TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

MD Tural Ismayilov 172681 IVSM

VIRTUAL IOT LAB FOR EMBEDDED SOFTWARE DEVELOPMENT FOR ESP32 AND RASPBERRY PI BASED DEVICES

Master's thesis

Supervisor

Juhan-Peep Ernits,

PhD

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

MD Tural Ismayilov 172681 IVSM

Virtuaalne IoT labor sardtarkvara arendamiseks ESP32 ja Raspberry Pi põhiste seadmete näitel

magistritöö

Juhendaja

Juhan-Peep Ernits,

PhD

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author:	MD Tural Ismayilov
Date:	9 August, 2020

Annotatsioon

Sardtarkvara koodiarenduskeskkonna pakkumine ilma riistvaralise sõltuvuseta Asjade Interneti-põhiste eksperimentide käitamiseks on keeruline ülesanne. IoT (Nutistu)-katsete tegemiseks on vähe virtuaalseid IoT keskkonnalaboreid, kuid enamik neist keskendus peamiselt võrgu simuleerimisele, mitte seadme emuleerimisele. Lõputöö eesmärk on uurida võimalikke viise, kuidas vältida IoT-rakenduste arendamisel füüsiliste seadmete kasutamise vajadust, emiteerides ESP32 ja Raspberry Pi riistvara ning pakkudes seejuures manustatud koodi arenduskeskkonda koos jäljendatud IoT-laboriga, mis võib olla algajale IoT kasutajale õpikeskkonnaks enne reaalsete füüsiliste seadmetega kokkupuutumist. Idee tõestusena võetakse levinud reaalmaailma probleemid Asjade Interneti arendusraamidest. Kasutatavateks töövahenditeks on QEMU (Quick Emulator), mida vajatakse ESP32 ja Raspberry Pi seadmete riistvaraemulatsiooniks ning Docker OS-tasemel virtualiseerimiseks. Lisaks sellele, toetutakse ESP32 rakenduste arendamisel ESP-IDF-i arendusraamistikule, mis annab võimaluse Raspberry Pi konfigureerimiseks ja kasutuselevõtuks. Kavandatud lähenemisviisil on mitmeid eeliseid: a) dokkerdatud rakendus võimaldab väljatöötatud rakendust hõlpsalt keskkondade vahel ümber tõsta; b) CLI (Command Line Interface) aitab hallata taustal töötavaid alamprotsesse, mis täiustab kasutajakogemust, jäljendades reaalse seadme arendamise kogemust ning aidates kasutajatel hõlpsalt liikuda reaalse füüsilise seadme arendamise juurde. Lähtekoodi skriptid kirjutati bash shell skriptikeeles Dockerfile, C/C++, Ansible Playbooks, Python, kasutades Microsoft Visual Studio Code keskkonda.

See lõputöö on kirjutatud inglise keeles ja on 45 lehekülge pikk, sisaldades 5 peatükki, 5 joonist ja 9 tabelit.

Abstract

Providing an embedded code development environment without any hardware dependency to run IoT related experiments is a challenging task. There are few virtual IoT environment labs to perform IoT experiments, but most of them mainly focused on network simulation rather than device emulation. The goal of the thesis is to look at possible ways to eliminate the need to use physical devices when developing IoT applications by emulating ESP32 and Raspberry Pi hardware and provide an embedded code development environment with an emulated IoT lab that can be a learning environment for new IoT users before starting with the real physical devices. As proof of concept, common real-world problems are taken from IoT development frames. Used tools are QEMU (Quick Emulator) for hardware emulation of ESP32 and Raspberry Pi devices, Docker for OS-level virtualization. Other than that ESP-IDF development framework is considered for ESP32 application development and Ansible for configuration and deployment of Raspberry Pi. The proposed approach has several advantages: a) Dockerized application allows developed application to be shuttled easily between environments; b) CLI (Command Line Interface) developed helps to manage subprocess running in the background that enhances the user experience by mimicking real device development experience to help the users to move to the real physical device development easily. The source code scripts were written in bash shell scripting language, Dockerfile, C/C++, Ansible Playbooks, Python using Microsoft Visual Studio Code environment.

This thesis is written in English and is 45 pages long, including 5 chapters, 5 figures and 9 tables.

List of abbreviations and terms

QEMU	Quick Emulator
OS	Operating System
IOT	Internet Of Things
RTOS	Real-time Operating System
CPU	Central Processing Unit
GDB	GNU Project Debugger
VM	Virtual Machine
CLI	Command Line Interface
UI	User Interface
IaC	Infrastructure as Code
WSL	Windows Subsystem for Linux
SBC	Single-Board Computer
PC	Personal Computer
LAMP	Linux, Apache, MySQL, PHP/Perl/Python
LFS	Large File Storage
CI	Continuous Integration
CD	Continuous Deployment

Table of Contents

List of Figures

List of Tables

1. Introduction

IoT stands for the Internet of things, but there are no common agreements on what does *"Thing"* mean in this context. [1] proposed a definition of IoT with *“A world where physical objects aree seamlessly integrated into the informationn networks, and where the physical objects can become active participants in the business process.”*. IoT devices as the internet of things are nonstandard computing devices. They connect with or without wire to a network and can transmit data. IoT is a trending research area that attracts the researchers, industrial and governmental sectors. Some organizations design and develop IoT based smart systems for monitoring and controlling the sub-station equipment. According to IoT Analytics estimates there were roughly 9.5 billion connected IoT devices at the end of 2019 which was in itself was considerably more than the forecast of 8.3B devices [2]. As more and more IoT devices make their way into the world, IoT systems present several unique challenges from scalability, security, and privacy to the integration of smart components [3]. Another challenge with IoT systems is related to developing and testing a system without having access to real IoT devices or sensors. The time takes to develop IoT systems increases considerably, as the embedded system involves complex tasks [4]. This makes a way for the development of platform-based design where it reduces the time spent on system software development and performance evaluation. Oce Technologies has introduced Software-In-the-Loop (SIL) simulation to build a virtual printer for the main purpose of developing hardware and software in parallel. This involves both hardware emulation and the addition of the emulation into SIL simulation. The hardware emulation is to build a virtual platform of all the essential functions and properties of a real hardware board [5].

The purpose of this report is to describe the results from a study to evaluate the potential system of recreating emulated RTOS based ESP32 together with Linux[1] based Raspberry PI on top of OS-level virtualization, where valid IoT development evaluations can be performed. To give an example of a similar system Tinkercad[2] can be considered which is a free, online 3D modeling program running on a browser, but is limited for emulating Arduino Uno together with few numbers of known sensors. Emulated ESP32 and Raspberry Pi IoT system will be a safe place for testing application code to check how it works before

[1]Linux: `https://en.wikipedia.org/wiki/Linux/`
[2]Tinkercad: `https://www.tinkercad.com/`

deploying on an actual device. The part of this research would be identifying possible common use cases of the ESP32 system together with Raspberry Pi so that better counter measurements can be developed and implemented for emulation. To give a possible use case, ESP32 can collect weather data from the sensors and can publish to the LAMP server that is running on Raspberry Pi. This system then may represent weather data with a graph that would be useful for understanding the recent changes in weather that is visually described in Figure 1.

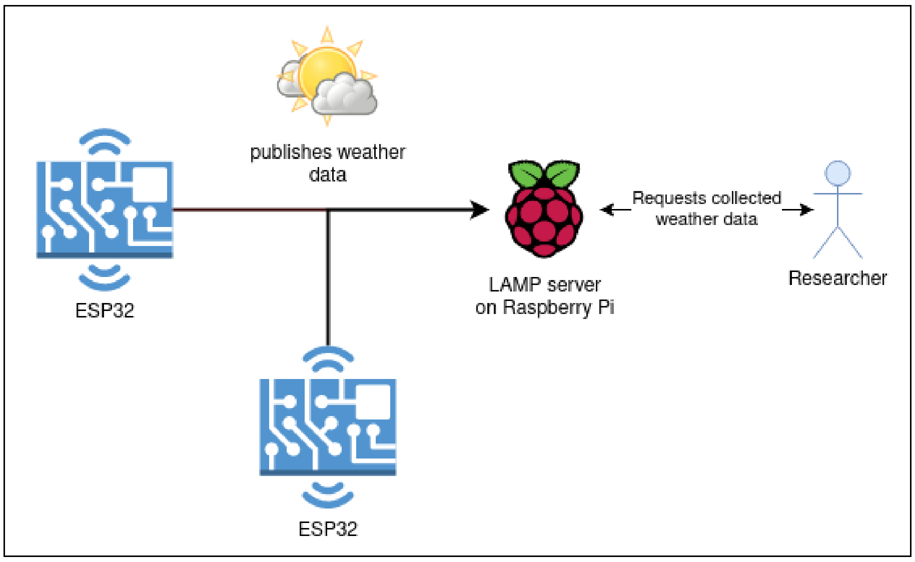


Figure 1. *Example usage for ESP32 and Raspberry Pi*

Another example could be having many houses with embedded environmental sensors that are distributed across cities and can send real-data to message broker from the single gateway and latter this data can be retrieved by stream processor to analyze and processed data could be stored in the database. Similar to this, MQTT *broker/client* example has been developed in the GitHub repository of the master thesis work[3] where MQTT client publishes message to *test/message* topic via MQTT protocol and as it's also subscribed to the same topic, it receives the same message.

1.1 Research motivation

As stated in Section 1 there were about 9.5 billion devices connected to IoT devices at the end of 2019. Given the increasing number of IoT devices making it accessible to everyone at low cost, especially to students who are mostly future users of these devices is one of the main challenges. Despite the existence of some solutions to help people to get started with IoT devices, they all have advantages and disadvantages, and they are mainly paid solutions.

[3]GitHub Repository for the IoT Lab Environment: `https://github.com/ismajilv/docker-emu/`

There would have some use-cases of IoT Lab Environment. To give example quality assurance engineers define and automate various types of tests in order to know whether a software build will pass or fails before actually deploying to production. In order to achieve this, they utilize CI and CD tools which are a big part of a culture in every big company that enables development teams to deliver applications more frequently and reliably. Jenkins[4], CircleCI[5], Travis CI[6] are the popular CI/CD tools. CD is the process of pushing application changes to delivery environments. A typical CD pipeline has build, test, and deploy stages. CI on the other hand is to combine all changes from multiple contributors into one software project. All CI/CD tools require environments to test the application code such as a server running Linux OS. But these environments don't cover ESP32, Raspberry Pi, Arduino Uno, and other alternative environments. To overcome this issue virtualized form of these devices can be used as an environment that would lead to faster testing, prototyping without needing actual devices but only Laptops. This approach would give the option not to own many instances of the same devices for every developer and as a result, would cut the cost

The developers of the IoT system would like to validate their developed embedded software application faster in an automated fashion so that they would do rapid development and testing. This would require virtualization of the resources where the users wouldn't need access to hardware to accomplish the tasks or run multiple tasks in parallel. This brings us to point of resource availability and as an example, stress testing of the IoT system with large networks can be given. These kinds of experiments require a bulk amount of hardware resources to be used which raises cost issues. As stated in the paper [4] turning the IoT system into virtualized one for prediction of the CPU usage saved them around $33,000.00. We can see resource availability, running cost, the time it takes to test the solution and maintaining such a system demands better work with IoT Systems.

In this thesis, we look into ESP32 and Raspberry Pi devices that have wide coupled usage in real-world IoT applications and are used by students, professionals. They will be assembled under a small IoT Lab Environment where users can practice their skills, automate some of the testing they do, and get started with these devices before working with them in real ones. One of the motivations in this research is to support open-source development of user-friendly IoT Lab Environment where IoT users can benefit from it.

[4]Jenkins: `https://www.jenkins.io/`
[5]CircleCI: `https://circleci.com/`
[6]Travis CI: `https://travis-ci.com/`

1.2 Research questions

The research questions follow the Trivium8[7] structure and the main question of thesis research is as follows: How to provide an embedded code development environment with an emulated IoT lab that can be used as a learning environment before starting with the real physical devices? In order to give the answer to the main research question is divided into the following hierarchy of sub-questions:

- **RQ-1: How to set up the IoT Lab Environment?**
  - RQ-1.1: What type of emulator is suitable for Raspberry Pi and ESP32?
  - RQ-1.2: What type of connection to use to establish network connectivity between Raspberry Pi and ESP32?
  - RQ-1.3: Which encapsulation tool to use to virtualize such a system so it can be shuttled easily between environments?
  - RQ-1.4: What IaC tools available to configure Raspberry Pi and which one is more appropriate?
  - RQ-1.5: What OS to choose for hosting QEMU inside virtualization?
- **RQ-2: How to get users to be familiar with IoT Lab Environment?**
  - RQ-2.1: What steps to take to mimic the experience working with physical devices and emulated ones?
  - RQ-2.2: Who are the interested parties to use IoT Lab Environment and what common examples can be solved with such a system that they encounter?

RQ-1 defines the way to build an IoT Lab Environment within the proposed solution by describing the technical requirements to run such a lab and maintain. **RQ-2** describes the use cases of this IoT Lab environment and how to make it more user friendly by eliminating the difference between working real and emulated devices and showing the solution for real-world problems.

1.3 Thesis structure

This section describes the chapters' contents of this thesis. The introduction provides aims, motivation, and objectives of the research. The problem to be solved is described here.

Chapter 1 provides the aims, motivation, objectives of the research, and the problem solved.

[7]Trivium8: `http://www.triviumeducation.com/`

Chapter 2 describes the technological background and tools used to develop IoT Lab Environment.

Chapter 3 gives information about details on the development of the proposed solution for the Lab.

Chapter 4 talks about the feedback by experts who tested the IoT Lab Environment.

Chapter 5 summarise the thesis and answer the main research questions.

2. Background

The following sections provide technological background and tools used to develop IoT Environment Lab. First in Section 2.1 proposed solution and in Section 2.2 the requirements for it is discussed, followed by Section 2.3 is a brief information about Docker, QEMU, GDB and there use case in our domain. Section 2.4 talks about emulated ESP32 and Raspberry Pi devices and in Section 2.5 all other supporting software such as application-deployment and development tools are described. This Chapter also gives the answers to RQ-1.1, RQ-1.3, RQ-1.4, RQ-1.5, and Chapter 2.6 says the final words.

2.1 Proposed solution

As mentioned earlier, in this thesis research we will focus on emulating popular ESP32 and Raspberry Pi devices. We call the whole system as IoT Environment Lab that will provide the way to run the most common tasks with real ESP32 and Raspberry Pi in an emulated environment with little or no difference between the experience of working with emulated and physical devices. For emulation of ESP32 and Raspberry, Pi QEMU will be used that will run on top of Ubunutu 16.04 inside separate Docker Containers build and managed by docker-compose, one for ESP32 emulation another one for Raspberry Pi. Network connectivity between ESP32 and Raspberry Pi will be established by Ethernet emulation over TCP/IP. Raspberry Pi container will have *detect_gpio_changes* executable that will perform analysis of GPIO state change to inform the user via CLI. To improve the user experience, CLI will be developed that will manage the IoT Environment Lab and application deployment and monitoring of ESP32. For getting started guides, common use case examples of the coupled system will be developed with instructions. Having a dockerized application lets us easily add new containers when new device emulation is needed that solves the *heterogeneity* challenge with IoT mentioned in [4], so we can scale up the number of Containers and have a realistic system with the network connectivity. The high-level graphical form of the proposed solution is given in Figure 2.

2.2 Requirements

In order to give the description of the functionalities and the features of the system, requirements are developed that specify what features the system should include and how those features should work together. Those requirements for different parts used in the development of IoT Environments Lab are given in Table 1.

Table 1. *Requirements for IoT Environment Lab*

Part	**Requirements**
Emulation	To be open-source to make it possible to customize for certain purposes Support Xtensa and ARMv8 architecture
Virtualization	Be shuttled easily between environments Containerized so isolated from other applications Support scaling up the instances
CLI	Support most of the functionalities of IoT Environment Lab
OS	Support the system used for Virtualization
Development and Deployment	Give the highest fidelity, be well-known, and well documented

The requirements will help us to develop the system with the target in mind and help the reader what are the expectations from the given application.

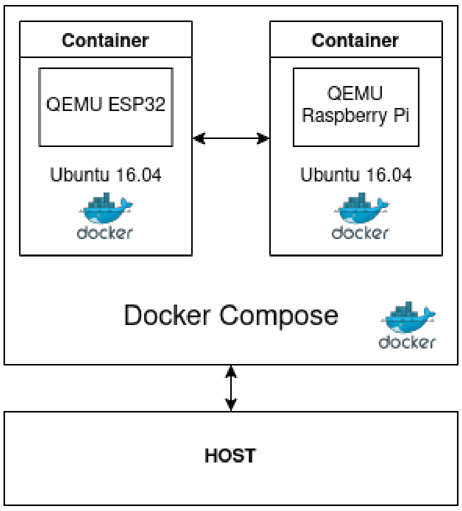


Figure 2. *Basic graphical overview of proposed solution*

2.3 Virtualization selection

2.3.1 QEMU

QEMU is an open-source machine emulator similar to VirtualBox[1] or VMware[2] and it is able to emulate operating systems and get near-native performance by executing the guest code directly on the host CPU, so it can execute the guest operating systems code on the real CPU like if it was installed here [6].

[1]VirtualBox: `https://www.virtualbox.org/`
[2]VMare: `https://www.vmware.com/`

QEMU is a functionally accurate virtual emulation platform that can be used to develop, debug, and run the software as it would be done on an actual development board using exactly the same development tools. It executes code on instruction level, rather than clock cycle level, which makes QEMU really fast. If used with KVM it can run virtual machines at near-native speed. QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another. For example, the QEMU simulation engine will take the arm v8 instruction or code and operands and will convert them on the fly to the equivalent x86 instruction which will execute on your laptop's processor. The result of the instruction execution the operand value and the system state will be reflected back into the arm create execution context which is a very powerful approach that takes full advantage of the large powerful processor. QEMU is different than Docker in a way that it enables to run software emulation fully without needing a host kernel driver. It means QEMU doesn't need a host virtualization support.

QEMU as an open-source emulator with performance advantage and ability to support nearly 50 different machines including Xtensa[3] and ARM[4] boards which are compatible with the ESP32 and Raspberry Pi devices emulation respectively makes it an easy choice as an emulator to the answer for the type of emulation we needed, therefore answer our RQ-1.1.

2.3.2 GDB

In the early days, in order to find a fault in the software, developers needed to "desk check" the software by reviewing source code. Another method was adding a strategically-placed serial output message in order to catch fault upon execution. The modern development environment provides a new approach, which is using interactive tools to run the code under a debugger.

GDB is the GNU Project Debugger is a powerful all-purpose debugger tool for C along with many other languages like Pascal, Fortran. One of the main features of GDB is to be able to debug programs remotely via a serial port, network connection, or some other means. It runs on many popular UNIX and Windows-based operating systems and its a command-line tool which means you interact with it in the terminal issuing command via the keyboard instead of clicking buttons with your mouse. The debugger can also evaluate arbitrary C expressions such as function calls on the remote target [7]. We will use GDB to debug the application in the emulated ESP32 device.

[3]QEMU Xtensa `https://wiki.qemu.org/Documentation/Platforms/Xtensa`
[4]QEMU ARM: `https://wiki.qemu.org/Documentation/Platforms/ARM`

2.3.3 Docker

Docker is a software development platform and a virtualization technology that makes it easy to develop and deploy applications inside packaged virtual containerized environments. Containers running on have isolated CPU processes, memory, network resources and can emulate hardware, and boot an entire operating system. Unlike VMs, the resources are shared with the host. Traditional VMs have a hypervisor that occupies about 10 to 15 percent of the capacity of a host, but having no hypervisor lets Docker run hyperscale numbers of containers on a host container because sit directly on top of the operating system that gives answer to RQ-1.3 [8]. Using the Docker approach gives us rapid instantiation, customizability, and portability of the components and having separate containers for ESP32 and Raspberry PI eliminate threads sharing the same resources inside a single container that makes way to faster emulation of the IoT devices. Having the Docker approach also lets us scale up the number of instances easily that is useful for testing real-world scenarios.

2.4 Emulated physical tools

2.4.1 ESP32-DevKitC

To expand the area of IoT applications, low-cost, low-power devices are required. ESP 32 has an integrated Wi-Fi and Bluetooth that uses the ESP 32 chip made by Chinese Manufacturer Espressif Systems. The ESP32 system on the chip shipes with 5 mm x 5 mm sized QFN packages and The ESP32-DevKitC is on other hand is board that uses ESP32 chip [9]. ESP32 supports a SDIO (Secure Digital Input Output) card is an extension of the SD specification to cover I/O functions that supports UART, SPI, I2C, PWM serial data communication protocols and has dual or single core versions of Tensilica Xtensa LX6 microprocessor [10]. Figure 3 shows the ESP32-DevKitC board.

2.4.2 Raspberry Pi

Raspberry Pi is a deck card-sized, cheap system on a chip SBC and recently it has become one of the most popular platforms to host software projects. It costs around 35$ (USD) and has different variations depending on need. As a small computer, it has everything to connect to a monitor, keyboard, mouse, Internet. Most of the modern computers on the market run Windows or Mac operating systems but the Raspberry Pi runs on Linux as its desktop operating system, to be specific it’s a version of Linux called a Raspbian designed

Figure 3. *ESP32-DevKitC board*

specifically for the Raspberry Pi. Most importantly, Raspberry Pi has general-purpose input-output pins or GPIO pins that are electrical signals to control IoT devices such as LED lights, radio switches, audio signals, even LCD. It has a 64-bit quad-core ARM Cortex-A53 processor with a clock speed of 1.2GHz and supports full HD output over HDMI, onboard Wireless LAN, and Bluetooth, 3.5-millimeter audio jack and micro USB [11].

2.5 Supporting tools

2.5.1 Ansible

Ansible is an IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs. Ansible doesn't use agents and no additional custom security infrastructure is needed, so it's easy to deploy - and most importantly, it uses a very simple language (YAML, in the form of Ansible Playbooks[5]) that allow you to describe your automation jobs in a way that approaches plain English. Ansible deploys modules to nodes that are resource models of the desired state of the system, usually over SSH, and when finishes, remove them. This step requires no database, servers, or daemons [12].

[5]Ansible Playbooks: `https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html`

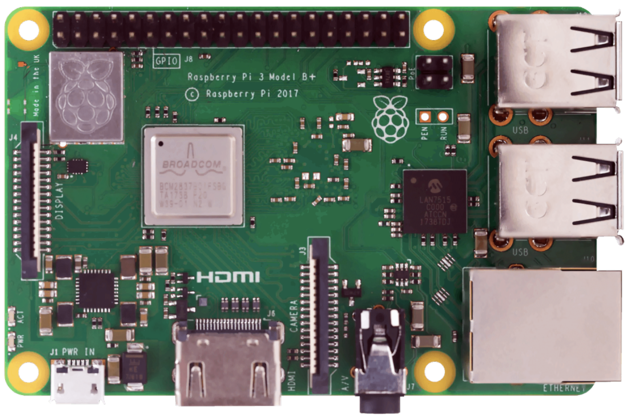


Figure 4. *Raspberry Pi board*

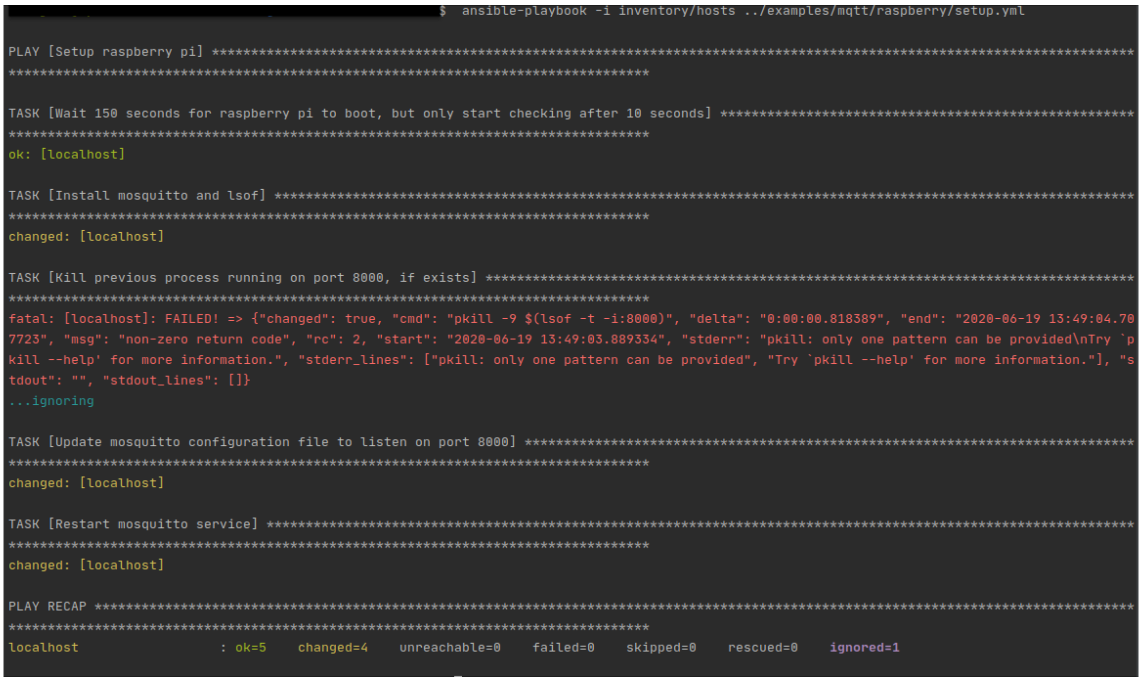
3 of the available most famous IaC tools to automate configuration are Ansible[6], Jenkins[7] and Puppet[8]. We choose Ansible over other solutions because of an easy learning curve, having no agents on the node, simplified automation that answers to RQ-1.4.

Ansible variables can be assigned in simple text files using subdirectory called 'group_vars/' or 'host_vars/' or directly in the inventory file. Ansible uses Playbooks to execute the commands. They contain the particular steps that the user wants to perform. Playbooks are one of the core features of Ansible and tell Ansible what to execute. In the IoT Environment Lab, we used password authentication to configure Raspberry Pi and example Ansible Playbook format taken from IoT Environment Lab source code [13] given in Appendix 1. Figure 5 illustrates how to setup MQTT broker[9] in Raspberry Pi and *ansible-playbook* command outputs.

2.5.2 CLI for IoT Environment Lab management

A CLI is a text-based UI that accepts text input to execute operating system functions. Shell handles CLI and they are separated from the underlying OS kernel. To use a shell, one must know the syntax of a scripting language [14]. Today most users prefer the GUI offered by operating systems because they are being intuitive, easy to learn, requiring

[6]Ansible: `https://www.ansible.com/`
[7]Jenkins: `https://www.jenkins.io/`
[8]Jenkins: `https://puppet.com/`
[9]MQTT broker: `http://mqtt.org/`



```
[redacted]$ ansible-playbook -i inventory/hosts ../examples/mqtt/raspberry/setup.yml

PLAY [Setup raspberry pi] ****************************************************************************************************************
**********************************************************************************

TASK [Wait 150 seconds for raspberry pi to boot, but only start checking after 10 seconds] ***********************************************
**********************************************************************************
ok: [localhost]

TASK [Install mosquitto and lsof] ********************************************************************************************************
**********************************************************************************
changed: [localhost]

TASK [Kill previous process running on port 8000, if exists] *****************************************************************************
**********************************************************************************
fatal: [localhost]: FAILED! => {"changed": true, "cmd": "pkill -9 $(lsof -t -i:8000)", "delta": "0:00:00.818389", "end": "2020-06-19 13:49:04.70
7723", "msg": "non-zero return code", "rc": 2, "start": "2020-06-19 13:49:03.889334", "stderr": "pkill: only one pattern can be provided\nTry `p
kill --help' for more information.", "stderr_lines": ["pkill: only one pattern can be provided", "Try `pkill --help' for more information."], "s
tdout": "", "stdout_lines": []}
...ignoring

TASK [Update mosquitto configuration file to listen on port 8000] ************************************************************************
**********************************************************************************
changed: [localhost]

TASK [Restart mosquitto service] *********************************************************************************************************
**********************************************************************************
changed: [localhost]

PLAY RECAP *******************************************************************************************************************************
**********************************************************************************
localhost                  : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=1
```

Figure 5. *Example ansible-playbook command output*

recognition rather than recall, and being good for exploratory analyses and graphics. But as the user of IoT Environment Lab is regular, not occasional, utilizing of CLI is preferred rather than GUI as they need less overhead than GUI, and can be enhanced easily as the source code is given in [13]. More information about the developed emu CLI is in Chapter 3.

2.5.3 ESP-IDF, ESP32 Development Framework

Choosing the right technology to develop the platform is one of the main tasks. To develop in ESP32 the knowledge of C and C++ is needed. ESP32 code can be developed with *arduino-esp32*[10] or *esp-idf*[11] and both frameworks are developed by Espressif System, the same company makes the ESP32 chip. The difference is, Arduino-esp32 is just a wrapper around the ESP-IDF and simpler to work with. But for more serious embedded development using an ESP-IDF framework is suggested. In this master thesis, an ESP-IDF development framework is supported by CLI.

2.5.4 Ubuntu Server

Ubuntu is an open-source software operating system mostly composed of free and open-source software that runs from the desktop to the cloud. It is an ancient African word,

[10]Arduino ESP32 `https://github.com/espressif/arduino-esp32`
[11]ESP-IDF: `https://github.com/espressif/esp-idf`

meaning "humanity to others". Ubuntu aims to provide easy-to-use, reliable, and open-source Linux distribution that is based on Debian Linux [15]. Ubuntu 16.04 version is a long-term support release based on Linux 4.4. Ubuntu makes setting up Docker so much easy and it works with nearly any hardware or virtualization platform that gives an answer to RQ-1.6 [16].

2.6 Conclusion

This chapter was dedicated to the choice of virtualization selection together with the proposed solution. Most of the selection criteria based on the simplicity of usage, popularity and compatibility of the system with other parts. A small description of the system is given for each selection. We choose QEMU as an emulator for Raspberry Pi and ESP32 devices, and Ubunutu 16.04 Server to host the QEMU application and Docker as a virtualization platform that is managed by docker-compose. To automate the configuration of Raspberry Pi Ansible is picked and emu CLI has been developed to manage the IoT Environment Lab and ESP-IDF framework to develop in the ESP32 device.

3. System integration of IoT Environment Lab

The following sections provide details on the development of the proposed solution for the IoT Environment Lab. First, in Section 3.1 setting up the IoT Environment Lab is discussed. Next, in Section 3.2 running device on Qemu and how it is wrapped inside Dockerfile for ESP32 with the answer to RQ-1.2 and followingly in Section 3.3 same for the Raspberry Pi is talked about. In Section 3.4 combining both ESP32 and Raspberry Pi containers under docker-compose is given. Section 3.5 describes the use-case of developed CLI to manage deployment and IoT Environment Lab which answers RQ-2.1. Section 3.6 gives information on the Ansible inventory file and Section 3.7 provides an overview of examples and getting started with IoT Environment Lab, Section 3.8 discuss what could be done for future, Section 3.9 gives information on the test developed and Section 3.10 adds related work part. Lastly, Section 3.11 wraps up everything described in this Chapter. This Chapter also answers RQ-2.2.

3.1 Setup guide for the IoT Environment Lab

Even before cloning GitHub repository of the IoT Environment Lab and setting it up certain release versions of programs are required which is described in Table 2. The *git* and *git LFS* needed to clone the repository as modified Raspberry Image has a bigger size and only content of this file can be stored in GitHub but file itself in Git LFS.

Table 2. *Versions*

Program	**Version**
git	*
git LFS	*
ansible	>= 2.7
ESP-IDF	v4.2
Docker	>= 19.*
docker-compose	>= 1.25
click python package	7.21
sshpass	*
python	>= 3.7

The IoT Environment Lab should be run on the Linux, better in Ubuntu 18.04, as the experts faced some issues with the other versions of Linux while setting up ESP-IDF or you may use PlatformIO[1] to setup ESP-IDF, but we have not tested it yet. Creating a bootable Ubuntu 18.04 USB stick would be easy option. To set up, first clone the GitHub repository of this thesis work with the command described in the code below:

```
$ git clone https://github.com/ismajilv/docker-emu
```

Then build the IoT Environment Lab with the command below in the root directory of the project:

```
$ sudo docker-compose build
```

The step builds the Raspberry Pi and ESP32 Containers and sets up the network of the Lab with Docker Compose and the last thing that needs to be done is to setup *emu cli* which is described more in Chapter 3.5. For that reason run:

```
$ cd cli_interface
$ python3 -m pip install --editable .
```

This should setup *emu-cli* interface on your Computer which we talk more about in Chapter 3.5 and the examples to run on the IoT Environment Lab is described in Chapter 3.7.

3.2 Running ESP32 on Qemu and wrapping it in Dockerfile

The CPU of ESP32 microcontroller is developed by Tensilica which is known for its customizable Xtensa configurable processor microprocessor core and starting from version 1.0 QEMU provides Xtensa architecture emulation [17]. There are 2 versions of modified QEMU to emulate ESP32, one is patched by Espressif who also makes ESP32 chip[2] and Ebiroll[3] ones. Espressif version is more accurate than Ebiroll's one but lacks some features like 1306 emulation which is OLED emulation, but Ebiroll's intention is to use Espressif one in the future [18]. We have been tested both QEMU by running different examples, and Espressif's one seems more stable, therefore for this and other reasons mentioned above in this thesis work, Espressif's QEMU modification has been chosen as an emulator for ESP32. To emulate ESP32 *qemu-system-xtensa* program has been built from the source code of fork of QEMU with patches for Espressif chips support that can run ESP-IDF built applications and placed in the Github repository of this thesis. When compiled ESP-IDF built applications, it creates a bootloader, partition table, and application itself which then combined into *bin* file. This executable bin file is loaded on QEMU startup. But as the IoT Environment Lab supports *flashing* and before *flash* operation QEMU needs to be

[1]PlatformIO: `https://platformio.org/`
[2]Espressif patched QEMU: `https://github.com/espressif/qemu/`
[3]Ebrioll patched QEMU: `https://github.com/Ebiroll/qemu-xtensa-esp32`

started, empty *flash_image.bin* file has been created and placed in the Github repository. Appendix 2 shows how *bin* files is build that creates create a 4MB empty file filled with *0xff*. It is possible to monitor and flash into ESP32, but then QEMU needs to be started with different strapping mode. To overcome this and other issues emu CLI is built and more information is given in Section 3.5. QEMU emulation with Espressif patches has enabled Ethernet emulation, but it needs an ESP-IDF branch that has Opencores Ethernet MAC driver support that is shipped after with ESP-IDF version 4.2. When compiling ESP-IDF built application, enable *CONFIG_EXAMPLE_CONNECT_ETHERNET=y* and *CONFIG_EXAMPLE_USE_OPENETH=y* to use the Ethernet emulation and use *open_eth* network device. Detailed information about used options with ESP32 QEMU emulation is described in Table 3

Table 3. *Used ESP32 QEMU emulation options*

Option	**Description**
-nic user,model=open_eth,id=lo0	use *open_eth* network device
-gdb tcp::1234	start with GDB server, waiting for connection on port 1234
-serial tcp::5555,server	output serial on port 5555, wait for connection
hostfwd=tcp::3333-:3333	enable forwarding on port 3333, use for socket conection
-global driver=esp32.gpio, property=strap_mode,value=0x0f	set strapping mode

and strapping moed choice in Table 4.

Table 4. *ESP32 QEMU emulation strapping mode*

Strapping mode value	**Description**
0x12	flash boot mode (default)
0x0f	UART-only download mode

The strapping mode value is chaged depending on if ESP-IDF compiled app should be flashed or monitored with emu CLI usage. During the talk with an expert from the Espressif, it is clarified that GPIO functionality in Espressif fork of QEMU is not implemented and the Wi-Fi MAC hardware register interface is not documented by Espressif so that has ruled out of the possibility of writing an emulator for it. Therefore, an Ethernet connection is chosen to connect ESP32 and Raspberry Pi instead of the Wi-Fi which answers RQ-1.2. During QEMU startup if (-bios) argument is not given ESP32 ROM code will be loaded, which is already presented in GitHub repository[4] and provided by Espressif.

It is sufficient to run ESP-IDF compiled application on QEMU but to fully virtualize the

[4]ESP32 rom: `https://github.com/ismajilv/docker-emu/blob/master/esp32/build/qemu-system-xtensa/esp32-r0-rom.bin`

system the application is shipped in Docker Container. Dockerfile builds QEMU on top of Ubuntu 16.04 and exposes 3 ports. Table 5 gives information numbers for each port and their descriptions.

Table 5. *ESP32 configured ports*

Port	**Description**
1234	To connect GDB
3333	To establish socket connection
5555	To output serial and conect `idf.py monitor`

3.3 Running Raspberry Pi on Qemu and wrapping it in Dockerfile

One of the main purposes of the IoT Environment Lab is to build in a way that mimics working with real physical devices. Raspberry Pi has a GPIO subsystem that has digital signal pins whose behavior includes whether PIN acts as an input or output and is controllable by the user at run time. Users can set the voltage high or low depending on the need with modified wiringPi library functions. Currently, QEMU does not support GPIO functionality of Raspberry Pi board, therefore modified QEMU and modified version of library wiringPi[5] is used. WiringPi is a PIN-based GPIO access library written in C for the BCM2835, BCM2836, and BCM2837 SoC devices. So it is compatible with the Raspberry Pi board we are using. This modification is described in the paper [19]. Modified QEMU has been built[6] and placed in the Github folder of IoT Environment Lab together with *detect_gpio_changes* executable file that is used by Docker. It is compiled from C code that detects the GPIO state change and outputs it in form of serial output. Source code is taken from [19]. To lessen the build time for docker-compose, lite version Debian “Jessie” Release image has been used and modified to enable emulating Jessie image with 4.x.xx kernel, ssh and support GPIO subsystem. To avoid executing an unimplemented SETEND instruction need to comment out every entry in *./etc/ld.so.preload* file and *./etc/fstab* after mounting Jessie image [20]. To enable the ssh placing empty file named *ssh* in the root folder should be enough [21]. And the QEMU modification is described in the paper [19]. Table 6 gives information numbers for each port and their descriptions.

Table 6. *Raspberry Pi configured ports*

Port	**Description**
2222	Forward connection over the tunnel back to the Pi on port 22
8000	Service port

Configuration of the emulated Raspberry Pi is more close to Raspberry Pi version 1.

[5]Wiringpi: `http://wiringpi.com/`

[6]Raspberry Pi build: `https://github.com/ismajilv/docker-emu/blob/master/raspberry/build`

3.4 Wrapping Dockerfiles into docker-compose

Compose is a tool for defining and running multi-container Docker applications and it is a YAML file defining services, networks, and volumes. It connects the Raspberry Pi and ESP32 containers. Version 3 of the Compose file format has been used. In our usage, the Compose file defines port forwarding between containers and to the outside world. Docker Compose file has been written in a way that, enables scaling of emulated ESP32 devices which means we can run *n* number of ESP32 devices simultaneously, but only one Raspberry Pi. It will set up the network and will let the devices talk to each other. It enables users to easily test out systems containing a scaled number of devices. After the feedback on the issue related to failed start of IoT Environment Lab as Raspbian image not found because of the git LFS was not installed, Docker Compose file updated to inform the user with error in this case.

3.5 Emu CLI usage

Emu CLI is developed in Python programming language to address the few issues related to IoT Environment Lab, also as an answer for RQ-2.1. This CLI is built on package called Click[7] and can be installed with pip which is the package installer for Python. Emu CLI is compatible with the Python version of 3. Click is a highly configurable Python package for creating beautiful CLI applications with as little code as possible. The CLI functions are described in Appendix 3. One of the most important features of CLI is to restart QEMU emulation for ESP32 with different strapping mode. ESP-IDF development framework supports the flashing and monitoring of the ESP32 device. While emulating ESP32 device, with *strap_mode=0x0f* option, the emulated chip is *download mode*. In a real development board, *esptoolpy* resets the chip into *flash boot* mode by DTR and RTS UART signals after loading the program. But with QEMU emulation, esptool communicates with the emulated chip over a TCP socket, so there are not DTR and RTS signals to toggle, and the emulated chip stays in the *download mode*. To run the program after flashing, QEMU needs to be restarted with different strapping mode, so it means someone needs to modify the strapping mode and reset QEMU between these commands. This is done by CLI that makes possible to use *emu monitor* after *emu flash*. *emu flash* command also builds ESP-IDF built application by combining bootloader, partition table, and application. Another important future is after scaling the ESP32 devices to certain number, you can give the *id* of the device to the CLI command that can run the desired command on a specific device. We can also see port information for a given device with *emu eport* command and can connect to the socket if enabled via *esocket*. The *echo* example enables using socket

[7]Click package: `https://click.palletsprojects.com/en/7.x/`

for UART like communication[8]. Another functionality of CLI is to show the output of Raspberry Pi GPIO states changes that is done by *emu rgpio* command and source code is taken from [19]. It also lets the users *ssh* into Raspberry Pi, see the *logs*, *start/stop/restart* the IoT Environment Lab that makes working with it intuitive. CLI's function is described in the Table 7.

Table 7. *Emu CLI functions*

Function	Description	Option	Option Description
start	Start IoT Environment Lab	*--scale-esp32 n*	Run *n* instances of ESP32
stop	Stop IoT Environment Lab		
restart	Restart IoT Environment Lab		
ssh	SSH into raspbian		
monitor	Same as idf.py monitor	*--id n*	monitor *nth* device
flash	Same as idf.py flash	*--id n*	flash into *nth* device
log	Log output	*esp32 --id n* or *raspberry_pi*	*nth* device
eport	Get port information of ESP32	--*id* n	*nth* device
esocket	Connect to socket port of ESP32	--*id* n	*nth* device
rgpio	See raspberry pi gpio state		

Emu CLI can scale up the IoT Environment Lab to have *n* number of ESP32 instances with *--scale-esp32 n* option. Example for such command that runs one instance of Raspberry Pi and two instances of ESP32 is given below:

```
$ emu start --scale-esp32 2
```

Then we can use option --id n with *emu eport/esocket/flash/log/monitor* commands to work with specific ESP32 device. Here, the *id* is assigned to ESP32 devices by 1, 2, ... n, depending on the number of instances running and default for *--scale-esp32* option is 1.

3.6 Ansible to configure Raspberry Pi

Most of the examples related to Raspberry Pi needs to automate the configuration of the device. Ansible works against "hosts" in our infrastructure at the same time, using a list or group of lists known as inventory. Therefore *ansible.cfg* file is created in the *inventory* folder of ansible. To enable communication with the node via ssh on port 2222, ansible inventory file has been added with the content described in Appendix 4.

[8] *echo* example: https://github.com/ismajilv/docker-emu/tree/master/examples/echo

3.7 Examples

The Github repo [13] of IoT Environment Lab comes with some examples that are shortly described in Table 8.

Table 8. *Examples description*

Example	Description
echo	To echo user input to ESP32 from Raspberry Pi board
mqtt	Mqtt client in esp32 connected to broker in raspberry pi
hello_gdb	Simple hello world demonstration with use case of GDB
raspberry_gpio	Usage of raspberry pi GPIO subsystem

To get familiar with IoT Environment Lab start with the examples[9]. In *echo* example, 2 numbers of ESP32 devices launch a TCP server that listens on a socket for incoming characters. After receiving character data from the terminal, sends it to the server running on Raspberry Pi via a POST request. Server running on Raspberry Pi keeps request data in the list and it can be obtained by GET request. In *mqtt* example MQTT broker runs on Raspberry Pi. ESP32 MQTT client publishes a message to test/message the topic via MQTT protocol. Because it's also subscribed to the same topic, it receives the message itself. *hello_gdb* starts a FreeRTOS task to print "Hello World" and also demonstrate usage of GDB with QEMU. And *raspberry_gpio* example shows how to use the Raspberry PI GPIO subsystem with the IoT Environment Lab. Each example contains README with separate *esp32* and *raspberry_pi* folders that describe how to run them.

The whole system has been tested on Ubuntu 18.04 and there are some known issues with Ubuntu 20.04 such as building up ESP-IDF from scratch and in WSL2 valid shell script giving syntax errors[10].

3.8 Further work on GPIO subsytem of ESP32

Espressif version of QEMU does not include patch for GPIO subsystem and has been left empty in *hw/gpio/esp32_gpio.c* file of *esp-develop* branch. It rules out the possibility to use GPIO functionality of ESP32 for now, but further work is needed to implement it. If done, then other peripherals such as I2C, SPI can be implemented with GPIO bit-banging. Also, the starting address of the *RPI_GPIO* device's memory region would need to be updated so that it can match the correct address for Raspberry 3. But this version in the thesis should be enough to run most of the software except the ones very specific to the newer ARM processor core families used in Raspberry Pi 2 and 3. The additional change

[9]Examples: `https://github.com/ismajilv/docker-emu/tree/master/examples`
[10]WSL issue: `https://github.com/Microsoft/WSL/issues/2107`

would be having a bridge on the interface *eth0* between Host and Guest QEMU instead of port forwarding that would add utilizing more ports than predefined ones.

3.9 Testing of emu CLI

Tests are one of the important parts of any Software development cycle. In order to validate the quality of code, provide the visual feedback, and clarify the requirements tests are developed. Software testing is categorized into functional and non-functional testing. Based on the requirements criteria functional testing tests the activities or operations of the application. On the other hand, non-functional testing checks the behavior of the application. In the GitHub repository of this master thesis work, we have used *Unit testing* to test the individual units in the emu CLI code[11].

3.10 Related Work

To enable one computer system to behave like another one Emulation is needed. This approach enables running OS in a virtual environment or playing Playstation games on PC. For example, as a network administrator emulation is helpful when running an embedded operating system from a computer that doesn't normally support the operating system which otherwise they would need separate machines to run a different OS. To cope with IoT Systems better and to address numerous issues with real IoT devices, some virtualized systems have been proposed. One of the customizable virtual labs called EMU-IoT is used to model heterogeneous IoT networks to investigate how IoT applications will perform on a specific network [4]. Heterogeneous in this context means the ability to extend and easily add new IoT devices without disturbing the stability of the existing network. In order to create this IoT network, 3 separate parts have been added to the EMU-IoT: the IoT producers, the IoT gateways, and IoT applications. To achieve the sensor's emulation, a program in C has been developed that emits integer values every second via an HTTP POST request to the virtual gateway which in itself cannot be considered as fully hardware emulation. By monitoring the application in terms of load tests CPU utilization is measured for the running IoT application together with running sensors and this collected data then is used to build a performance model to predict the CPU utilization that it would reach depending on the number of IoT devices. For example, they have computed a regression function to make a prediction about the number of devices required in order to CPU resource hit 60-70% capacity usage. This paper described the study mainly focusing on stress test IoT applications, rather than IoT device emulation. One of the take-away is the design pattern which is to separate the resource as much as possible such as using a

[11]Testing of emu CLI: https://github.com/ismajilv/docker-emu/blob/master/cli_interface/test.py

containerized approach to comply with heterogeneity.

In 2015, an IoT emulation environment with COOJA has been proposed which can emulate Contiki systems but cannot emulate for example Raspberry Pi [22].

Many proposed solutions look at the process of IoT Gateway emulation which is the communication between sensors by means of general-purpose Internet protocols, but not the embedded software development side. This would require the emulation of IoT devices and the integration of different parts together. The IoT Environment is a step toward overcoming this problem and helping to improve IoT skills in an easy way for students and IoT developers to test out scripts before deploying to actual devices, learn about how to develop a software for embedded devices, evaluate the performance of the software and do stress testing for the application they want to deploy. In a nutshell, IoT wants to connect all potential objects to interact with each other on the internet in a secure means, therefore establishing a connection between IoT devices in the virtualized IoT lab is one of the requirements.

3.11 Conclusion

Throughout this chapter, we state design for IoT Environment Lab mainly taking into account ESP32 and Raspberry Pi. Having QEMU as the emulator is an easy choice as it is open source and fast compared to other competitors. We choose Espressif patched version of QEMU emulation for ESP32, as it has better stability and very good documentation of usage instead of Ebiroll's one. For the QEMU emulation of Raspberry Pi, we take advantage of [19], as it provides GPIO functionality, but with additional modifications to Jessie image to run it under QEMU with SSH support and eliminating error proneness. To automate the configuration of Raspberry Pi Ansible IaC has been chosen over other solutions that are really simple to use. Solutions for common problems have been added to the examples folder in the Github repository of the project that also helps users to get familiar with IoT Environment Lab. To mimic the experience and have helper functions, emu CLI has been developed with documentation and added to the Github repository, which also solves the problem of starting ESP32 QEMU application with different strapping modes to enable using *flash* and *monitor* at the same time.

4. Evaluation

Evaluation is an essential activity to measure the outcome of the proposed solution. We want to see how well the proposed functionality in artifact fits the real-world scenarios, and what are the user's expectations and benefits it carries for potential.

4.1 Evaluation method

It was hard to reach to more people to try the lab but having feedback from knowledgeable experts who understand the concept very well would be considered as an important addition to further evaluation of proposed solution with addition of the some modifications if needed based on feedback. There are 2 main roles in proposed artifact: *users* and *service* which in itself is consisted of 2 parts: *raspberry* and *esp32*. Experts perform various tests and try to implement their day-to-day activities in the IoT Environment Lab to measure the wellness of proposed artifact. To get the answers to necessary questions Questionnaire Form given in Appendix 5 has been formed and sent to experts with additional Github URL, so they can try the IoT Environment Lab and write their feedback.

4.2 Evaluation result

Overall, the proposed solution was liked by experts. One of the main issues was setting up the IoT Environment Lab. The issues were related to with different OS and even with different versions of same OS some packages there were bugs and some package needs to be installed additionally. 2 of these bugs shortly described in Chapter 3. Other than that certain versions of the packages needs to be used that was not clarified very well before and therefore to address all these issues Docker Compose build file has been updated and *Setup guide for the IoT Environment Lab* Section is added in Chapter 3 with the required packages to install. Having the summary table where it would clearly what functionality is missing from the virtualized environment compared to actual hardware is suggested and the table is added to *Conclusion* Section of Chapter 5. To address the experience working with physical and emulated devices in terms of user experience, deployment, and maintenance it is noted that using IoT Environment lab was a seamless experience and for many things the physical boards were not necessary and emu CLI had enough functions to perform most of the basic tasks, but it would be great to see the commands used under the emu CLI for debugging reason which resulted in updating emu CLI commands to echo sub-commands. Emulating sensors and actuators, on the other hand, may feel different but it would need further addition to the emu CLI. To answer the lab's point to perform their day-to-day activities with the ESP32 or Raspberry Pi, experts mentioned that it was pretty convincing already - fast testing and prototyping, no need to carry the devices around, and the possibility to test embedded software in CI setting, accessibility of the system to a large number of students and easy way to test out systems containing a scaled number of the device. The primary application of the system will be in teaching and learning. To point what addition would make sense to IoT Environment Lab, it was noted that having emulation of sensors and actuators would be a huge plus. After this feedback, GPIO subsystem emulation has been added as a starting point to Raspberry Pi emulation that is discussed in Chapter 3. Another expert suggested having a way to manage the number of devices and command line support for addressing a particular device in a larger network of devices. To address this feedback, emu CLI and Docker Compose file are updated to support scaling up the n number of ESP32 devices with Raspberry Pi including network setup.

Another thing is to have an installation script that can be considered as further work keeping in mind it is not easy to make one general install script for different OS. This could be solved by PlatformIO[1] that handles setting up desired framework for ESP32.

[1]PlatformIO: `https://platformio.org/`

5. Summary

In this chapter, we summarize the thesis, answer the main research questions, initially outlined in Chapter 1. Section 5.1 introduces a general conclusion of this thesis, Section 5.2 gives the answers for research questions. Then in Section 5.3 we talk about the limitations faced in this thesis. And finally, Section 5.4 provides an overview of future work.

5.1 Conclusion

This thesis is dedicated to bringing a solution to the potential system of recreating emulated RTOS based ESP32 together with Linux based Raspberry PI on top of OS-level virtualization, where valid IoT development evaluations can be performed among students and professionals. The proposed artifact handles emulating both Raspberry Pi and ESP32 board that offers some advantages. Firstly it emulates both Raspberry Pi and ESP32 devices and connects these 2, secondly, it mimics the experience with working real physical devices with the help of emu CLI, support scaling up and lastly, it comes with different examples that show use case for common IoT solutions with coupled usage of Raspberry Pi and ESP32. As a summary of what functionalities can be used for ESP32 and Raspberry Pi, Ethernet is enabled for both, but Wi-Fi and Bluetooth are not working, from the peripherals, perspective GPIO subsystem can be used for Raspberry Pi but not for ESP32. Other than that, ESP32 supports *flash* and *monitor* operations and both can run any application that doesn't require hardware functionalities. Short summary table given in Table 9

Table 9. *Functionalities summary for ESP32 and Raspberry Pi*

Functionality	**Raspberry Pi**	**ESP32**
WIFI	✖	✖
Ethernet	✓	✓
Bluetooth	✖	✖
Peripherals	GPIO	UART (limitied)
Applications	Any application requiring no hardware	Any application requiring no hardware
Additional		monitor and flash

5.2 Answering research questions

In Chapter 1, we have stated the main research question as follows: *How to provide an embedded code development environment with an emulated IoT lab that can be used as a learning environment before starting with the real physical devices?*. For the purpose of reaching the answer, we have divided it into 2 sub-questions. The summary of answers for each question is defined below.

RQ-1: How to set up the IoT Environment Lab? As the main purpose of this master thesis is to make the IoT Environment Lab accessible to everyone, we decided to go with the virtualization method and built the Lab on top of it that made way to easily shuttle Lab between different environments. To set up the IoT Lab Environment selection criteria of different software based on the simplicity of usage, popularity, and compatibility of the system with other parts.

RQ-2: How to get users to be familiar with IoT Environment Lab? To answer this sub-question, examples folder have been developed together with emu CLI that addresses some issues with QEMU emulation. Examples folder shows both use case of emu CLI and holds C code with ESP-IDF library usage to describe the solution and Ansible Playbooks file to configure Raspberry Pi. CLI describes the most useful functions to make working with IoT Environment Lab easily. Having GPIO support for Raspberry Pi provides user experience to work with the Raspberry Pi GPIO subsystem.

5.3 Limitations

Several limitations in research are faced in different parts of this thesis. First, GPIO functionality in Espressif fork of QEMU is not implemented and the Wi-Fi MAC hardware register interface is not documented by Espressif so that has ruled out of the possibility of writing an emulator for it. Therefore, an Ethernet connection is chosen to connect ESP32 and Raspberry Pi instead of the Wi-Fi. Secondly, the GPIO subsystem is not implemented for ESP32 but Raspberry Pi.

5.4 Future work

The research should be conducted to further develop QEMU with Espressif patches to support the GPIO subsystem of ESP32. It would lead to having some peripheral emulation which is very needed functionality. Other than that, the starting address of the *RPI_GPIO* device's memory region would need to be updated so that it can match to the correct address for Raspberry 3. But this version in the thesis should be enough to run most of the software except the ones very specific to the newer ARM processor core families used in Raspberry Pi 2 and 3. The additional change would be having a bridge on the interface *eth0* between Host and Guest QEMU instead of port forwarding that would add utilizing more ports than predefined ones.

Bibliography

[1] In Lee and Kyoochun Lee. “The Internet of Things (IoT): Applications, investments, and challenges for enterprises”. In: *Business Horizons* 58.4 (2015), pp. 431–440. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2015.03.008. URL: http://www.sciencedirect.com/science/article/pii/S0007681315000373.

[2] IoT Analytics. *IoT 2019 in Review: The 10 Most Relevant IoT Developments of the Year*. https://tools.ietf.org/html/rfc2324. [Accessed: 12-01-2020].

[3] Rob van Kranenburg and Alex Bassi. “IoT Challenges”. In: *Communications in Mobile Computing* 1.1 (Nov. 2012). DOI: 10.1186/2192-1121-1-9. URL: https://doi.org/10.1186/2192-1121-1-9.

[4] Brian Ramprasad et al. “EMU-IoT-A Virtual Internet of Things Lab”. In: *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE. 2019, pp. 73–83.

[5] Bart Rem et al. “Automatic Handel-C generation from MATLAB® and Simulink® for motion control with an FPGA”. In: vol. 63. Sept. 2005, pp. 43–69.

[6] Daniel Bartholomew. “QEMU: a multihost, multitarget emulator”. In: *Linux Journal* 2006 (2006), p. 3.

[7] Bill Gatliff. “Embedding with GNU: GNU debugger”. In: *Embedded Systems Programming* 12 (1999), pp. 80–95.

[8] Charles Anderson. “Docker [software engineering]”. In: *IEEE Software* 32.3 (2015), pp. 102–c3.

[9] Alexander Maier, Andrew Sharp, and Yuriy Vagapov. “Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things”. In: *2017 Internet Technologies and Applications (ITA)*. IEEE. 2017, pp. 143–148.

[10] Wikipedia contributors. *ESP32 — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-January-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=ESP32&oldid=963206594.

[11] Matt Richardson and Shawn Wallace. *Getting started with raspberry PI*. " O’Reilly Media, Inc.", 2012.

[12] Red Hat Ansible. *How Ansible Works*. URL: https://www.ansible.com/overview/how-ansible-works.

[13] Tural Ismayilov. *QEMU system emulation for Raspbian Stretch Lite and ESP32*. https://github.com/ismajilv/docker-emu/. 2020.

[14] Margaret Rouse. *What is command line interface (CLI)? - Definition from WhatIs.com*. Apr. 2018. URL: https://searchwindowsserver.techtarget.com/definition/command-line-interface-CLI.

[15] Richard Petersen. *Ubuntu 18.04 LTS Desktop: Applications and Administration*. surfing turtle press, 2018.

[16] Jack Wallen. *Ubuntu Server: A cheat sheet*. Mar. 2017. URL: https://www.techrepublic.com/article/ubuntu-server-the-smart-persons-guide/.

[17] *Xtensa on QEMU*. URL: http://wiki.linux-xtensa.org/index.php/Xtensa_on_QEMU.

[18] Ebiroll. *Ebiroll/qemu$_e$sp*32. URL: https://github.com/Ebiroll/qemu_esp32.

[19] Evan Robert Platt. "Virtual peripheral interfaces in emulated embedded computer systems". In: (2016). URL: https://repositories.lib.utexas.edu/handle/2152/46169.

[20] dhruvvyas90. *dhruvvyas90/qemu-rpi-kernel*. URL: https://github.com/dhruvvyas90/qemu-rpi-kernel/wiki/Emulating-Jessie-image-with-4.x.xx-kernel.

[21] dhruvvyas90. *SSH (Secure Shell)*. URL: https://www.raspberrypi.org/documentation/remote-access/ssh/README.md.

[22] BA Bagula and Zenville Erasmus. "Iot emulation with cooja". In: *ICTP-IoT workshop*. 2015.

Appendices

Appendix 1 - Example Ansible Playbook

```
---

- name: Setup raspberry pi
  hosts: pi
  tasks:
    - name: Wait for connection
      wait_for_connection:
        delay: 10
        timeout: 150

    - name: Install mosquitto and lsof
      apt:
        name:
          - mosquitto
          - lsof
        update_cache: yes
        autoclean: yes
        state: present
      become: True

    - name: Kill process in port 8080
      shell: "pkill -9 $(lsof -t -i:8000)"
      ignore_errors: yes

    - name: Listen on port 8000
      become: yes
      blockinfile:
        path: /etc/mosquitto/mosquitto.conf
        block: |
          listener 8000
    - name: Restart mosquitto service
```

```
      become: yes
      shell: "/etc/init.d/mosquitto restart"
```

Appendix 2 - ESP32 image builder

```
#!/usr/bin/env bash
set -e
arg_project=$1
arg_flashimg=$2

if [ -z "$2" ]; then
     echo "Usage: make-flash-img.sh flash_img_file.bin"
     echo "if no param given we used default name flash_img.bin"
     arg_flashimg="flash_image.bin"
fi
rm -vf flash_image.bin
dd if=/dev/zero bs=1024 count=4096 of=${arg_flashimg}
dd if=build/bootloader/bootloader.bin bs=1 seek=$((0x1000))
    of=${arg_flashimg} conv=notrunc
dd if=build/partition_table/partition-table.bin bs=1
    seek=$((0x8000)) of=${arg_flashimg} conv=notrunc
dd if=build/${arg_project} bs=1 seek=$((0x10000))
    of=${arg_flashimg} conv=notrunc
```

Appendix 3 - Emu CLI - Functions

```
Usage: emu [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  flash     Same as idf.py flash
  log       ['esp32', 'raspberry_pi'] log one of the device logs
  monitor   Same as idf.py monitor
  restart   Restart IoT lab environment
  rgpio     See raspberry pi gpio state
  ssh       SSH into raspbian
  start     Start IoT lab environment
  stop      Stop IoT lab environment
```

Appendix 4 - Ansible inventory file

```
[all:vars]
ansible_user=pi
ansible_ssh_pass=raspberry
ansible_ssh_extra_args='-o UserKnownHostsFile=/dev/null
                        -o StrictHostKeyChecking=no'

[pi]
localhost:2222
```

Appendix 5 - Questionnaire form

- What was the challenging part about setting up and running the Lab, what would you improve about it?
- How similar the experience is between working with the physical and emulated devices in terms of user experience, deployment, and main?
- How would an emulated IoT Environment Lab ease your day-to-day activities with the real ESP32 or Raspberry Pi device?
- What you would like to add to the emulated IoT Environment Lab?
- What additional tasks do you think would be beneficial to try on such an emulated environment?
- How would you rate the ease of use of the emulated IoT Environment Lab, do steps taken to run system reasonably understandable, or would like to enhance it? If so, please provide a bit of detail.
- General thoughts on the emulated IoT Environment Lab?