi Zero, 16 GB microSD card used. Debian image downloaded and imaged into the microSD using the Win32 Disk Imager application [25].

### Python

Python is a high-level dynamically typed scripting and object-oriented language. It is the most commonly used language that is easy to learn to program. On the other hand, it gives a programmer a great opportunity to program in various operating systems.

In this project, Python 3.7.7 [26] used to make sensors run and convert data into readable for the user. To use Python, updates, and upgrades were done first. Then Python version 3.7.7 was installed.

root@orangepizero: ~# apt-get update

root@orangepizero: ~# apt-get upgrade

root@orangepizero: ~# apt install python3.7

To use GPIO pins, pyA20 [27], and OPi GPIO [28] libraries downloaded. To download libraries, pip3, which is a package manager for Python 3, installed.

root@orangepizero: ~# apt-get install python3-pip

root@orangepizero: ~# pip3 install pyA20

root@orangepizero: ~# pip3 install OPi.GPIO

### Putty

For remote connection to operate Orange Pi, Putty [29] can be the best option. It uses SSH port to connect the device and Linux operating system based platforms. It is free and opensource software.

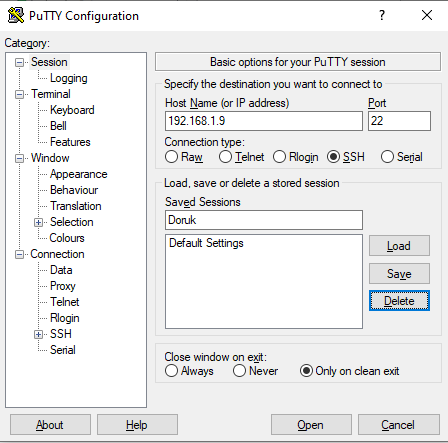


Figure 3. 13: Putty

The interface of the Putty program shown in Figure 3.13. To connect the device, the Ip address of the device was used, by using the Advanced IP Scanner [30] program, and port 22 of the device used.

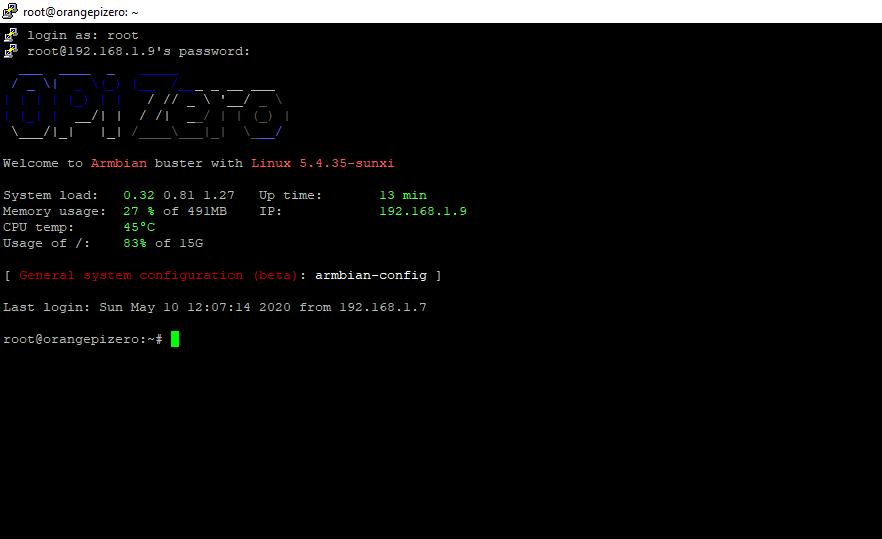


Figure 3. 14: Connection over Putty

When Orange Pi Zero is connected to SSH via Putty program, the interface is as in Figure 3.14. This interface is used to take the necessary actions.

# EXPERIMENTAL RESULTS

This section examines the effects of container virtualization technology on an IoT device. Looking at the test results, the data were categorized as CPU Utilization, CPU Temperature, CPU Speed ​​(MHz), and Network Bandwidth (Mbps). In the graphs, orange data show the Method A presented, that is, the average method presented by the traditional method for sensors using the Orange Pi device. The gray data shows the results where the Method B presented is done using a separate Docker Container structure for each sensor code. Finally, the green data show the effects of the remote access advantage of the virtualization technology provided by us, such as version update using a single Docker Container proposed for the solution of the problem presented in this project. All test results were made considering the data that changes in the device are updated every five seconds. The operating time of the method considered in the first 6 seconds and the final data were taken into consideration the fifth second after the method used was stopped.

These tests were not performed by installing any open source program on the device. The -top function on the device is used to list processes currently running on the system.

root@orangepizero:~# top

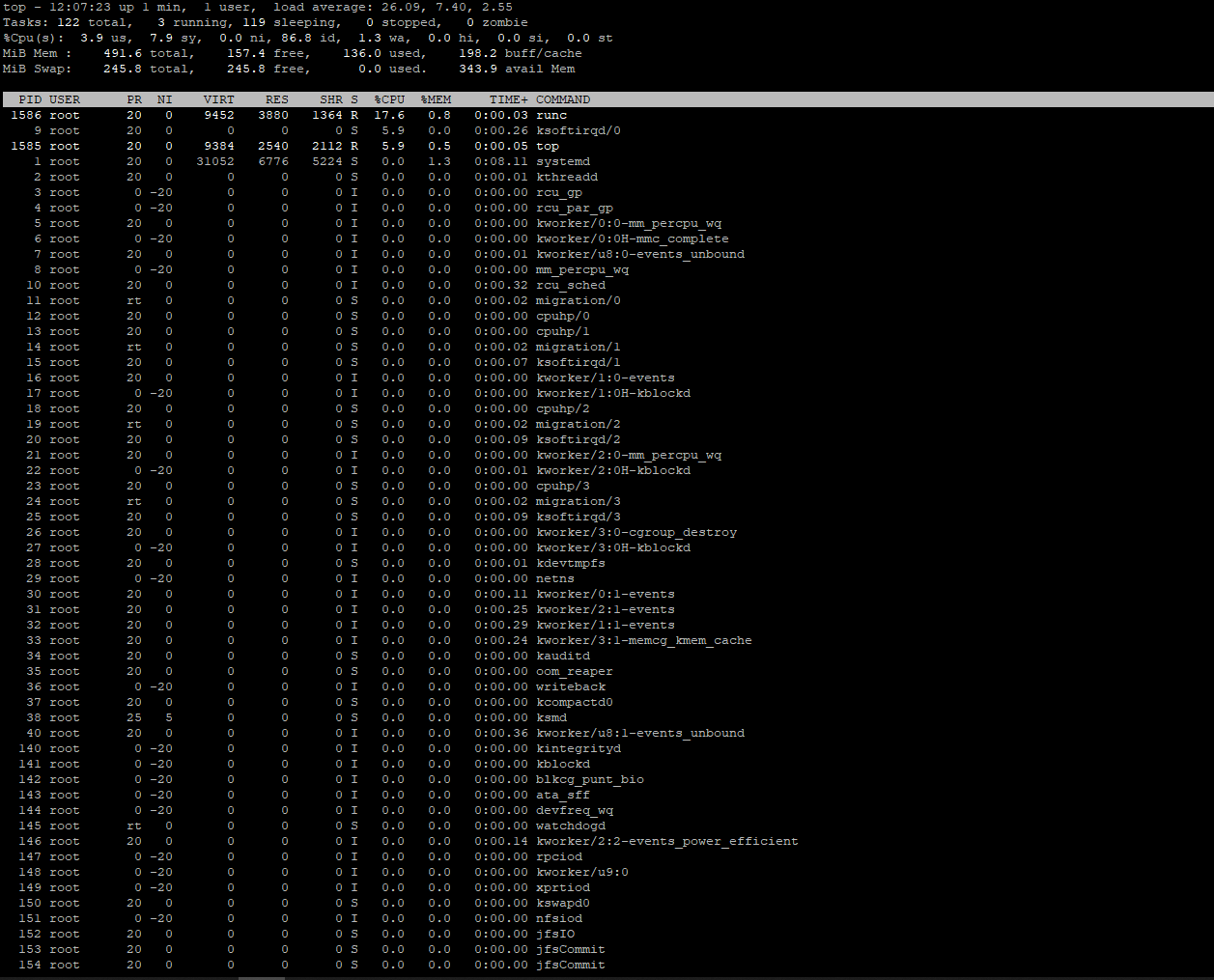


Figure 4. 1: Example of -top command

To observe CPU Speed, CPU temperature, CPU loading, cooling state, etc., armbianmotinor -m command used as shown in Figure 4.2.

root@orangepizero:~# armbianmonitor -m

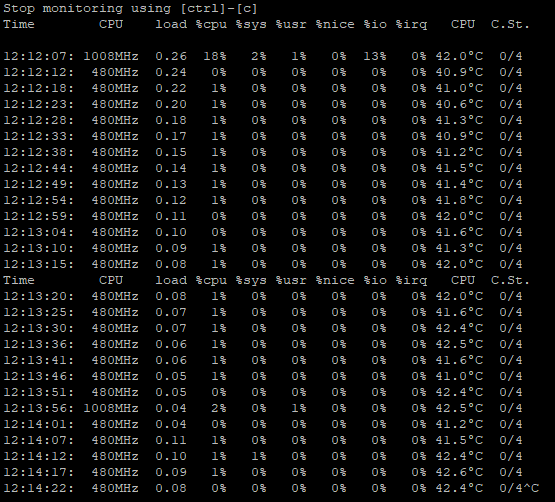


Figure 4. 2: Example of armbianmotinor

The data received through these commands were used to collect the data of the tests.

To benchmark network bandwidth, the iperf command used. Wi-Fi connection used on a home network. The home network connection speed is 24Mbps.

## CPU Utilization

As seen in Figure 4.3, the difference between the traditional method and the proposed all-in-one container method is too small to be considered. CPU utilization result is the expected result for the Method B. Method B causes the same performance loss as a computer running more than one VMs. Although the modules compromise on performance, the independent operation of the modules gives the flexibility of management and updating. The increase in CPU utilization by 27 times has a negative effect on performance, but SBCs have enough hardware to handle this performance. Other methods used in this project, Method A and Method C, are almost negligible in CPU utilization. In other words, it can be said that there is no performance difference between Method A and C.

Figure 4. 3: Comparison with the proposed methods (CPU Utilization)

## CPU Speed

Figure 4.4 shows that when Method B is used, the processor speed is higher, and the processor is used more. It is seen that this method is not preferred in terms of performance. The ability of multiple microservices to work with pipelines simultaneously plays an important role in influencing such performance. However, it can be used when running multiple files simultaneously. If multiple runtime services are needed, it would be better to use an individual container for each sensor. It provides the flexibility to maintain service without touching other services.

Figure 4. 4: Comparison with the proposed methods (CPU Speed MHz)

Apart from that, Method A and Method C do not show any difference in performance and processor usage. Method C method has important advantages such as remote access and version updates. This allows remote management and version updates to solve problems seen in the traditional method using the Method C method without causing degradation of performance management.

CPU speeds (MHz), Method B and Method C measurement of Method A are given in Figure 4.4. These test results show that, as with the CPU Usage test, the container structure is affected only when more than one container is running at a time.

## Network Bandwidth

No change in bandwidth was observed for each method tested. As seen in Figure 4.5, the average 6-second Network Bandwidth values of each method were calculated together. Docker containers have minimized bandwidth usage thanks to its infrastructure.

Figure 4. 5: Overall Network Bandwidth (Kbps)

## CPU Temperature

Temperature is one of the biggest challenges of IoT devices. When the methods used in this study are tested, the average temperature values are shown in Figure 4.6. When each method is used for approximately one minute, the average temperature values are 45.81 ° C in Method A, 45.87 ° C in Method B and 45.74 ° C in Method C. In terms of performance, Method B has a higher probability of heating, as it tears the device more. Temperature is an important consideration.

Figure 4. 6: CPU Temperature (°C)

# DISCUSSION

Single Board Computers is a computer containing memory, microprocessor, input/output, and all other necessary features of a circuit board. These devices can usually be easily connected to the systems. It can work regardless of the working environment and process. It does not require the developer to have a comprehensive programming experience as it is intended for the younger generation to learn about programming. Usually, Python programming language, which is less complex than existing languages, is used. For example, it allows concepts to be written using fewer lines. In addition, SBCs allow new ideas to be tried and turned into something completely different. SD cards on the device can be changed easily, so there is no need to spend a lot of time to reinstall the software. These devices are perfect for adaptive technologies. For example, prototype embedded systems and digital jukeboxes can be easily made using this device. These devices can create complex and effective products more economically. Nick Heath’s [31], SBCs like Raspberry Pi, testing firewall designs It creates a low-cost solution for such processes. Besides, SBCs are successful in energy saving, easy to recycle, unlike large servers that require a lot of energy and cooling systems, they emit very little emissions. SBCs generally provide less performance than standard computers, although they can also be used as personal computers where daily operations can be performed. SBC's being low cost, small, running like a server, performing different tasks, easily connecting to other hardware provide advantages over normal computers. However, even though SBCs can perform different tasks, they have some limitations for hardware reasons. It is not compatible with some popular operating systems such as x86-based Linux and Windows. This incompatibility requires the use of open-source operating systems such as Linux-based Debian. As seen in Table 3.1 above, the lack of a Wi-Fi module on the device may make it necessary to connect via Ethernet only. Also, low download speed and processor performance may cause interactive multitasking. Additional accessories such as USB power supply, SD card, keyboard, mouse, HDMI cable, Ethernet cable are needed for the use of these devices. Connecting the pins incorrectly can damage the device as there is no fuse protection. As with the Arduino device, GPIO pins do not have the ability to convert analog data to digital data. Therefore, when it is necessary to increase the reliability of the data coming from the sensor to the device, it is more efficient to work with analog signals. To convert digital signals to analog signals, it is necessary to have Analog to Digital Converter.

When using SBC devices, the user or developer must be physically on the same network to connect to the device. This restriction poses a major disadvantage for developers, especially in version update situations. This project was carried out to program SBC devices remotely using Docker container infrastructure, to update versions, to perform maintenance and tests. Docker is an application virtualization platform that uses the container structure. The Docker container image is the software package required for an application to work. The container infrastructure and operating system are independent, light, and executable. In this way, it can work on every operating system by overcoming Linux and Windows conflict. Before container technology, VMs were used to host applications. When using the VM structure, even a network simulation can be made by dividing a large server into multiple virtual machines. Having virtual machines requires installing and updating the operating system of each virtual machine, as well as having operating system licenses. However, container technology gives a much better level of abstraction than VMs. Containers operate at the application layer. More than one container can work on the same machine and since they do not contain an advanced operating system, they take up less space than VMs. Other advantages of containers are low resource consumption, fast preloading, work in any environment, facilitate microservice management. Also, the container is a reliable platform. Each container operates in its own namespace and all containers use the same kernel to manage their domain names. However, since there is no full operating system, unsafe systems housed in containers can be captured by hackers. Since containers use the same core, situations where complete insulation is not present, can pose a risk.

If the right cloud platform can be chosen, virtualization technologies such as Docker can be eliminated using the PaaS structure. Some cloud platforms still do not support Linux or Windows operating systems locally, but the workspace can be prepared with the container structure and moved to the cloud platform. The gradual connection of automation processes with SBC cards allows IoT, Docker, and Cloud structure to come together more.

In the project, the Docker container structure was compared in two different ways, compared to the situation used without Docker. Method A, the first method without Docker virtualization technology, was created to set standards for device testing. The operating system, Python and its libraries have been installed on the Orange Pi Zero device so that seven different sensors can work. As a result of performance tests, CPU Utilization value was observed as maximum 0.03%, minimum 0.01%, average 0.01%, CPU Speed ​​average 480Mhz, and temperature average 45.8 degrees Celsius. In Method B, which is the first method using the Docker container, seven different containers were created for seven different sensors. This method is important for updating or changing microservices regardless of stopping. In terms of usage, it tired the device, its temperature increased, and its performance decreased. According to test results, CPU Utilization value is maximum 0.27%, minimum 0.01%, average 0.05%, CPU Speed ​​is 3 times higher than the standard value, it has increased from minimum 480Mhz speed to maximum 1008Mhz speed, the average temperature has been 45.9 degrees Celsius. In Method C, the second method using the Docker container, a single container was created for seven different sensors. This method provides an advantage in terms of performance but prevents the independence of microservices. According to the test results, CPU Utilization maximum 0.03%, minimum 0.01%, average 0.01%, CPU Speed ​​is only once faster than the standard state, showing 648Mhz. The average temperature of the device was observed at 46.2 degrees Celsius. According to the results of the network bandwidth test, using the Docker structure did not cause a change.

Developing and updating the desired modules without affecting the running modules can ensure that the low CPU performance is ignored. According to the conclusions made based on the test results, it is recommended to use hardware-optimized versions of SBC devices in studies where microservices will be used with Docker. Docker virtualization technology provides an indispensable advantage for the installation and use of development environments for SBC devices.

Table 5. 1: Advantages and Disadvantages of Methods

|  |  |  |  |
| --- | --- | --- | --- |
|  | Method A | Method B | Method C |
| Advantages | The simplest method for beginners. | Thanks to its modular structure, changes in one module do not affect other modules. | Since all codes are run in a single container, it uses less source and provides high performance. |
| Disadvantages | Not suitable for remote control and version update. | This method causes lower performance by using more sources due to the operation of more than one container. | Since all codes are in a single container, all codes are affected when making changes to a code |

Table 5.1 shows the advantages and disadvantages of proposed Methods.

# CONCLUSIONS

To update or make some code changes for smart developer cards such as raspberry pi and orange pi, the project has been tested for the docker platform and the usage patterns used to remove the need to connect to the device each time. Virtualization technology, which has been popular in recent years and used mostly in servers, has many disadvantages. Therefore, docker technology, a new type of virtualization, helps solve many problems.

In this study, three different usage methods were tested. These methods are traditional, Individual Container Usage, and all in one. When a traditional method is desired to be changed and updated, it has to be connected to the device. In such cases, version update or code update is not possible as you need to connect the device with SSH on the same network, so this method of use is not efficient on IoT devices. Method B of use is the individual method. This method is not usable in terms of performance because each code file runs in separate containers. The last method is the all in one method. In this method, each code file is optimized and run on a single container. This method is equivalent in performance tests, but it is more advantageous compared to Method A in remote control and version update. It is a preferable technology in IoT devices due to its success in remote control and updating.

Finally, in this study, three methods were examined, and Method C is more comfortable than the other two methods and there is no difference in performance compared to Method A. Therefore, Method C will meet many needs.

# REFERENCES

1. https://www.docker.com/
2. Matthias Kovatsch, Martin Lanter, Simon Duquennoy, (2012) Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications
3. A. P. Fournaris, S. Giannoulis and C. Koulamas, (2019) Evaluating CoAP End to End Security for Constrained Wireless Sensor Networks
4. J. Phung, Y. C. Lee and A. Y. Zomaya, (2020) Lightweight Power Monitoring Framework for Virtualized Computing Environments
5. K. Fysarakis, D. Mylonakis, C. Manifavas and I. Papaefstathiou, (2016) Node.DPWS: Efficient Web Services for the Internet of Things
6. L. Mainetti, V. Mighali, L. Patrono, P. Rametta and S. L. Oliva, (2013) A novel architecture enabling the visual implementation of web of Things applications
7. Patrick Dittmer, Marius Veigt, Bernd Scholz-Reiter, Nils Heidmann, Steffen Paul, (2012) The Intelligent Container as a Part of the Internet of Things
8. A. Celesti, D. Mulfari, M. Fazio, M. Villari and A. Puliafito, (2016) Exploring Container Virtualization in IoT Clouds
9. A. Celesti, M. Fazio, M. Giacobbe, A. Puliafito and M. Villari, (2016) Characterizing Cloud Federation in IoT
10. A. Celesti, A. Buzachis, A. Galletta, M. Fazio and M. Villari, (2018) A NoSQL Graph Approach to Manage IoTaaS in Cloud/Edge Environments
11. Antonio Celesti, Mulfari Davide, Antonino Galetta, (2019) A study on container virtualization for guarantee quality of service in Cloud-of-Things
12. Saiful Hoque, Mathias Santos de Brito, Thomas Magedanz, Alexander Willner, Oliver Keil (2017) Towards Container Orchestration in Fog Computing Infrastructures
13. S. Hoque, M. S. De Brito, A. Willner, O. Keil, and T. Magedanz, (2017) Towards Container Orchestration in Fog Computing Infrastructures
14. Hong-Linh Truong and Schahram Dustdar, (2015) Principles for Engineering IoT Cloud Systems
15. Roberto Morabito (2016) A Performance Evaluation of Container Technologies on Internet of Things Devices
16. Roberto Morabito (2017) Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation
17. B. Grandhi, S. Chickerur and M. S. Patil, (2018) Performance Analysis of MySQL, Apache Spark on CPU and GPU
18. S. S. Prayogo, Y. Mukhlis and B. K. Yakti (2019) The Use and Performance of MQTT and CoAP as Internet of Things Application Protocol using NodeMCU ESP8266
19. http://www.orangepi.org/orangepizero/
20. https://www.arduino.cc/
21. https://docs.docker.com/get-started/overview/
22. https://docs.docker.com/engine/release-notes/#19038
23. [PAULOFRAZAO](https://www.docker.com/blog/author/paulofrazao/) (2014) Happy Pi Day with Docker and Raspberry Pi, URL: <https://www.docker.com/blog/happy-pi-day-docker-raspberry-pi/>
24. https://dl.armbian.com/orangepizero/archive/
25. <https://www.armbian.com/download/>
26. https://www.python.org/downloads/release/python-377/
27. https://pypi.org/project/pyA20/
28. https://pypi.org/project/OPi.GPIO/
29. https://www.putty.org/
30. https://www.advanced-ip-scanner.com/tr/
31. Heath, A. (2012) Raspberry Pi: Five ways business can use it. According to the article retrieved from https://www.techrepublic.com/blog/european-technology/raspberry-pi-five-ways-business-can-use-it/

# APPENDIX

## APPENDIX A – Method A

Buzzer.py:

import OPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setwarnings(False)

GPIO.setup(PA19, GPIO.OUT)

GPIO.setup(18,GPIO.OUT)

tone1 = GPIO.PWM(21, 100)

tone2 = GPIO.PWM(20, 250)

tone1.start(50)

tone2.start(0)

c = [32, 65, 131, 262, 523]

db= [34, 69, 139, 277, 554]

d = [36, 73, 147, 294, 587]

eb= [37, 78, 156, 311, 622]

e = [41, 82, 165, 330, 659]

f = [43, 87, 175, 349, 698]

gb= [46, 92, 185, 370, 740]

g = [49, 98, 196, 392, 784]

ab= [52, 104, 208, 415, 831]

a = [55, 110, 220, 440, 880]

bb= [58, 117, 223, 466, 932]

b = [61, 123, 246, 492, 984]

cmajor = [c, d, e, f, g, a, b]

aminor = [a, b, c, d, e, f, g]

def playScale(scale, pause):

for i in range(0, 5):

for note in scale:

tone1.ChangeFrequency(note[i])

time.sleep(pause)

tone1.stop()

starwars\_notes = [c[1], g[1], f[1], e[1], d[1], c[2], g[1], f[1], e[1], d[1], c[2], g[1],

f[1], e[1], f[1], d[1]]

starwars\_beats = [4,4,1,1,1,4,4,1,1,1,4,4,1,1,1,4]

def playSong(songnotes, songbeats, tempo):

tone1.ChangeDutyCycle(50)

for i in range(0, len(songnotes)):

tone1.ChangeFrequency(songnotes[i])

time.sleep(songbeats[i]\*tempo)

tone1.ChangeDutyCycle(0)

playSong(starwars\_notes, starwars\_beats, 0.2)

Flame.py:

import OPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

GPIO.setup(7,GPIO.IN)

input = GPIO.input(7)

while True:

if (GPIO.input(7)):

print("Flame Detected")

else:

print("Not Detected")

Gas.py:

import os

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

from time import sleep

channel = port.PA18

gpio.init()

gpio.setcfg(channel, gpio.INPUT)

sign = 0

space\_start = 0

space\_end = 0

space = 0

pulse = 0

try:

os.system("clear");

print ("Press CTRL+C to exit")

while True:

while gpio.input(channel)==1:

pulse = 0

space\_end = time.time()

while gpio.input(channel)==0:

pulse = pulse + 1

space\_start = time.time()

sleep(0.5)

print (pulse)

space\_duration = (space\_start - space\_end) \*100000

print (space\_duration)

except KeyboardInterrupt:

print ("Bitti.")

Motion.py:

import os

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

from time import sleep

channel = port.PA18

gpio.init()

gpio.setcfg(channel, gpio.INPUT)

sign = 0

space\_start = 0

space\_end = 0

space = 0

pulse = 0

try:

os.system("clear");

print ("Press CTRL+C to exit")

while True:

while gpio.input(channel)==1:

pulse = 0

space\_end = time.time()

while gpio.input(channel)==0:

pulse = pulse + 1

space\_start = time.time()

sleep(0.5)

print (pulse)

space\_duration = (space\_start - space\_end) \*100000

print (space\_duration)

except KeyboardInterrupt:

print ("Bitti.")

Rain.py:

import os

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

from time import sleep

channel = port.PA18

gpio.init()

gpio.setcfg(channel, gpio.INPUT)

sign = 0

space\_start = 0

space\_end = 0

space = 0

pulse = 0

try:

os.system("clear");

print ("Press CTRL+C to exit")

while True:

while gpio.input(channel)==1:

pulse = 0

space\_end = time.time()

while gpio.input(channel)==0:

pulse = pulse + 1

space\_start = time.time()

sleep(0.5)

print (pulse)

space\_duration = (space\_start - space\_end) \*100000

print (space\_duration)

except KeyboardInterrupt:

print ("Bitti.")

DHT11.py for Temperature.py:

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

#import RPi

class DHT11Result:

'DHT11 sensor result returned by DHT11.read() method'

ERR\_NO\_ERROR = 0

ERR\_MISSING\_DATA = 1

ERR\_CRC = 2

error\_code = ERR\_NO\_ERROR

temperature = -1

humidity = -1

def \_\_init\_\_(self, error\_code, temperature, humidity):

self.error\_code = error\_code

self.temperature = temperature

self.humidity = humidity

def is\_valid(self):

return self.error\_code == DHT11Result.ERR\_NO\_ERROR

class DHT11:

'DHT11 sensor reader class for Raspberry'

\_\_pin = 0

def \_\_init\_\_(self, pin):

self.\_\_pin = pin

def read(self):

gpio.setcfg(self.\_\_pin, gpio.OUTPUT)

# send initial high

self.\_\_send\_and\_sleep(gpio.HIGH, 0.05)

# pull down to low

self.\_\_send\_and\_sleep(gpio.LOW, 0.02)

# change to input using pull up

#gpio.setcfg(self.\_\_pin, gpio.INPUT, gpio.PULLUP)

gpio.setcfg(self.\_\_pin, gpio.INPUT)

gpio.pullup(self.\_\_pin, gpio.PULLUP)

# collect data into an array

data = self.\_\_collect\_input()

# parse lengths of all data pull up periods

pull\_up\_lengths = self.\_\_parse\_data\_pull\_up\_lengths(data)

# if bit count mismatch, return error (4 byte data + 1 byte checksum)

if len(pull\_up\_lengths) != 40:

return DHT11Result(DHT11Result.ERR\_MISSING\_DATA, 0, 0)

# calculate bits from lengths of the pull up periods

bits = self.\_\_calculate\_bits(pull\_up\_lengths)

# we have the bits, calculate bytes

the\_bytes = self.\_\_bits\_to\_bytes(bits)

# calculate checksum and check

checksum = self.\_\_calculate\_checksum(the\_bytes)

if the\_bytes[4] != checksum:

return DHT11Result(DHT11Result.ERR\_CRC, 0, 0)

# ok, we have valid data, return it

return DHT11Result(DHT11Result.ERR\_NO\_ERROR, the\_bytes[2], the\_bytes[0])

def \_\_send\_and\_sleep(self, output, sleep):

gpio.output(self.\_\_pin, output)

time.sleep(sleep)

def \_\_collect\_input(self):

# collect the data while unchanged found

unchanged\_count = 0

# this is used to determine where is the end of the data

max\_unchanged\_count = 100

last = -1

data = []

while True:

current = gpio.input(self.\_\_pin)

data.append(current)

if last != current:

unchanged\_count = 0

last = current

else:

unchanged\_count += 1

if unchanged\_count > max\_unchanged\_count:

break

return data

def \_\_parse\_data\_pull\_up\_lengths(self, data):

STATE\_INIT\_PULL\_DOWN = 1

STATE\_INIT\_PULL\_UP = 2

STATE\_DATA\_FIRST\_PULL\_DOWN = 3

STATE\_DATA\_PULL\_UP = 4

STATE\_DATA\_PULL\_DOWN = 5

state = STATE\_INIT\_PULL\_DOWN

lengths = [] # will contain the lengths of data pull up periods

current\_length = 0 # will contain the length of the previous period

for i in range(len(data)):

current = data[i]

current\_length += 1

if state == STATE\_INIT\_PULL\_DOWN:

if current == gpio.LOW:

# ok, we got the initial pull down

state = STATE\_INIT\_PULL\_UP

continue

else:

continue

if state == STATE\_INIT\_PULL\_UP:

if current == gpio.HIGH:

# ok, we got the initial pull up

state = STATE\_DATA\_FIRST\_PULL\_DOWN

continue

else:

continue

if state == STATE\_DATA\_FIRST\_PULL\_DOWN:

if current == gpio.LOW:

# we have the initial pull down, the next will be the data pull up

state = STATE\_DATA\_PULL\_UP

continue

else:

continue

if state == STATE\_DATA\_PULL\_UP:

if current == gpio.HIGH:

# data pulled up, the length of this pull up will determine whether it is 0 or 1

current\_length = 0

state = STATE\_DATA\_PULL\_DOWN

continue

else:

continue

if state == STATE\_DATA\_PULL\_DOWN:

if current == gpio.LOW:

# pulled down, we store the length of the previous pull up period

lengths.append(current\_length)

state = STATE\_DATA\_PULL\_UP

continue

else:

continue

return lengths

def \_\_calculate\_bits(self, pull\_up\_lengths):

# find shortest and longest period

shortest\_pull\_up = 1000

longest\_pull\_up = 0

for i in range(0, len(pull\_up\_lengths)):

length = pull\_up\_lengths[i]

if length < shortest\_pull\_up:

shortest\_pull\_up = length

if length > longest\_pull\_up:

longest\_pull\_up = length

# use the halfway to determine whether the period it is long or short

halfway = shortest\_pull\_up + (longest\_pull\_up - shortest\_pull\_up) / 2

bits = []

for i in range(0, len(pull\_up\_lengths)):

bit = False

if pull\_up\_lengths[i] > halfway:

bit = True

bits.append(bit)

return bits

def \_\_bits\_to\_bytes(self, bits):

the\_bytes = []

byte = 0

for i in range(0, len(bits)):

byte = byte << 1

if (bits[i]):

byte = byte | 1

else:

byte = byte | 0

if ((i + 1) % 8 == 0):

the\_bytes.append(byte)

byte = 0

return the\_bytes

def \_\_calculate\_checksum(self, the\_bytes):

return the\_bytes[0] + the\_bytes[1] + the\_bytes[2] + the\_bytes[3] & 255

Temperature.py:

from pyA20.gpio import gpio

from pyA20.gpio import port

#import RPi.GPIO as GPIO

import dht11

import time

import datetime

# initialize GPIO

#gpio.setwarnings(False)

#gpio.setmode(GPIO.BCM)

PIN2 = port.PG6

gpio.init()

#gpio.cleanup()

# read data using pin 14

instance = dht11.DHT11(pin=PIN2)

try:

while True:

result = instance.read()

if result.is\_valid():

print("Last valid input: " + str(datetime.datetime.now()))

print("Temperature: %d C" % result.temperature)

print("Humidity: %d %%" % result.humidity)

time.sleep(0.5)

except KeyboardInterrupt:

print ("Bye.")

## APPENDIX B – Method B

Buzzer.py:

import OPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setwarnings(False)

GPIO.setup(PA19, GPIO.OUT)

GPIO.setup(18,GPIO.OUT)

tone1 = GPIO.PWM(21, 100)

tone2 = GPIO.PWM(20, 250)

tone1.start(50)

tone2.start(0)

c = [32, 65, 131, 262, 523]

db= [34, 69, 139, 277, 554]

d = [36, 73, 147, 294, 587]

eb= [37, 78, 156, 311, 622]

e = [41, 82, 165, 330, 659]

f = [43, 87, 175, 349, 698]

gb= [46, 92, 185, 370, 740]

g = [49, 98, 196, 392, 784]

ab= [52, 104, 208, 415, 831]

a = [55, 110, 220, 440, 880]

bb= [58, 117, 223, 466, 932]

b = [61, 123, 246, 492, 984]

cmajor = [c, d, e, f, g, a, b]

aminor = [a, b, c, d, e, f, g]

def playScale(scale, pause):

for i in range(0, 5):

for note in scale:

tone1.ChangeFrequency(note[i])

time.sleep(pause)

tone1.stop()

starwars\_notes = [c[1], g[1], f[1], e[1], d[1], c[2], g[1], f[1], e[1], d[1], c[2], g[1],

f[1], e[1], f[1], d[1]]

starwars\_beats = [4,4,1,1,1,4,4,1,1,1,4,4,1,1,1,4]

def playSong(songnotes, songbeats, tempo):

tone1.ChangeDutyCycle(50)

for i in range(0, len(songnotes)):

tone1.ChangeFrequency(songnotes[i])

time.sleep(songbeats[i]\*tempo)

tone1.ChangeDutyCycle(0)

playSong(starwars\_notes, starwars\_beats, 0.2)

Dockerfile:

FROM arm32v7/python:3-buster

RUN apt-get update

RUN apt-get upgrade -y

RUN apt-get install -y apt-utils

RUN apt-get install python3 -y

RUN apt-get install git -y

RUN apt-get install python3-pip -y

RUN apt-get install python3-setuptools

RUN pip install wheel

RUN pip install OrangePi.GPIO

COPY buzzer.py /

RUN python3 hello\_world.py

Flame.py:

import OPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

GPIO.setup(7,GPIO.IN)

input = GPIO.input(7)

while True:

if (GPIO.input(7)):

print("Flame Detected")

else:

print("Not Detected")

Dockerfile:

FROM arm32v7/python:3-buster

RUN apt-get update -y

RUN apt-get upgrade -y

RUN apt-get install -y apt-utils

RUN apt-get install python3 -y

RUN apt-get install git -y

RUN apt-get install python3-pip -y

RUN apt-get install python3-setuptools

RUN pip install wheel

RUN pip install OrangePi.GPIO

COPY flame.py ./

RUN python3 flame.py

Gas.py:

import os

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

from time import sleep

channel = port.PA18

gpio.init()

gpio.setcfg(channel, gpio.INPUT)

sign = 0

space\_start = 0

space\_end = 0

space = 0

pulse = 0

try:

os.system("clear");

print ("Press CTRL+C to exit")

while True:

while gpio.input(channel)==1:

pulse = 0

space\_end = time.time()

while gpio.input(channel)==0:

pulse = pulse + 1

space\_start = time.time()

sleep(0.5)

print (pulse)

space\_duration = (space\_start - space\_end) \*100000

print (space\_duration)

except KeyboardInterrupt:

print ("Bitti.")

Dockerfile:

FROM arm32v7/python:3-buster

COPY gas.py ./

RUN apt-get update -y

RUN apt-get upgrade -y

RUN apt-get install python3-pip -y

RUN apt-get install -y apt-utils

RUN pip3 install --user python-dev-tools

RUN pip3 install --no-cache-dir OrangePi.GPIO

CMD ["python3", "gas.py"]

Motion.py:

import os

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

from time import sleep

channel = port.PA18

gpio.init()

gpio.setcfg(channel, gpio.INPUT)

sign = 0

space\_start = 0

space\_end = 0

space = 0

pulse = 0

try:

os.system("clear");

print ("Press CTRL+C to exit")

while True:

while gpio.input(channel)==1:

pulse = 0

space\_end = time.time()

while gpio.input(channel)==0:

pulse = pulse + 1

space\_start = time.time()

sleep(0.5)

print (pulse)

space\_duration = (space\_start - space\_end) \*100000

print (space\_duration)

except KeyboardInterrupt:

print ("Bitti.")

Dockerfile:

FROM arm32v7/python:3-buster

RUN apt-get update

RUN apt-get upgrade -y

RUN apt-get install -y apt-utils

RUN apt-get install python3 -y

RUN apt-get install git -y

RUN apt-get install python3-pip -y

RUN apt-get install python3-setuptools

RUN pip install wheel

RUN pip install OrangePi.GPIO

COPY motion.py ./

RUN python3 motion.py

Rain.py:

import os

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

from time import sleep

channel = port.PA18

gpio.init()

gpio.setcfg(channel, gpio.INPUT)

sign = 0

space\_start = 0

space\_end = 0

space = 0

pulse = 0

try:

os.system("clear");

print ("Press CTRL+C to exit")

while True:

while gpio.input(channel)==1:

pulse = 0

space\_end = time.time()

while gpio.input(channel)==0:

pulse = pulse + 1

space\_start = time.time()

sleep(0.5)

print (pulse)

space\_duration = (space\_start - space\_end) \*100000

print (space\_duration)

except KeyboardInterrupt:

print ("Bitti.")

Dockerfile:

FROM arm32v7/python:3-buster

RUN apt-get update

RUN apt-get upgrade -y

RUN apt-get install -y apt-utils

RUN apt-get install python3 -y

RUN apt-get install git -y

RUN apt-get install python3-pip -y

RUN apt-get install python3-setuptools

RUN pip install wheel

RUN pip install OrangePi.GPIO

COPY rain.py ./

RUN python3 rain.py

DHT11.py for Temperature.py:

import time

from pyA20.gpio import gpio

from pyA20.gpio import port

#import RPi

class DHT11Result:

'DHT11 sensor result returned by DHT11.read() method'

ERR\_NO\_ERROR = 0

ERR\_MISSING\_DATA = 1

ERR\_CRC = 2

error\_code = ERR\_NO\_ERROR

temperature = -1

humidity = -1

def \_\_init\_\_(self, error\_code, temperature, humidity):

self.error\_code = error\_code

self.temperature = temperature

self.humidity = humidity

def is\_valid(self):

return self.error\_code == DHT11Result.ERR\_NO\_ERROR

class DHT11:

'DHT11 sensor reader class for Raspberry'

\_\_pin = 0

def \_\_init\_\_(self, pin):

self.\_\_pin = pin

def read(self):

gpio.setcfg(self.\_\_pin, gpio.OUTPUT)

# send initial high

self.\_\_send\_and\_sleep(gpio.HIGH, 0.05)

# pull down to low

self.\_\_send\_and\_sleep(gpio.LOW, 0.02)

# change to input using pull up

#gpio.setcfg(self.\_\_pin, gpio.INPUT, gpio.PULLUP)

gpio.setcfg(self.\_\_pin, gpio.INPUT)

gpio.pullup(self.\_\_pin, gpio.PULLUP)

# collect data into an array

data = self.\_\_collect\_input()

# parse lengths of all data pull up periods

pull\_up\_lengths = self.\_\_parse\_data\_pull\_up\_lengths(data)

# if bit count mismatch, return error (4 byte data + 1 byte checksum)

if len(pull\_up\_lengths) != 40:

return DHT11Result(DHT11Result.ERR\_MISSING\_DATA, 0, 0)

# calculate bits from lengths of the pull up periods

bits = self.\_\_calculate\_bits(pull\_up\_lengths)

# we have the bits, calculate bytes

the\_bytes = self.\_\_bits\_to\_bytes(bits)

# calculate checksum and check

checksum = self.\_\_calculate\_checksum(the\_bytes)

if the\_bytes[4] != checksum:

return DHT11Result(DHT11Result.ERR\_CRC, 0, 0)

# ok, we have valid data, return it

return DHT11Result(DHT11Result.ERR\_NO\_ERROR, the\_bytes[2], the\_bytes[0])

def \_\_send\_and\_sleep(self, output, sleep):

gpio.output(self.\_\_pin, output)

time.sleep(sleep)

def \_\_collect\_input(self):

# collect the data while unchanged found

unchanged\_count = 0

# this is used to determine where is the end of the data

max\_unchanged\_count = 100

last = -1

data = []

while True:

current = gpio.input(self.\_\_pin)

data.append(current)

if last != current:

unchanged\_count = 0

last = current

else:

unchanged\_count += 1

if unchanged\_count > max\_unchanged\_count:

break

return data

def \_\_parse\_data\_pull\_up\_lengths(self, data):

STATE\_INIT\_PULL\_DOWN = 1

STATE\_INIT\_PULL\_UP = 2

STATE\_DATA\_FIRST\_PULL\_DOWN = 3

STATE\_DATA\_PULL\_UP = 4

STATE\_DATA\_PULL\_DOWN = 5

state = STATE\_INIT\_PULL\_DOWN

lengths = [] # will contain the lengths of data pull up periods

current\_length = 0 # will contain the length of the previous period

for i in range(len(data)):

current = data[i]

current\_length += 1

if state == STATE\_INIT\_PULL\_DOWN:

if current == gpio.LOW:

# ok, we got the initial pull down

state = STATE\_INIT\_PULL\_UP

continue

else:

continue

if state == STATE\_INIT\_PULL\_UP:

if current == gpio.HIGH:

# ok, we got the initial pull up

state = STATE\_DATA\_FIRST\_PULL\_DOWN

continue

else:

continue

if state == STATE\_DATA\_FIRST\_PULL\_DOWN:

if current == gpio.LOW:

# we have the initial pull down, the next will be the data pull up

state = STATE\_DATA\_PULL\_UP

continue

else:

continue

if state == STATE\_DATA\_PULL\_UP:

if current == gpio.HIGH:

# data pulled up, the length of this pull up will determine whether it is 0 or 1

current\_length = 0

state = STATE\_DATA\_PULL\_DOWN

continue

else:

continue

if state == STATE\_DATA\_PULL\_DOWN:

if current == gpio.LOW:

# pulled down, we store the length of the previous pull up period

lengths.append(current\_length)

state = STATE\_DATA\_PULL\_UP

continue

else:

continue

return lengths

def \_\_calculate\_bits(self, pull\_up\_lengths):

# find shortest and longest period

shortest\_pull\_up = 1000

longest\_pull\_up = 0

for i in range(0, len(pull\_up\_lengths)):

length = pull\_up\_lengths[i]

if length < shortest\_pull\_up:

shortest\_pull\_up = length

if length > longest\_pull\_up:

longest\_pull\_up = length

# use the halfway to determine whether the period it is long or short

halfway = shortest\_pull\_up + (longest\_pull\_up - shortest\_pull\_up) / 2

bits = []

for i in range(0, len(pull\_up\_lengths)):

bit = False

if pull\_up\_lengths[i] > halfway:

bit = True

bits.append(bit)

return bits

def \_\_bits\_to\_bytes(self, bits):

the\_bytes = []

byte = 0

for i in range(0, len(bits)):

byte = byte << 1

if (bits[i]):

byte = byte | 1

else:

byte = byte | 0

if ((i + 1) % 8 == 0):

the\_bytes.append(byte)

byte = 0

return the\_bytes

def \_\_calculate\_checksum(self, the\_bytes):

return the\_bytes[0] + the\_bytes[1] + the\_bytes[2] + the\_bytes[3] & 255

Temperature.py:

from pyA20.gpio import gpio

from pyA20.gpio import port

#import RPi.GPIO as GPIO

import dht11

import time

import datetime

# initialize GPIO

#gpio.setwarnings(False)

#gpio.setmode(GPIO.BCM)

PIN2 = port.PG6

gpio.init()

#gpio.cleanup()

# read data using pin 14

instance = dht11.DHT11(pin=PIN2)

try:

while True:

result = instance.read()

if result.is\_valid():

print("Last valid input: " + str(datetime.datetime.now()))

print("Temperature: %d C" % result.temperature)

print("Humidity: %d %%" % result.humidity)

time.sleep(0.5)

except KeyboardInterrupt:

print ("Bye.")

Dockerfile:

FROM arm32v7/python:3-buster

RUN apt-get update

RUN apt-get upgrade -y

RUN apt-get install -y apt-utils

RUN apt-get install python3 -y

RUN apt-get install git -y

RUN apt-get install python3-pip -y

RUN apt-get install python3-setuptools

RUN pip install wheel

RUN pip install OrangePi.GPIO

COPY temperature.py ./

COPY dht11.py ./

RUN python3 temperature.py

## APPENDIX C – Method C

Main.py:

import OPi.GPIO as GPIO

import dht11

import time

import datetime

from pyA20.gpio import gpio

from pyA20.gpio import port

PIN2 = port.PG6

gpio.init()

#gpio.cleanup()

instance = dht11.DHT11(pin=PIN2)

GPIO.setwarnings(False)

GPIO.setmode(GPIO.BOARD)

GPIO.setup(14, GPIO.IN) #Read output from Gas sensor

GPIO.setup(13, GPIO.IN) #Read output from Gas sensor

GPIO.setup(12, GPIO.IN) #Read output from PIR motion sensor

GPIO.setup(11, GPIO.IN) #Read output from PIR motion sensor

GPIO.setup(3, GPIO.OUT) #LED output pin

try:

while True:

result = instance.read()

if result.is\_valid():

print("Last valid input: " + str(datetime.datetime.now()))

print("Temperature: %d C" % result.temperature)

print("Humidity: %d %%" % result.humidity)

flame=GPIO.input(11)

gas=GPIO.input(12)

motion=GPIO.input(13)

rain=GPIO.input(14)

if flame==0: #When output from motion sensor is LOW

print ("No Flame",flame)

elif flame==1: #When output from motion sensor is HIGH

print ("Flame detected",flame)

if gas==0:

print ("No gas",gas)

elif gas==1:

print ("Gas detected", gas)

if motion==0: #When output from motion sensor is LOW

print ("No Motion", motion)

elif motion==1: #When output from motion sensor is HIGH

print ("Motion detected", motion)

if rain==0: #When output from motion sensor is LOW

print ("No Rain",rain)

elif rain==1: #When output from motion sensor is HIGH

print ("Rain detected",rain )

time.sleep(0.5)

except KeyboardInterrupt:

print ("Bye.")

Dockerfile:

FROM arm32v7/python:3-buster

COPY main.py ./

RUN apt-get update -y

RUN apt-get upgrade -y

RUN apt-get install python3-pip -y

RUN apt-get install -y apt-utils

RUN pip3 install --user python-dev-tools

RUN pip3 install --no-cache-dir OrangePi.GPIO

CMD ["python3", "main.py"]