AALBORG UNIVERSITY

STUDENT REPORT

Kuiga Box:
A Test Bed for Large Scale Testing of Virtualized IoT Devices

Project Group:
sw704e16

Supervisor:
Brian Nielsen

AALBORG UNIVERSITY
STUDENT REPORT

Cassiopeia
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East
Phone: 9940 9940
Fax: 9940 9798
`http://www.cs.aau.dk/`

Title:
Kuiga Box:
A Test Bed for Large Scale Testing of Virtualized IoT Devices

Theme:
Internet Technology

Project Period:
September - December 2016

Project Group:
sw704e16

Participants:
Casper Møller Bartholomæussen
Carsten Vestergaard Risager
Jonas Sand Madsen
Kaj Printz Madsen
Rene Mejer Lauritsen
Rune Willum Larsen

Supervisor:
Brian Nielsen

Copies: 9
Number of pages: 68
Date of Completion: 21st December 2016

Abstract:

The Internet of things presents great challenges in the field of wireless communication in terms of situations where numerous of small devices are within the same vicinity of a given base station. This can however be a difficult and impractical scenario to test.
In this project, we investigate how to design and implement a test bed, capable of executing and hosting potentially thousands of virtual devices, each running a provided application, for the purpose of testing communication between devices and a base station. We approach this problem by analyzing the technicalities related to virtualization and scaling as well as constituting requirements that concentrates on the performance aspects of constructing a solution to such a problem.
In the conclusive application, we gather test results that show it is possible to virtualize at least 500 devices using a computation node with the choice of using Docker as a virtualization tool for virtualization a RIOT application. Moreover, with regards to the set of requirements put forward for the project, we have satisfied all of the requirements regarded as essential for the solution, and we have satisfied some of the wanted but not necessitated requirements.

Preface

This paper was written by six 7th semester software students (1st semester of Master of Engineering (Software)) from the Department of Computer Science at Aalborg University (AAU). The development and writing of this paper took place during the fall semester of 2016, starting at the 2nd of September until the 21st of December. The semester theme is: *Internet Technology*.

We want to use this space to thank the people who helped us throughout this semester. First of all a big thanks to Brian Nielsen for giving us good advice and guidance during the project. Also a big thanks to Andrea Cattoni, whom has provided us with excellent feedback and helped us during the creation of requirements. We would also like to express our gratitude towards group 16gr950 from Department of Electronic Systems, who were great at answering our questions.

Reading Guide

The reading guide should help you (The reader) get the most out of this paper, by removing any doubt of how the paper's content, citations and sources should be read and understood.

Bibliography

In this paper we use the Vancouver reference style for the bibliography (found in Chapter 8.3). The format of a citation for a book is: *Author(s), Title of article/section/book title, publisher, editor, Date/Year/Month of publication. ISBN number, URL.*

The format of a citation of a online source is: *Author(s). (Date of publication), Title of article/section/book title, URL, visitation day.* Any of the mentioned information that could not be found is omitted from the bibliography.

Kuiga Box

The software system which has been created during this project will be referenced to as Kuiga Box. Kuiga Box describes the whole software suite, consisting of the components: Kuiga Master, Kuiga Worker, Kuiga Plexer, Image Store, and Kuiga Web. An overview of the system is illustrated on Figure 3.2. The word Kuiga is supposedly Swahili for emulate/imitate, and stems from the project vision of being able to virtualize thousands of IoT devices.

Figures

Figures in this report are centred horizontally. A caption beneath the figure describes the content of the figure, as well as its reference tag. Figure 0 is an example of this.

Figure 0: *This is an example of a figure*

Listings

Listings are used throughout this report to display code examples. Code examples are syntax highlighted and might be simplified to exclude lines that are unnecessary to convey its meaning.

```
object HelloWorld extends Application {
  println("Hello World")
}
```

Listing 1: *This is an example of a listing*

Glossary

At the end of the paper you can find the glossary list. The glossary serves as a description of the acronyms and concepts that we have defined and used throughout the paper. Next to a glossary is/are the page number(s) of where the term has been used.

Contents

1 Problem Analysis **1**
- 1.1 Problem Domain . . . 2
- 1.2 Virtualization & Hypervisors . . . 3
- 1.3 Problem Statement . . . 7

2 Technical Analysis **9**
- 2.1 Operating System . . . 9
- 2.2 Virtualization Tools . . . 11
- 2.3 Virtual Infrastructure Management . . . 14
- 2.4 Requirements . . . 15
- 2.5 Technical Analysis Summary . . . 19

3 Design **20**
- 3.1 Initial Design . . . 20
- 3.2 Overall Architecture . . . 21
- 3.3 Kuiga Master . . . 22
- 3.4 Image Store . . . 22
- 3.5 Kuiga Worker . . . 23
- 3.6 Kuiga Plexer . . . 23
- 3.7 Kuiga Web . . . 25
- 3.8 Communication . . . 26
- 3.9 Life Cycle of a Virtual Instance . . . 27
- 3.10 Test Run . . . 28
- 3.11 Design Summary . . . 29

4 Implementation **30**
- 4.1 Actor Model . . . 30
- 4.2 Inversion of Control . . . 34
- 4.3 Kuiga Plexer . . . 38
- 4.4 Image Store . . . 39
- 4.5 Building Docker Images and Containers . . . 43

5 Tests **47**
- 5.1 Software Verification . . . 47
- 5.2 Load Test . . . 50

6 Discussion **57**
- 6.1 Extensive Actor Model . . . 57

6.2 The Test Run Abstraction 57
6.3 Broad Delimitation 58

7 Conclusion 59

8 Future Work 61
8.1 Kuiga Master REST API 61
8.2 Kuiga Plexer 62
8.3 Failure Handling 63

Bibliography 66

Glossary 67

1 Problem Analysis

In recent years, we have witnessed the rapid development of various new Internet-connected devices in numerous fields. This includes devices such as smart watches, refrigerators and light bulbs. The development is not only seen in everyday appliances but also in transportation systems and the industry; Internet-connected things are cropping up everywhere. This concept is known as the Internet of Things (IoT), which The International Telecommunication Union (ITU) defines as: *"A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies."* [1].

Aside from our own observations, the considerable growth of the IoT market has also been forecasted by Gartner [2], who believes we will reach 20.8 billion Internet-connected devices by 2020.

Testing is a vital part of the development process in almost any industry, and the IoT industry is no different. The devices need to be tested in many domains, which may include security and communication. The communication domain is especially interesting considering the growth of Internet-connected products, some of which may be geographically close to each other. For instance, a hot spot could be a modern city with thousands of people, who each walk around with several devices connected to the Internet. This many devices close to each other could potentially lead to issues with base stations, which cannot serve all of the connections. Thus, it is interesting to test the communication of IoT applications before they are sent out on the market.

Although there exists tools capable of testing IoT applications, it was brought to our attention from a project proposal by Andrea Cattoni, an industry contact working at Keysight, that there is a demand for a tool capable of running IoT applications and devices on a large scale with the purpose of testing the communication between the devices and a common base station.

We accept to investigate the proposal and establish an initial problem:

> *How can we construct a test environment that is capable of running several IoT devices with the purpose of testing the communication between the devices and a base station?*

1.1 Problem Domain

We conduct a meeting with our industry contact in which we coarsely outline the purpose of the project proposal and illustrate it as a rich picture in Figure 1.1. The figure illustrates a physical test bed for testing IoTs devices. We define a test bed as: *A test execution environment consisting of hardware, software and network configurations dedicated to testing an Application Under Test (AUT).*

We observe that the test bed in Figure 1.1 is comprised of a user wanting to run tests of a given embedded application installed on an IoT device with a given Radio Frequency (RF) module that is responsible for its radio communication. The application, which the user wants to test, is in Figure 1.1 referred to as the AUT. We use the picture in Figure 1.1 to illustrate the user's desires:

- The test results are collected from the interaction between the IoT devices and the base station.
- All IoT devices have similar hardware and run the same embedded AUT.
- The amount of IoT devices can range from a single device to thousands of devices.
- The IoT devices may consist of embedded computing platforms, microcontrollers, sensors, an Operating System (OS) and a RF component.
- The test results are read from a single point of entry.

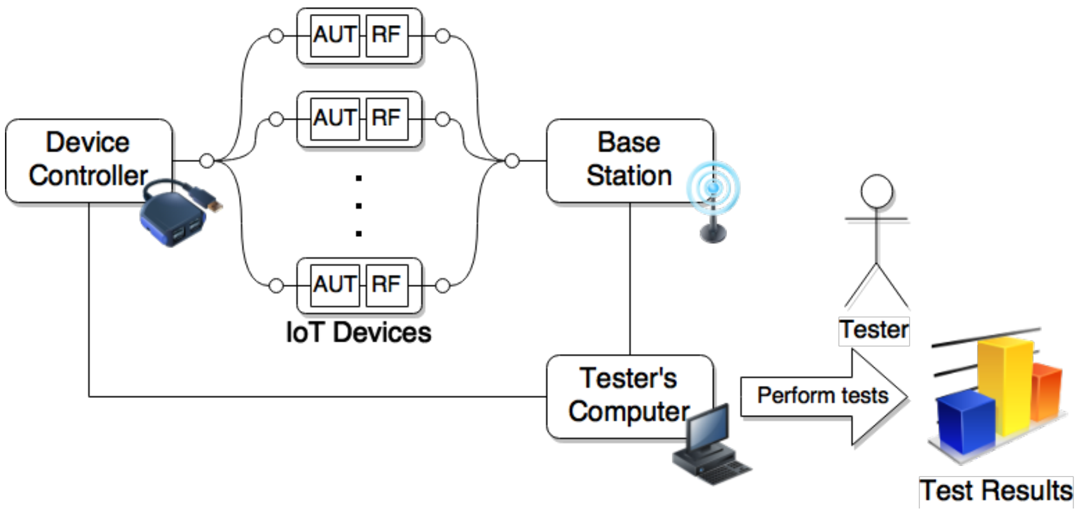

Figure 1.1: *Rich picture illustrating the problem domain of a physical test bed*

We consider that the scenario proposed in Figure 1.1 can be useful for testing the circumstances when multiple IoT devices communicate with the same base station.

However, we recognize several difficulties that render the test bed in Figure 1.1 impractical.

There is a high cost of the initial setup because the user will need to produce and provide thousands of test devices. All devices have to be connected and configured to both the device controller and the base station; moreover, the test bed will at some point need to be physically disassembled again. Lastly, for every component in the test bed, there is another potential point of physical failure which collectively increases the risk of a wholesome system failure. In case of failure, e.g. the test bed becomes stuck in an inconsistent state, we are likely to experience difficulties in recovering because of the sheer amount of devices.

1.1.1 Scope Delimitation

From a meeting with our industry contact, it is clear that a physical test bed consisting of numerous test devices (as described in Section 1.1) is impractical with regards to the cost of producing the devices and the time it takes to setup the test bed or make changes to it. The project proposal suggests solving these issues by creating a virtualizes test bed.

We reason that the project proposal requires all virtualized devices to have access to an RF component as part of the solution. This RF component is under development by another semester group studying Wireless Communication Systems, namely group 16gr950. In more details, we expect that the otherwise physically coupled RF component is substituted by a software component, that can manage the communication between multiple IoT devices and a single physical RF module. We refer to this component as the Software-Defined Radio (SDR).

We define our solution as a virtualization of a test bed and delimit the project scope to mainly focus on the virtualization and handling large amounts of virtualized devices. Onwards, we refer to our solution as Kuiga Box.

1.2 Virtualization & Hypervisors

We investigate the concepts behind hypervisors and virtualization as a means of recognizing what implications these technologies have on creating Kuiga Box. We define virtualization as: *The separation or grouping of resources (storage, network, OS, etc.) into a logical (virtual) layer* and a hypervisor as: *software that controls and runs Virtual Machines (VMs).*

1.2.1 Virtualization Properties

We use three formal properties of virtualization proposed by Popek and Goldberg [3] and explained by Dittner and Rule [4, p. 19]. We use these properties to analyse how the virtualization of the devices impacts Kuiga Box. In addition to the formal properties, we explore generality as a virtualization property.

Fidelity

We define fidelity as: *The degree to which a program running in a virtualized environment is exhibiting behaviour that matches the same program running without virtualization.* The following is an example of fidelity: If a program running in its virtualized environment executes the instructions: $C \rightarrow B \rightarrow A$ but without virtualization executes: $A \rightarrow B \rightarrow C$ then the virtualized program has low fidelity because it does not match the program running in a non-virtualized environment. Another example of low fidelity is: If a device in its non-virtualized environment computes with 10-decimal precision but the virtualized only computes with 2-decimal precision, hence the virtualized device may not output the same results as the non-virtualized device upon the same input.

Efficiency

We define efficiency as: *The degree to which the host system natively executes the virtualized program.* The definition is two-fold. Firstly, the efficiency property is affected by the degree to which the program's binary code needs to be translated to run in the virtualized environment. E.g. a program that requires most of its binary code to be translated in order to be virtualized, has a lower efficiency than a program that only requires a limited amount of translation. And secondly, how well the translated instructions perform comparatively to a non-virtualized program. For instance, let a non-translated program consist of 200 instructions and the translated program consist of 300 instructions; the efficiency is lowered by the overhead of the 100 extra translations. Comparing efficiency against the fidelity property, we observe a potential relationship between the two. This high standard of virtualization negatively affects its efficiency.

Isolation

Our definition of isolation follows the definition of resource control [3, p. 417], which we define as: *the degree of control a virtualized program may gain to a hardware or software resource.* In details, a virtualized program allowed to modify system resources directly is seen as having a comparatively lower level of isolation, than one that is disallowed from doing so. The same applies if a virtualized program may gain access to the host OS's protected files. We observe that the isolation property has a potential trade-off relationship with the efficiency property because isolation likely requires additional computations to operate some layer of interface to a hardware resource required by a virtualized program.

Generality

We regard generality as an informal property meaning that it is hard to quantify in the same manner as for example efficiency. We define generality as: *The amount of different* *hardware components or programs that are capable of being virtualized.* In details, a host capable of virtualizing five different components has a higher generality than if it can only virtualize four. We find that generality is related to efficiency in some capacity since a high generality induce a low efficiency if the generality is achieved through virtualizing specific hardware components that are required by a given program virtualization.

1.2.2 Hypervisor Types

A hypervisor is responsible for distributing and managing its host's hardware resources across its VMs, or guests, and does this by creating virtual hardware resources that are scheduled between its VMs. Consequently, every VM may be given the illusion of having exclusive rights to some resource [5]. We distinguish hypervisors in two generalized types (also see Figure 1.2) as proposed by Masmano, Ripoll, and Crespo [6, p. 264]:

- **Type-1**: Running directly on the hardware of a host
- **Type-2**: Being hosted atop an existing OS on the host

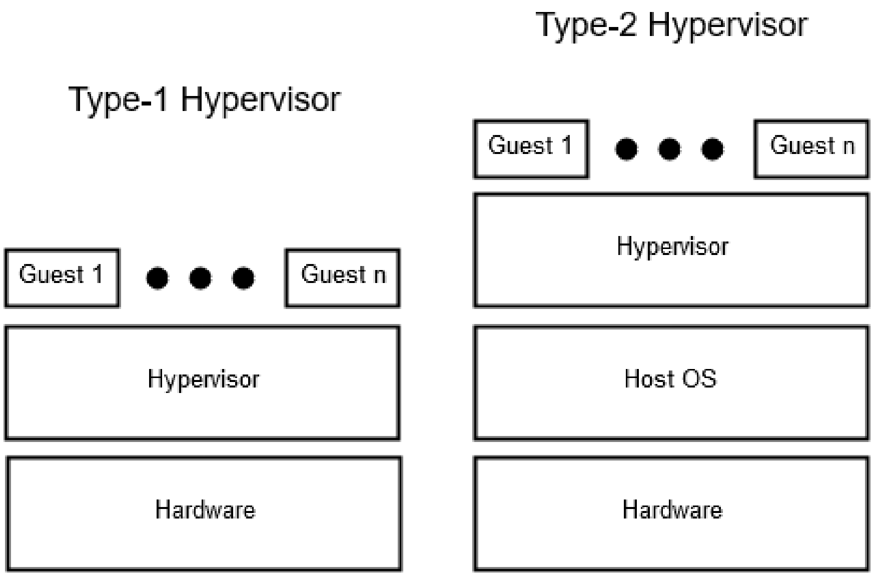

Figure 1.2: *The difference between type-1 and type-2 hypervisors[7]*

If we consider the previously defined virtualization properties, we see that a type-1 hypervisor potentially has better efficiency, because it has direct access to the hardware and it can therefore not be isolated by an underlying host OS. We observe that a type-1 hypervisor essentially has the role of a host OS meaning that only one can be booted per physical computer. This restriction does not necessarily apply to type-2 hypervisors provided that the given hypervisor of this type does not require access to some limited resource that is already exclusively accessed by another type-2 hypervisor.

1.2.3 Virtualization Methods

We research what methods of virtualization there exists and find it convenient to categorize virtualization by three different methods. This categorization is inspired by other categorizations or work performed by various authors [4, p. 22] [8, p. 862] [9, p. 3–6]: We analyze the advantages and disadvantages of these method in relation to the virtualization properties seen in Section 1.2.1.

Full Virtualization

Full virtualization, also called hardware virtualization, uses a hypervisor to virtualize all of the guest OS's required hardware resources, such that the guest OS is executed as a VM on the host without the knowledge of being virtualized.

We argue that full virtualization is highly flexible, since the virtualization of hardware resources allows creating the complete hardware environment required by a VM. This comes with the cost of efficiency because the system calls from a guest needs to be trapped and translated to instructions known by the host OS [8, p. 862]. It is possible to cache the already translated instructions for later reuse to regain some of the efficiency. Another consequence of the hardware being virtualized is that the fidelity of its virtualization is highly dependent on the quality of the virtualized hardware. The method uses VMs that create an isolated environment for each instance such that one VM can not effect the other VMs.

Paravirtualization

In paravirtualization, the guest OS is required to be modified to be hypervisor-aware, which allows coordination between the guest and the hypervisor letting certain privileged instructions be directly executed on the hypervisor.

Paravirtualizaion has potential for better efficiency compared to full virtualization since the guest and host are able to communicate directly on a common ground instead of through virtualized hardware. However, this communication has to be supported by the guest OS or it needs to be modified directly for the hypervisor which may necessitate special drivers [8, p. 862]. Like full virtualization, paravirtualization also uses VMs to create an isolated environment, which results in the same costs and benefits as full virtualization.

Operating System Virtualization

OS virtualization uses a modified version of the host's OS to allow multiple encapsulated versions of the host's OS to be executable at the same time atop the host OS.

It has close to native efficiency since it can almost directly execute its instructions atop the host. The OS virtualization can use containers to create its isolated environments that, unlike VMs, share the same kernel. This sharing of the kernel can lead to a lower resource consumption compared to a VM based virtualization [8, p. 862] but also causes less flexibility as the guest's kernel cannot deviate from the host's kernel. Consequently, all containers are affected if the kernel crashes or gets compromised [8, p. 862]. The same thing is also applicable in other virtualization methods, where the hypervisor can be compromised affecting all of the VMs.

1.2.4 Virtualization at a Large Scale

We use Sections 1.2.1, 1.2.2 and 1.2.3 to explore the concepts of virtualizing a single instance. This research provides us with a framework for reasoning about different approaches to virtualization and hypervisors. However, the solution we intent to create potentially involves many virtual instances involving hundreds or thousands of instances. This fact requires that we consider how to provide a service that functions as intended while increasing the number of virtual instances.

The three methods of virtualization we defined, full-, para-, and OS virtualization, and the implementations of these methods are likely to allow various loads of virtualized instances that it can reliably maintain. For example, given a large load of virtualized instances, performance could become an issue earlier on if we were to choose full virtualization rather than one of the less resource demanding methods.

Software systems that uses virtualization will commonly leverage a hypervisor to provide the virtualization methods defined in section 1.2.3. One should take into account that the different hypervisors and their configurations might have made some compromises in regards to performance. As a part of designing and implementing these software systems, it is therefore not sufficient to merely assess which virtualization method is most appropriate, but the hypervisor must also be judged by the same criterion, as it may put additional constraints on the virtualization.

1.3 Problem Statement

Together with our industry contact, we envision that the system, when given enough hardware resources, should be capable of virtualizing thousands of IoT devices.

We consider that developing this complete system is well outside the scope of a single semester project. This is in agreement with our industry contact, whom envision that this system is to be developed over the course of several years and semester projects.

This project primarily serves as an initial system for this vision and the specific requirements of the concept is for us to address and research. The initial problem (see the beginning of Chapter 1) can be reduced to a final problem statement that defines the goal of the project:

> *How do we design and implement a system, capable of executing and hosting potentially thousands of virtual devices, each running a user provided application, for the purpose of testing communication between devices and a base station?*

We have defined the following sub-questions that elaborate on areas of particular interest in regards to the problem statement.

- *How can we maintain and control multiple virtual instances?*
- *What OSs are relevant to virtualize?*
- *Are there any existing virtualization tools that Kuiga Box can utilize and how?*

These questions composes the reasoning behind the subjects to be investigated in the proceeding chapter.

2 Technical Analysis

We will in this chapter, as a continuation of the proposed problem statement (Section 1.3), identify, analyze and introduce the technical possibilities within the setting of this project. We find it necessary to analyze what OS to virtualize, which virtualization tools are relevant, how we can achieve a distributed system and what is required of that system.

2.1 Operating System

For the purpose of deciding on which OSs are suitable to target for virtualization, we have to recognize the difference between commonly known desktop OSs such as Microsoft Windows and OSs made for IoT devices. Firstly, we observe that due to the hardware constraints of IoT devices, OSs made for IoT devices generally have a smaller footprint in regard to RAM, ROM and CPU usage. Secondly, we observe that while desktop OSs can adapt to various combinations of hardware, OSs for IoT devices usually have to be compiled specifically for the targeted hardware. Finally, it is preferable if the OS supports writing applications that are very effective at conserving battery power. This is a requirement since IoT applications often rely on a battery as their energy source. These applications often have to manage their battery while still being required to perform battery intensive tasks such as transmitting data using wireless [10].

Considering these observations along with the consideration that there exists several OSs for IoT devices [11], we realize that there is a delicacy associated with choosing OSs for IoT devices and there is a need for investigation. Furthermore, we find it appropriate to initially only support the virtualization of a single OS out of the candidates, as it allows us to engage in other challenges at greater depths and spend less time on investigating how we can virtualize multiple OSs. The design of our system should allow for multiple OSs but it is not initially supported.

2.1.1 Operating Systems for IoT Devices

We gather a selection of candidate OSs based on our impression of their reputation, support for cross-platform compilation, community activity and documentation.

We observe from FIT [12] and Brown [11] that *FreeRTOS*, *RIOT* and *Contiki* satisfy these requirements. To determine a suitable OS, we use these two sources along with each OS's respective website as well as our own impressions from trying out the OS. We investigate what the OS is used for, how to build a application for the OS and if the OS can run natively on Linux. An overview of the comparison is presented in Table 2.1.

From FreeRTOS's website [13] we discover that FreeRTOS emphasizes on being a real-time OS and is therefore naturally used for real-time IoT applications. FreeRTOS requires a different port of the OS for each platform/microcontroller e.g. ATSAMD20 ARM Cortex-M0+ and MSP432 ARM Cortex-M4F. Applications written for FreeRTOS can be written in C. It is not immediately clear if there exists any requirements as to which desktop OS is required to develop an application, but based on their example projects they seems to offer Linux and macOS support as well as a Windows port, which is solely dedicated to introduce FreeRTOS.

From observing RIOT's website [14] and their paper [15], we discover that is a real-time OS, with focus on being able to run the same code across different platforms/microcontrollers. RIOT supports multi-threading, drivers for wireless communication and it only needs around 1.5 kB RAM. RIOT has compared to the other OSs an exhaustive documentation, and a very active community. Applications written for RIOT can be written in C and C++. RIOT can be build using Linux or macOS and with some dedication also on Windows. We found it easy to work with RIOT OS as the RIOT toolchain allows us to run the application natively on Linux without making changes to the code.

From Contiki's website [16] we find that this OS is designed to deliver good wireless capabilities with a low RAM footprint. Applications written for Contiki can be written in C. Contiki can also be used on Linux, macOS, and with some trouble on Windows. Contiki allow us to run applications natively on Linux.

	RIOT OS	FreeRTOS	Contiki
Cross-platform compilation	Partially	Partially	Partially
OS primarily designed for	Real-time systems	Real-time systems	Internet communication systems
Language for applications	C, C++	C	C
Runs natively on Linux	Yes	No	Yes

Table 2.1: *Comparison overview between RIOT OS, FreeRTOS, and Contiki*

We find that every candidate is suitable for our use case, however, we decide to settle on using RIOT OS as the initially supported OS, because it is, from our experience, the easiest to get started with, provides the most accessible documentation while also allowing us to run applications on Linux.

2.2 Virtualization Tools

We use this section to identify and compare existing software virtualization tools suitable for our use-case.

2.2.1 QEMU

Quick Emulator (QEMU) is a type-2 hypervisor that utilizes a full virtualization technique such that an unmodified OS can be executed as a guest VM on top of a host system. Since QEMU utilizes full virtualization it can virtualize a different hardware architecture than the host's hardware architecture. QEMU is supported on Linux, Window and macOS . It supports emulation of a wide range of different hardware platforms such as x86, PowerPC and various ARM architectures [17, p. 41].

QEMU uses, during run-time, a dynamic translator that translates the CPU instructions from the guest VM into instructions understood by the host's CPU. The converted binaries are cached such that the translation only has to happen once [17, p. 41].

QEMU is able to make use of a Linux kernel module, Kernel Virtual Machine (KVM), which speeds up the virtualization process.

2.2.2 Xen

The Xen hypervisor is a type-1 hypervisor, which uses both the para- and full virtualization techniques. As a type-1 hypervisor, the Xen hypervisor runs directly on the hardware (bare metal) of the host as represented in Figure 2.1. The hypervisor is responsible for managing CPU, memory and interrupts. [18]

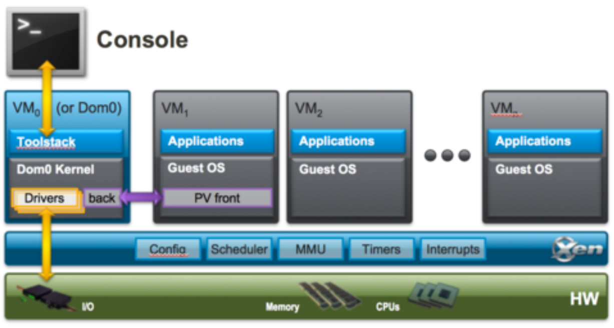

Figure 2.1: *The architecture of the Xen hypervisor [18].*

It uses a specially designed VM called Domain 0 to manage the VMs and exposes a control interface (Toolstack) for which Domain 0 can be controlled. The Xen project offers different Toolstacks, each exposing a different API and their associated tools. Domain 0 also contains the system's hardware drivers, giving it direct access to the hardware. It is granted special privileges like controlling the access to the system's I/O functions and manage the other VMs. [18]

The Xen hypervisor is capable of running both para- and full virtualization VMs simultaneously. For running a VM in full virtualization, Xen requires Intel VT or AMD-V virtualization extensions from the host [18]. It uses the QEMU hypervisor to virtualize the hardware, so OSs with a different kernel than the host can be used. This virtualization overhead is costly, consequently making it, in most cases, slower than a paravirtualised guest. To increase the performance of the full virtualization, the hypervisor can use special paravirtualization drivers, which can help bypass certain parts of the virtualization in the full virtualization environment. [18]

2.2.3 Docker

Docker uses the OS-level virtualization, allowing multiple containers to run in an isolated userspace on the same host. It can run natively on any x64 host running a modern version of the Linux kernel, Turnbull [19, p. 15] recommends version 3.10 and later.

It is built on a client-server architecture, with each client communicating with the Docker Engine. The clients can control the Docker Engine through its terminal or REST API. The architecture of the host running Docker can be seen in Figure 2.2, where the containers are illustrated as a combination of an application with its corresponding binaries and libraries on top of the Docker Engine.

To create containers, Docker uses different features from the Linux kernel like namespaces, cgroups (control groups) and the union file system. By using the namespace feature, each aspect of a container is divided into a namespace for the given container, e.g. managing network interfaces (net) and process isolation (pid). The cgroups can limit a container to only use a specific set of resources. With the union file system, files and directories of different filesystems can be transparently overlaid, with the option of mixing rights like read-only or read-write. [20]

A container is lightweight and can be started nearly instantly, because containers share the kernel of the host [21], compared to starting a fully independent machine, which is the case when doing full virtualization. Docker's drawbacks lie in the flexibility and security of the virtualization. The containers are often limited to a few OSs, that must be similar to the host OS [19, p. 6]. When all containers share the same kernel, it raises security concerns of one container getting compromised and damaging the kernel, which could affect all the other containers. Turnbull states a counter argument to this, that the lightweight container has a smaller attack surface than the combination of a VMs OSs and the hypervisor layer of a VM [19, p. 6].

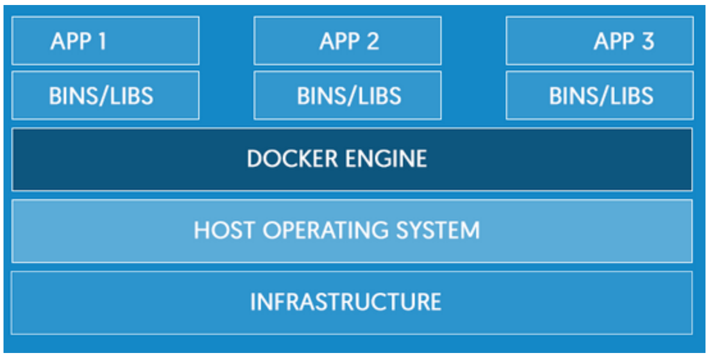

Figure 2.2: *The architecture of a Docker host [21]*

2.2.4 Comparison

Comparing Docker, QEMU and Xen, we find that Docker and QEMU are quite easy to install and setup, whereas Xen takes more time. Xen and QEMU make it possible to use the same toolkit to manage the virtual instances namely libvirt [22], which allows switching between these two virtualization tools easier, whereas Docker uses its own unique API.

We want to utilize Xen's paravirtualization features, but we are not able to find any modified RIOT OS for it, and we consider it out of the scope for this project to create a custom-made image ourselves. RIOT can be built for x64/x86 and ARM architectures, which fortunately can be virtualized using Xen's full virtualization.

Both Xen and QEMU support hardware emulation, which makes them more flexible compared to Docker, as mentioned in Section 1.2.3, which is limited by the host's OS.

We are able to get virtual instances running using Xen hosted on an x86 architecture but we experience problems when we try to virtualize ARM CPU architectures using Xen running on the x86 architecture. These issues might be due to our shortcomings, nevertheless we cannot allocate more time to examine Xen.

We examine QEMU and experience performance issues when executing standard Bash commands. QEMU, on a host with only a single virtual instance, took several seconds to fill the terminal buffer in response to entering various program commands such as *ls* running a Linux distribution[1] for ARM. QEMU can make use of the KVM hypervisor built into the Linux kernel. QEMU uses emulation under normal circumstances, but KVM makes QEMU able to use hardware assisted virtualization which speeds up execution.

We find that Docker is fast at creating multiple containers and in its execution of the code inside the containers, while also using very few resources for each container. We consider the positives of using Docker, namely fast container creation and good performance, as

[1]The QEMU image: `debian_wheezy_armel_standard.qcow2`
from: `https://people.debian.org/~aurel32/qemu/armel/`

more important than Xen's and QEMU's flexibility. We will, based on this reasoning, focus on using Docker as the primary virtualization tool.

We will be virtualizing RIOT applications running in Docker containers hosted on a x86 server. Some difficulties arise as a consequence of choosing Docker namely if we later decide to support an IoT OSs that executes on an architecture that is different from the host architecture.

2.3 Virtual Infrastructure Management

The problem statement in Section 1.3 expresses a need for scalability, in the ability of being able to manage and virtualize thousands of IoT devices. This can partly be accomplished by horizontal scaling by distributing the virtualization workload across multiple computers. For this purpose, virtual infrastructure management is an interesting field. A virtual infrastructure manager is a tool which main responsibility is to operate two resources, namely the computing and storage resources across many nodes [23]. We investigate such a tool, to find out whether it can be used in our context.

2.3.1 OpenStack

We investigate OpenStack[2] as a possible solution and source of inspiration, as it is open-source, used by recognized brands, supported by a large community and has a comprehensive documentation. At the time of writing, OpenStack offers six core services as well as thirteen optional [24], from which we examine a handful of services that we consider suited for our project:

Heat: An orchestration service for launching multiple applications using templates.

Nova: Manages the computation nodes of an OpenStack environment hereby scheduling and spawning machines.

Glance: The OpenStack image service.

Magnum: An API service that makes the use of container software, such as Docker or Kubernetes, work well in combination with other OpenStack services.

Experiences with OpenStack Services

From our initial interest in OpenStack and the relevant services it provides, we investigate the technology and try to get some hands-on experience with it. OpenStack and the previously mentioned services have a lot to offer, more than what Kuiga Box needs, so there

[2]OpenStack website: `http://www.openstack.org/`

should be some clear advantages to using them. From our investigation we encounter early difficulties trying to install and configure any of the services. Moreover, it is also troublesome to integrate all of the services together and the official installation guides recommend installing additional services such as *Barbican*, which does not bring any necessary functionality to Kuiga Box. We have allocated a significant amount of hours for installing and examining the services of interest, as a means of discovering whether they satisfy our needs.

From a technical discussion with our industry contact, we gather that they also had similar experiences with OpenStack and suspect that it will not fit the final system, because it seemingly has a too much overhead in performance. We are though not completely certain if this is the case. If the performance issues with OpenStack can be resolved, then it might be a viable solution, however we assert that spending more time on investigating the performance issues of OpenStack entails a risk that is too great in case OpenStack is still not suitable.

OpenStack offers much of the needed functionality, but also more than we actually need. However, combined with the difficulties confronted in our investigation, our industry contact's recommendations and the associated risks, we decide to not use any of the listed OpenStack services.

2.4 Requirements

We obtain the system requirements for Kuiga Box throughout multiple stages of analysis. It is noteworthy, that the organizational requirements were found together with our industry contact during the initial phases of development.

We analyze and process the organizational requirements gathered from our industry contact with the purpose of generating system requirements that are feasible for developing Kuiga Box, within the available time. The organizational requirements are not only suited for the development of Kuiga Box but for the complete problem domain and must ultimately work as a part of a greater system.

2.4.1 Non-functional Requirements

The non-functional requirements are expressed as a characterization of Kuiga Box as a system, which entails that we may describe Kuiga Box in terms of its non-functional requirements. We use the types of non-functional requirements proposed by Sommerville [25, p. 88], for which we consider product, organizational, and external requirements to Kuiga Box. Within these classifications and from analyzing the requirements put forward by our industry contact, we elicit high-level organizational requirements in terms of the operational requirements that apply to the whole project proposal:

- A user can give the system an application for which the system will generate a given amount of virtualized instances of this application.
- The applications to be tested are small; applications that run on platforms with the size of a wrist watch battery are common.
- The communication with a base station is the primary test area. What are the effects of letting thousands of IoT devices communicate with a common base station?
- The level of fidelity is expected to be low and real-time critical systems are not targeted in the project.
- The system can virtualize up to 50 000 of the same IoT device with its application.
- The user can specify which hardware platform should be virtualized
- A test scenario consisting of 50 000 IoT devices should only in the worst case have 100 active devices at the same time.

We use these requirements along with our collected knowledge from the problem analysis (Chapter 1) and technical analysis (Chapter 2) to stipulate which features Kuiga Box can feasibly cover and how non-functional should be prioritized. Firstly, we eliminate some general types of product requirements [25, p. 88]: Security and usability requirements, because there are no highlighting of any security issues and we delimit Kuiga Box from user interaction as seen in Section 1.1.1. The remaining product requirements are efficiency and dependability.

Efficiency

We argue that, in terms of space requirements there are no specific requirements of importance to Kuiga Box, since the organizational requirements already state that the space taken up by the AUTs should be insignificant.

We recognize that there is an emphasis on the performance requirements, as the organizational requirements cover the need of virtualize thousands of devices. In relation to the investigation on virtualization and hypervisors in Section 1.2, we find that the relationship between a virtualization's fidelity, performance and isolation potentially has a significant influence in the requirement of performance. We argue that it is more important to concentrate Kuiga Box's performance characteristics than it is on fidelity and isolation, because we suspect that the collective performance requirement of thousands of IoT devices will cause issues. In a long perspective, if Kuiga Box is able to handle at least 1 000 devices with low fidelity it will be relevant to concentrate on other requirements like fidelity.

As a means of achieving a suitable performance, the requirement of scalability is appropriate to consider. We expect that in extreme workloads reaching 50 000 devices, achieving a satisfactory performance through vertically scaling is by all means impractical and we therefore consider the means of horizontally scaling. We undergo the consideration of scaling virtualized instances in Section 1.2.4. We determine no specific tangible goal for scalability, but we require Kuiga Box to be designed to support some horizontal scalability.

Dependability

Considering the organizational requirements of Kuiga Box, we find no further requirements related to dependability, as the system is not critical in operation. However, when considering the technical aspects of the designated solution, we realize that Kuiga Box does have some dependability requirements in regards to resilience.

We define the resilience requirement as *a system's capabilities in detecting and recovering from errors*. Seeing that Kuiga Box is planned to be designed as a distributed system, the resilience requirement becomes relevant, because for each computer node in a system, the risk of experiencing errors increases [26, p. 22–24]. Considering the vision of running thousands of virtual instances, we find that resilience will have an important and beneficial impact, because an increased amount of virtual instances necessitates scaling across multiple nodes where each node is susceptible to crashing. We prioritize efficiency-related requirements above dependability-related requirements, because efficiency-related requirements are more important to the concepts of Kuiga Box. With this in mind, we must consider resilience as part of the design, but we do not put forward any tangible goals.

2.4.2 MoSCoW Analysis

In continuation of the organizational requirements, we proceed to formalize the high-level user requirements, as specific requirements in a *MoSCoW* analysis [27]. The requirements of the *MoSCoW* analysis are illustrated in Table 2.2.

Must have	Simple API for managing the virtual instances
	Create, start and terminate virtual instances
	Manage at least 100 virtual instances
	Support one virtualization method
	Support one OS for virtualization
	Capture the network traffic of the virtual instances
Should have	Virtualization of various OSs
	Support horizontal scaling
	Connect Kuiga Box with the SDR
Could have	Support various virtualization methods
	Hardware platform virtualization
	Support +10 000 virtual instances
Won't have	Extensive GUI for managing virtual instances

Table 2.2: *The table representing our MoSCoW analysis*

Must Have

We consider the tasks in the *Must have* category to be mandatory for Kuiga Box and can collectively be seen as the minimum subset of features that makes Kuiga Box functional.

We require Kuiga Box to provide an API for which a developer is able to manage the virtual instances. The management through the API involves the ability to create new virtual instances from the application provided by the user. The user must be able to start and terminate these virtual instances through the API.

We intent on supporting the virtualization of RIOT OS and applications, by using OS virtualization with Docker. From Section 2.2 and 2.1, we discover that the combination of RIOT and Docker show a potential in terms of performance and it is easy to work with, in comparison to the other virtualization methods and OSs.

The system must be able to manage at least 100 virtual instances concurrently, without a significant drop in performance for the individual virtual instances. We find this reasonable, because the investigation of Docker show that this amount of instances is achievable.

The support for forwarding network traffic relates to the organizational requirement, that Kuiga Box must at some point be connected to an SDR. We support some of this requirement by connecting the virtual instances to a common network interface, that we also use for debugging by logging the network traffic produced by the virtual instances.

Should Have

We consider the features in the *Should have* category to be important for Kuiga Box but not essential.

From the organizational requirements, we acquire that the system should be general in terms of its support of OSs. This is important, because various users will likely have an application for various OSs, however based on a discussion with our industry contact, we gather that solely supporting a single operating system is sufficient.

The support for horizontal scaling is not a direct organizational requirement but an aftereffect of the performance requirements. We expect horizontal scaling to be a hard task to complete in a satisfying quality, hence we wish to design Kuiga Box with some effort on allowing horizontal scaling, however we do not expect a suitable implementation to be constructed.

As covered in Section 1.1.1, there is an expectation of another component, the SDR, to be coupled with Kuiga Box at some point. We consider this a goal by itself and although this is essential to the final system, this can be omitted in Kuiga Box, because it is uncertain whether the SDR will be ready for integration at Kuiga Box's end of development. Instead of connecting the actual required SDR, we intent on at least creating some of the intermediary components that are required to transport the network traffic between the virtual instances and a potential SDR.

Could Have

We regard the features in the category *Could have* to be implemented if time is available, but not necessary for the system we are currently developing.

The arguments for the prioritization of *virtualization of various OSs* also applies to the support of *various virtualization methods* and *hardware platform virtualization*. We find that these features are likely important to a final system, as the various virtualization methods and hypervisor types have different properties and features (Section 1.2.2 and 2.2) which may or may not be expected by the user. We expect that these will be needed features in the future, but as they are not necessary now, we consider these as low priority.

Supporting 10 000 virtualized IoT devices can be seen as a milestone. From our organizational requirements, we are aware that virtualizing 10 000 is an ambitious requirement for the system. Lesser is also viable, however the expectation is that supporting 10 000 devices is desired in some test scenarios.

Won't Have

We have excluded the features in the *Won't have* categories from this version of the product, but they could be included in a potential future version.

For this version of the system we will not create an extensive GUI, as the functional features of the back-end are at a higher priority.

2.5 Technical Analysis Summary

We have settled on using Docker as the virtualization tool, and RIOT as our OS. We use a MoSCoW analysis to identify the system requirements. These requirements include a simple GUI, support for one IoT OS (RIOT), a single virtualization tool (Docker), and support for a common network interface that forwards network traffic. We also aim to support at least 100 virtual instances initially.

With these requirements and their priorities in mind we will now design Kuiga Box.

3 Design

We have established the required technical knowledge of the project, and we will proceed by creating a design for Kuiga Box, that will aim to satisfy the requirements in Section 2.4 as well as obeying the technical constraints and necessities from Section 2.5.

3.1 Initial Design

The initial design of the project has been derived together with our industry contact and is informally presented in Figure 3.1. The Kuiga Box in Figure 3.1 illustrates a control unit for monitoring, deploying and handling resources for multiple virtual instances. As a means of accessing the functionality of Kuiga Box, we plan to implement an externally accessible API that decouples the functional logic from for instance a potential GUI.

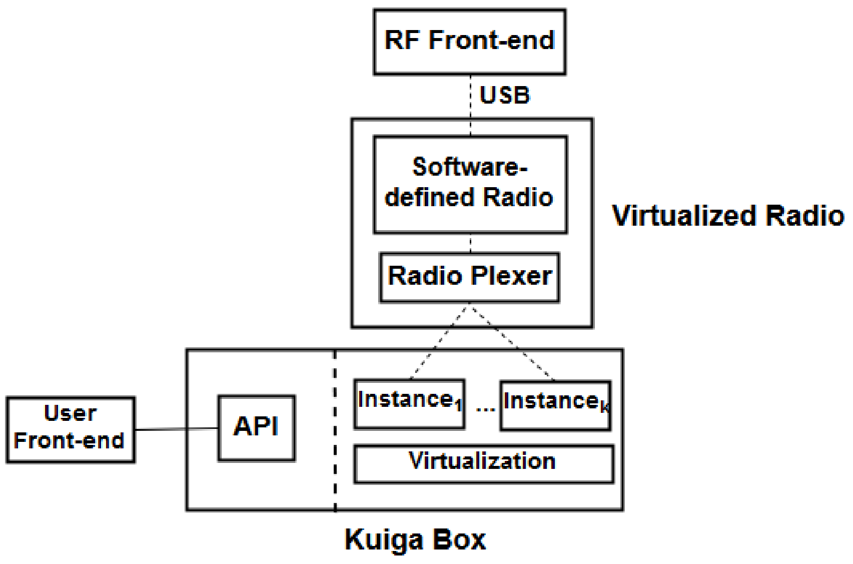

Figure 3.1: *The initial design of the project*

In Figure 3.1, we note that every instance is attached to a *Radio Plexer* component in such a way, that the *Radio Plexer* component acts as an interface for which every instance sends their network packets through. The *Radio Plexer* component works as an intermediate layer between each virtualized instance and the *SDR*, creating the illusion of an instance having exclusive rights to the radio.

The *SDR* is a component that is capable of switching between various wireless communication techniques at run-time and is connected to a physical radio device, the *RF front-end*, that is capable of communicating with a base station. The *SDR* and the *RF* module are maintained by another student group, 16gr950, from Wireless Communication Systems and are thus outside the scope of our project.

3.2 Overall Architecture

From the initial design in Section 3.1, we construct an overall design representing Kuiga Box's architecture.

We confront the issue of performance regarding the virtualization to be carried out by Kuiga Box. We realize, that the virtualization can be computational expensive if there is a large amount of virtual instances. Hence, we design Kuiga Box to allow for horizontal scaling of its virtual instances by distributing the virtual instances across multiple physical computer nodes also known as the worker nodes. By distributing the workload among multiple worker nodes, we potentially achieve a better and more stable performance on each worker, but also allow for a broader support of IoT devices, if the worker nodes are using different hardware platforms and virtualization methods. The distribution also adds some overhead and an extra layer of complexity atop the product, as a load balancer is needed for partitioning the workload between the worker nodes.

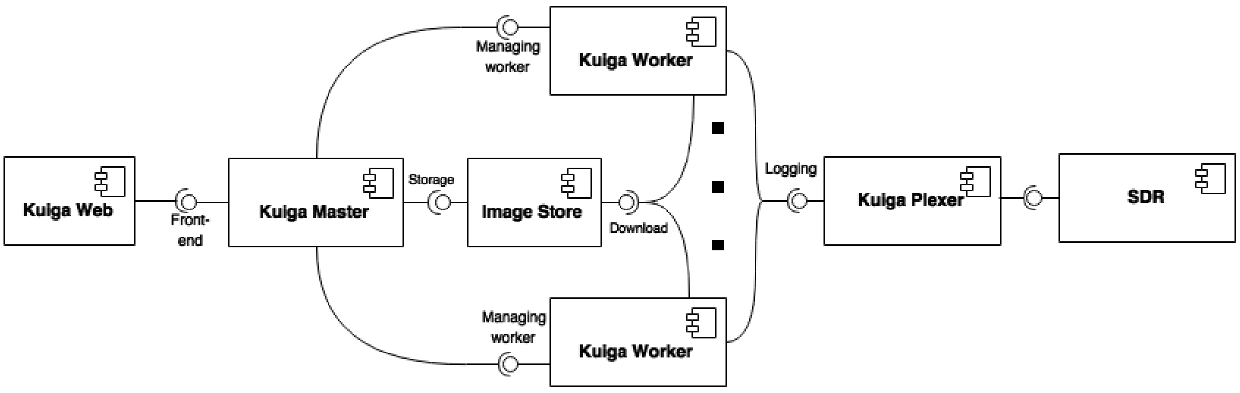

Figure 3.2: *The overall architecture of Kuiga Box as a component diagram*

To address the distribution problem, we use the abstraction of a master-worker relationship, where a single computer is given the role as the master node, that controls one to many worker nodes and also serves as the load balancer of the system. In Figure 3.2 we observe the architecture in broad details, where we outline the *Kuiga Web*, *Kuiga Master*, *Image Store*, *Kuiga Workers*, *Kuiga Plexer* and *SDR* as components that are potentially physically separated. *Kuiga Master* is the component on the master node controlling the *Kuiga Workers*, which are placed on the worker nodes. The small circles on Figure 3.2 represent that a component provides an interface, and the half circles represent the consumption of an interface. We do not provide details on the interface between the *Kuiga Plexer* and *SDR* components, because their connection is not within the scope of the project. We design the system such that all communication with a front-end developer happens through the *front-end* interface provided by *Kuiga Master* as seen in Figure 3.2. The architecture also contains the *Image Store* that has the responsibility of storing and distributing images among the Kuiga Workers. Section 3.4 goes into more details on how the *Image Store* is used in Kuiga Box. *Kuiga Web* is a small web application that gives developers access to the system.

3.3 Kuiga Master

We design Kuiga Master to delegates jobs to the worker nodes, creates images for the worker nodes, as well as other managerial tasks.

One of the Kuiga Master's responsibilities is to create and distribute images. Through the *front-end* interface the Kuiga Master receives an AUT. The Kuiga Master built the AUT with the base image and distributes this newly created image to the worker nodes that satisfies the needed specification. Letting Kuiga Master create the image helps to ensure that each worker have the same image, and the image creation process only have to happen at one node instead of many.

Following the image creation, Kuiga Master conveniently stores the image on the Image Store, such that the Kuiga Worker can retrieve the image from the store.

3.4 Image Store

The Image Store provides storage and retrieval of images and exposes two interfaces: The *storage* interface and the *download* interface. The Kuiga Master uses the *storage* interface to upload images, containing the user-provided AUT, to the Image Store. The Kuiga Workers use the *download* interface to download these images uploaded by the Kuiga Master.

In Kuiga Box, we use two concepts of images, both stored on the Image Store: Base images and runnable images. A base image consists of the data needed to create a virtual instance. where the files in a base image may vary depending on the OS and virtualization method. In the case of Docker and RIOT, a base image contains a *Dockerfile*[1] and a launch script for executing the AUT. The Image Store stores a single base image for each OS and architecture supported by Kuiga Box. When Kuiga Master is asked to create virtual instances, it first locates the necessary base image and retrieves it from the Image Store. It then assembles the base image with the AUT to create a *Docker image*, which in Kuiga Box is generalized

[1] a text file describing how Docker should create an image

as a runnable image. The process of creating images using Docker is further described in Section 4.5.

We illustrate the interaction with the Image Store through the sequence diagram in Figure 3.3. A runnable image is a base image combined with the user's provided AUT, ready for the Kuiga Workers to create virtual instances from. When the Kuiga Master has created a runnable image, it stores the image on the Image Store, since the image can be needed on multible workers, and provides the Kuiga Workers with the location of the image. The Kuiga Workers can then download the image and create virtual instances of it. Using Docker, the Kuiga Workers download a Docker image, loads it into Docker and creates a number of containers from it. As a result of the *Dockerfile* and a launch script, the containers can start executing the AUT from the runnable image.

3.5 Kuiga Worker

From our observations of the overall architecture in Section 3.2, we enable the possibility of distributing the workload of Kuiga Box among multiple worker nodes. Each worker node can, by the use of different environments, enable support for a range of various virtualizations of IoT devices, or horizontal scaling when running the same environment. Worker nodes are responsible for executing jobs received by the Kuiga Master. Furthermore, it is also the worker nodes that configures the network connection between its virtual instances and the middleware.

In Figure 3.4, we show the layers of a single worker node which uses Docker to provide containerization. In the case of Figure 3.4, the worker can virtualize IoT OSs capable of being built together with an application to a Linux x64 host, like RIOT. All the worker nodes are also running a service, Kuiga Worker, which is managing the virtual instances accordingly to the request from Kuiga Master.

We observe in Section 2.2 that the different virtualizations tools might not use the same interface for managing the virtual instances. This problem can be addressed with the use of an adapter between the communication on either the Kuiga Master or the Kuiga Worker. We decide to place it on the worker node, as we consider it the workers' responsibility to interpret messages from the Kuiga Master and not the other way around.

3.6 Kuiga Plexer

In Figure 3.2, we find the Kuiga Plexer component located between the SDR and all Kuiga Workers. Moreover, we observe that all virtual instances are connected to a shared Kuiga Plexer that is located between them and the RF module.

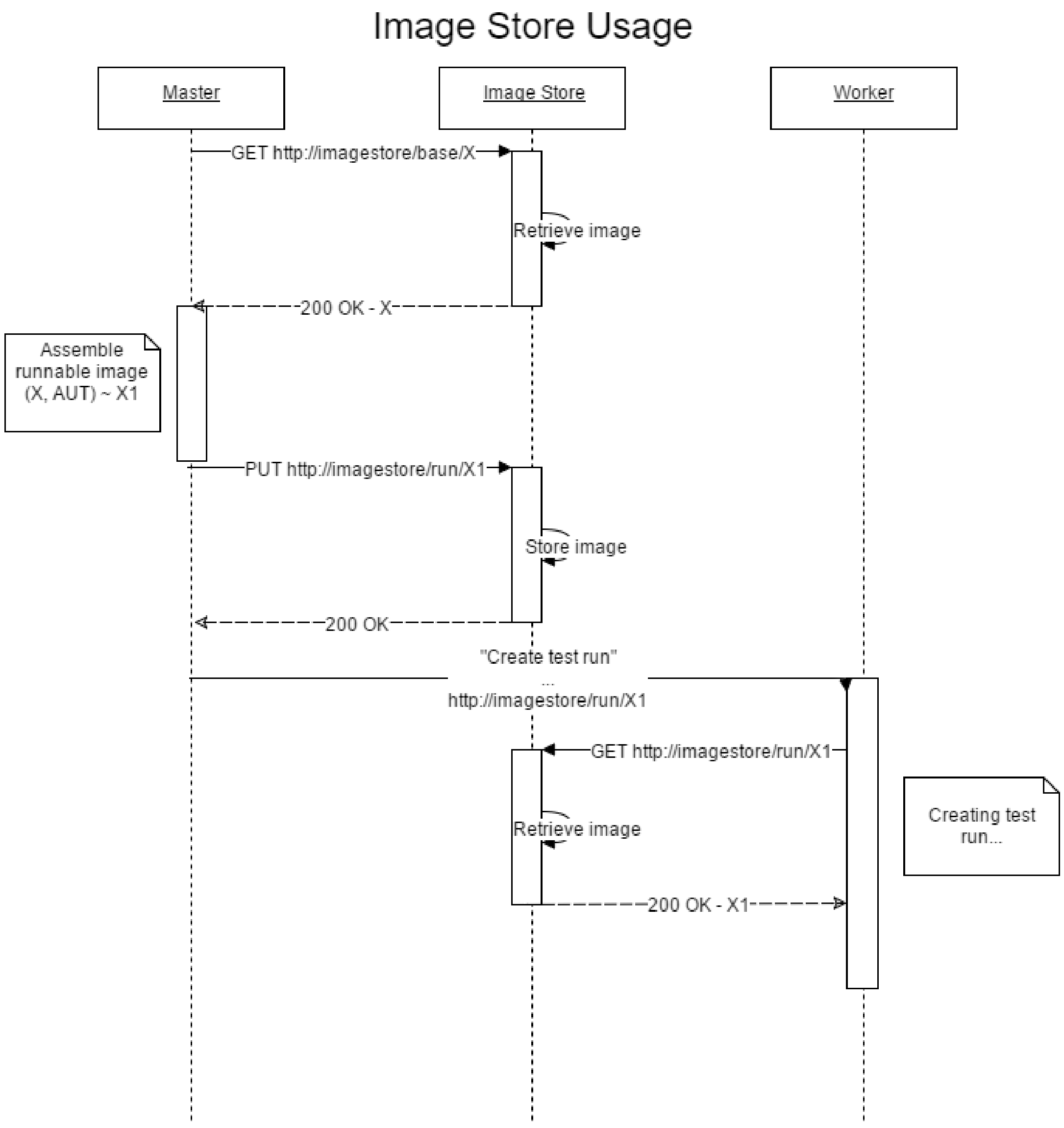
Image Store Usage
Master
Image Store
Worker
GET http://imagestore/base/X
Retrieve image
200 OK - X
Assemble runnable image (X, AUT) ~ X1
PUT http://imagestore/run/X1
Store image
200 OK
"Create test run" ... http://imagestore/run/X1
GET http://imagestore/run/X1
Retrieve image
200 OK - X1
Creating test run...

Figure 3.3: *A sequence diagram showing the Image Store usage*

Based on a technical discussion with the group, 16gr950, who develops the SDR component, we conclude that the Kuiga Plexer must communicate with a program that is recognized as a device on the computer running the SDR. We gathered that the SDR component is composed by multiple User Equipment (UE) components where every UE component represents a single device that supports wireless communication and thus we simplify the job of the Kuiga Plexer to connect the virtual instances to the UEs.

Due to the uncertainty of how the communication between the virtual instances and UEs is going to work, we aim to design Kuiga Plexer as flexible as possible and with the purpose

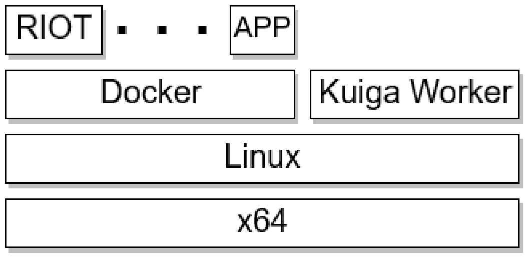

Figure 3.4: *Abstraction layer of a worker node running Docker on a Linux x64 environment*

of simply serving as a mock-up. Consequently, we design Kuiga Plexer with logging capabilities used for debugging and testing, and we plan to do this by capturing the network traffic that are sent by the virtual instances.

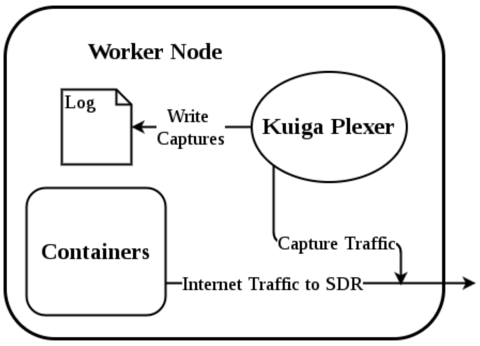

Figure 3.5: *Kuiga Plexer logs container internet traffic by capturing their packets*

In Figure 3.5, we illustrate that the Kuiga Plexer program captures the traffic that each container sends to the SDR system to log this traffic for later retrieval and analysis. Lastly, we aim to implement the logic that captures the network traffic as low level as it is feasible, regarding the Open Systems Interconnection (OSI) [28] layer protocol data units, as it will be easier, later on, to match the needs of the UE's.

3.7 Kuiga Web

Kuiga Web is a simple web application that consumes the front-end interface, allowing a developer or user to interact with Kuiga Box. The requirements in Section 2.4 do not necessitate a Graphical User Interface (GUI) for the system, however we find it very convenient to have a graphical interface as a means of interacting with Kuiga Box through its API. When developing an API, like the front-end interface, an application that consumes the API can lead to the discovery of missing functionality in the API, that was otherwise not thought of in the design.

We design Kuiga Web to provide a simple GUI where the developer can see all of the registered test runs. Before any test runs can appear in the GUI, the developer must create one, which requires uploading an AUT and providing the number of virtual instances that should be created. When a test run has been created, the developer should be able to start it, and while it is running, download log files for that test run. Finally, Kuiga Web should provide the ability to stop any running test run.

Making the GUI as a web application is very practical in terms of location accessibility. The user or developer can connect to Kuiga Box from any computer with Internet access, instead of having to use a physical terminal next to Kuiga Box.

3.8 Communication

We want Kuiga Box to operate in a distributed environment. This requirement necessitates that we identify and confront possible issues related to the communication between our distributed system components.

We can facilitate communication between computer nodes in a multitude of ways i.e. web APIs, web sockets or Remote Procedure Calls (RPC). However, we find interest in the actor model as it both serves as a solution to the communication as well as integrating well with horizontal scaling. We will use the following paragraphs to argue the merits of the actor model as a method of implementing a distributed system.

First, the actor model uses a messaging system, which can send and receive messages. The actor model uses these messages to support communication between computer nodes [29, p. 1]. This message-driven communication design is a characteristic of distributed systems [26, p. 2].

Secondly, we see that asynchronous programming is suitable for communication because every computer node in a distributed system has its own processes and storage. Hence, it is uncertain, when a message sent to a node will return or if it will respond at all. We also wish to have the ability to await a response to a message. The message might not yield an immediate response, but when the receiving node has computed the result, it should send it back to the original sender. In the actor model, we find the notion of a "future" [29, p. 17] which covers this ability. We believe it is an appropriate abstraction because it allows us to use synchronizing messages without blocking the sender node's main thread.

We also note that, from the developer's perspective, a distributed system is represented as a single system and not as a network of arbitrarily many computer nodes. Ideally, the more we can hide the details on how the nodes in a distributed system communicate, the more time we will have to develop and focus on the user's desired features of the system as a whole. The actor model provides help on this through location transparency in such a way that the programming of actors' (the computational unit in the actor model) functionality is independent of the physical location of other actors. The messages are sent by addresses between actors [29, p. 3].

As it is with distributed systems, failures are to be expected. To help prevent failures we can make use of the child-parent relationship between actors. Just like we have parents, an actor should also have a parent. As our parents looked after us, a parent actor should also look after the child or more formally supervise it. In this way, when a child actor encounters an error that it does not know how to handle, the parent makes sure the child gets restarted.[30]

We realize that the potential benefits of using the actor model may be achieved by other means. However, we observe that the actor model offers these benefits as tools in a complete package. In Section 4.1 we investigate Akka[2] as an implementation of the actor model and highlight further details.

3.9 Life Cycle of a Virtual Instance

In Figure 3.6, we present a state diagram for the life cycle of a virtual instance. The life cycle is simple in its design, such that it only includes the states that are necessary to support the management of the virtual instance as mentioned in Section 2.4.2: The ability to create, start and terminate a virtual instance.

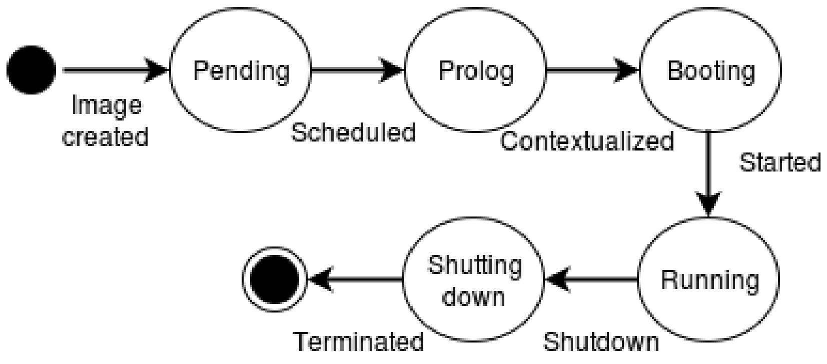

Figure 3.6: *The life cycle of a virtual instance in Kuiga Box*

In Figure 3.6, we see that a virtual instance starts its life cycle when it is created. The image is a runnable image (described in 3.4) and it contains the AUT to be executed. The virtual instance transitions to the *Pending* state after it has been created. The instance stays in this state until Kuiga Master has assigned it to a worker node.

From here, Kuiga Master assigns the virtual instance to a worker and the state transitions to the *Prolog* state, where the contextualization process happens. The contextualization

[2]Website: `http://akka.io/`

process takes care of adding the virtual instance to networks used for communication, giving it an IP address and a name. Furthermore, the worker allocates required resources such as disk space and filesystems to the virtual instance.

Once done, the virtual instance transitions to the *Booting* state, where it awaits being started by the host.

When a Kuiga Worker starts the virtual instance, it transitions to the *Running* state in which the user's AUT is executed until it is explicitly told to shutdown, which is invoked by a user's request to terminate a test. We let the user be responsible for issuing a shutdown command, because we cannot know when the virtual instance should be shutdown or not.

When the virtual instance is told to shutdown, it enters a *Shutting down* state in which its host does any necessary cleaning routines and finally completely terminates the virtual instance ending its lifespan.

3.10 Test Run

We encourage the user of Kuiga Box to think in terms of test runs: How many IoT devices should execute the AUT? The user only needs to provide the number of a supported device to virtualize and the associated AUT, where Kuiga Box will decide how the workload should be executed and distributed among its components. We want to hide the information that Kuiga Box is in fact a distributed system from user, as they only need to see Kuiga Box as a single coherent system enable the user to test their AUT on different devises.

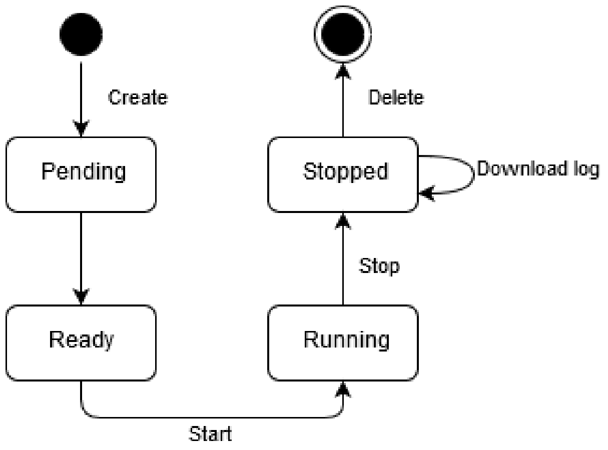

Figure 3.7: *State machine for a test run*

Through the front-end interface provided by Kuiga Master in Figure 3.2, we supply the essential means for the user to manage a test run. We design the interactions between the user of Kuiga Box and a test run as the transitions illustrated in Figure 3.7. The first thing a user would do is to *create* a test run. When a test run has been created the test run goes into the *Pending* state. The user cannot interact with the test run in this state, as the virtual instances of the test run are being created. When all of the requested instances have been created, the test run transitions to the *Ready* state. From this state the user can *start* the test run, which is equivalent to booting all the virtual instances in the test run. As IoT devices are often kept alive by switching between sleep and being active, we cannot know when the user's AUT has finished executing, hence we let the user be responsible for explicitly telling the system when a test run should be *stopped*.

3.11 Design Summary

We observe that the goal of achieving a scalable system, as stated by the requirements in Section 2.4, has influenced our choices during the design process. With this goal in mind, we have designed our components such that they can be implemented to be used for test runs that involve anything from a few virtual instances to several thousands. We have chosen a distributed architecture with a separation between the Kuiga Master distributing work between workers, the Image Store storing the images for Kuiga Master and Kuiga Worker running the virtual instances. We believe that the system has good scalability by using this architecture.

In the following chapter, we implement the design specifications and explain our choice of technology.

4 Implementation

We will use this chapter to present Kuiga Box implementation details of significant parts, due their significance to fulfilling the requirements. We will elaborate on the implementation the actor model, inversion of control, the building of Docker images and containers and the Kuiga Plexer.

4.1 Actor Model

We use the Akka framework[1] to implement our actor model in Kuiga Box. We choose Akka because it is developed for the JVM which corresponds to our choice of mainly programming in Scala. Moreover, we found that Akka has tools for managing pools of actors, it has built-in routing and remoting features and its documentation leaves an impression of being comprehensive and well written. We argue that both the actor pool management and routing is helpful for Kuiga Box in terms of scalability, because these tools can be used to manage multiple nodes and load balancing. The remoting feature is used to connect actors across nodes and is helpful in creating a layer of location transparency when programming with actors.

4.1.1 Actor Structure

We decide to construct Kuiga Box heterogeneous in the sense that the computer nodes in a distributed system of Kuiga Box will not contain the same piece of application. We found this desirable, because we have a clear distinction of the nodes' responsibilities and we expect that a system's hardware in a final solution will be heterogeneous as well to an extent that some combination of Kuiga Box component and hardware will match poorly. Furthermore, we do not predict any requirements in elasticity in such a way that any node can take any responsibility, because we do not expect a serious strain on the user front-end or load balancer (Kuiga Master).

Considering the actor model, this results in the master node and the worker nodes to contain distinguishable actor systems for which we construct a master actor system controlling the actors on the master node and a worker actor system for each worker node controlling the actors of the given node. In the system's perspective this has some immediate

[1] The Akka framework's website: `http://akka.io/`

disadvantages, because the system loses some of its location transparency, however, on the level of developing the actors and the functionality for Kuiga Box, we persist its transparency, because the linking between nodes happens through configuration.

We create five different variants of actors each with their own responsibility. Figure 4.1 illustrates the interactions between the actors.

WorkerActor

Each Kuiga Worker node in the system is represented by a *WorkerActor*. This actor can receive messages from the master node e.g. messages telling the worker to start a test run and respond with messages telling the master whether the action was successful.

LookupActor

The Kuiga Master node needs to keep track of the worker nodes in the system, it accomplishes this by having a *LookupActor* for each *WorkerActor* in the system. The *LookupActor* has the purpose of preserving a reference to its corresponding *WorkerActor* and reestablish the connection between the nodes if it should break. All messages between the Kuiga Master node and a Kuiga Worker node goes through their corresponding *LookupActor*.

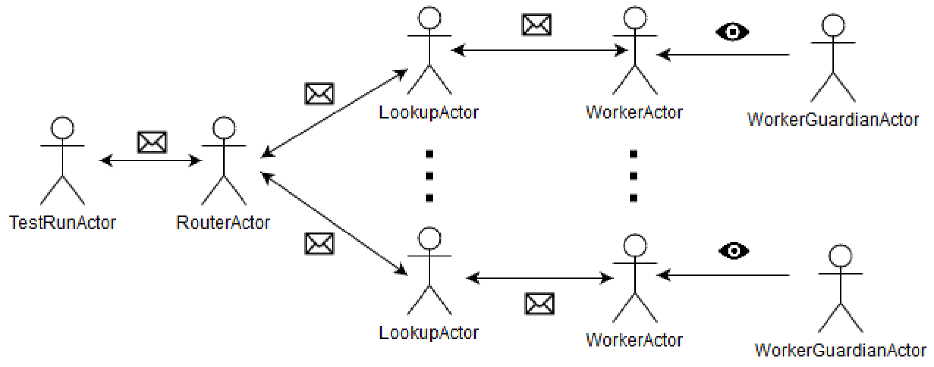

Figure 4.1: *Diagram of the interactions between the actors. The mail icon represents a message relationship (the ability to send messages between the actors), where the eye is a guardian relationship (one actor watches another e.g. WorkerGuardianActor reboots a WorkerActor if it terminates). There can be multiple LookupActors and WorkerActors in the system.*

RouterActor

The *RouterActor* is responsible for distributing work to the *LookupActors* and to redirect message between multiple actors. As we limit Kuiga Box to only support a single worker in this project, the *RouterActor*'s distribution capabilities are not functional to an acceptable degree.

TestRunActor

The *TestRunActor* monitors the status of the instantiated test runs in the system. Listing 2 illustrates an enumeration containing all the possible test run statuses.

```scala
object TestRunStatus extends Enumeration {
  type TestRunStatus = Value
  val pending = Value("pending")
  val ready   = Value("ready")
  val running = Value("running")
  val stopped = Value("stopped")
  val deleted = Value("deleted")
  val dirty   = Value("dirty")
}
```

Listing 2: *The possible statuses of a test run*

In Figure 3.7 (Section 3.10) we describe the states of a test run. These states actually correspond to the statuses seen in Listing 2. If the *TestRunActor* receives a message with the *pending* status, that means the test run is in the *Pending* state. There is however a slight difference between the state and statuses: There is the possibility of failure from any test run state, so we introduce the *dirty* status, which should lead to the removal of the test run.

WorkerGuardianActor

Each *WorkerActor* is monitored by a *WorkerGuardianActor*. The *WorkerGuardianActor* can detect if a *WorkerActor* is terminated and replace it with a new instance of *WorkerActor*. These two actors are located on the same worker node, where a *WorkerActor* upon termination dispatches a termination message to the *WorkerGuardianActor*.

To get an intuition of how messages flow through the system, we illustrate a simple example, where a test run needs to change state, as a sequence diagram in Figure 4.2.

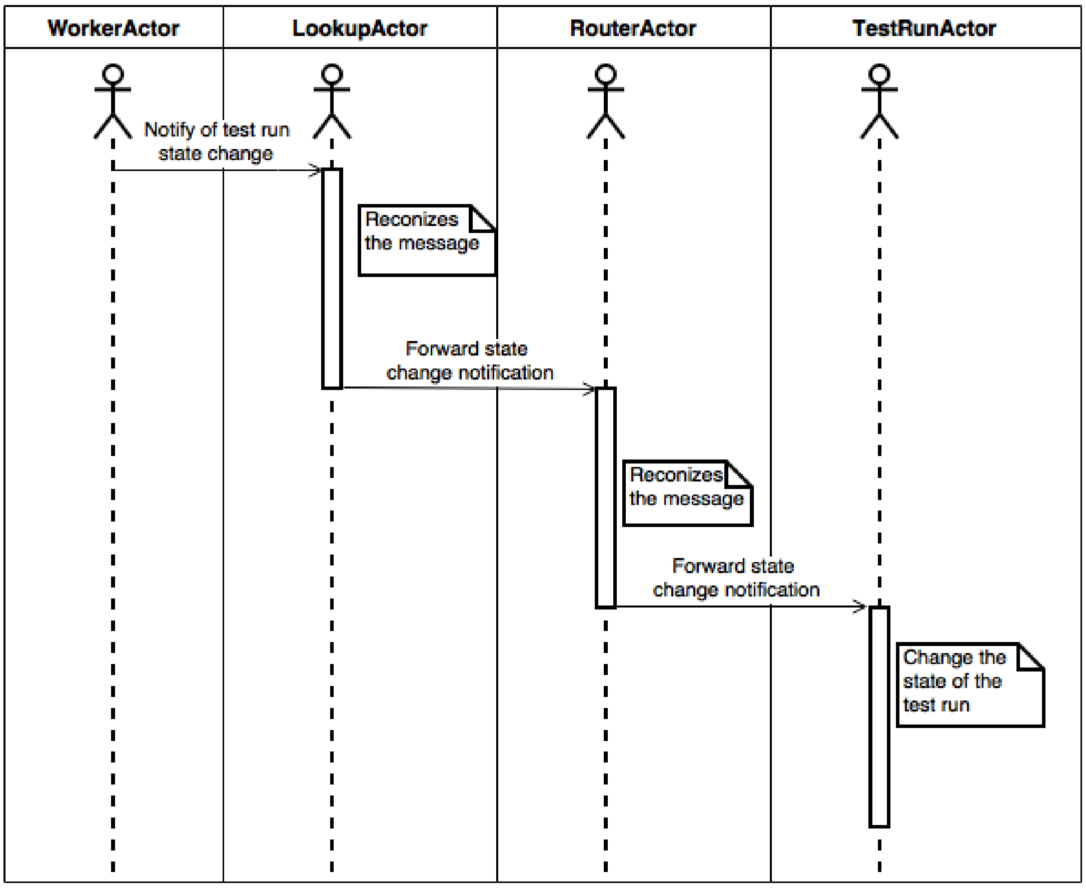
WorkerActor
LookupActor
RouterActor
TestRunActor
Notify of test run state change
Reconizes the message
Forward state change notification
Reconizes the message
Forward state change notification
Change the state of the test run

Figure 4.2: *The sequence of messages sent between the actors when a test run needs to change state*

The steps in the sequence diagram can be described as the following:

1. The *WorkerActor* internally registers what made the change
2. The *WorkerActor* notifies its corresponding *LookupActor* that a test run is changing state
3. The *LookupActor* recognizes the message should be forwarded to the *RouterActor* and does so accordingly
4. The *RouterActor* recognizes that the message exclaims a test run state change and will forward the message to the *TestRunActor*
5. The *TestRunActor* unpacks the whole message and will change the state of the specified test run if it is legal.

4.1.2 Actor Encapsulation

We realize that the actor model deviates from a classical object-oriented paradigm as covered by Bent Thomsen in a lecture on the actor model [31]. The actor model induces concepts that are unacquainted to us and it is unsure whether it will entail difficulties if Kuiga Box is to be almost purely implemented in the actor model. With this reasoning we chose to encapsulate the master actor system's functionality as illustrated in Figure 4.3, where the classes on the left-side contains the encapsulation of the actor system and can identify the actors relevant to its functionality. As a means of forcing a loose coupling from a potential use of these classes, we hide any class fields that would otherwise expose direct access to the master actor system. Figure 4.3 shows how some of the functionality of a *TestRunActor* is encapsulated by the *TestRun* class, that upon a method call will perform the required operations on the actor system in order to return a result to the callee.

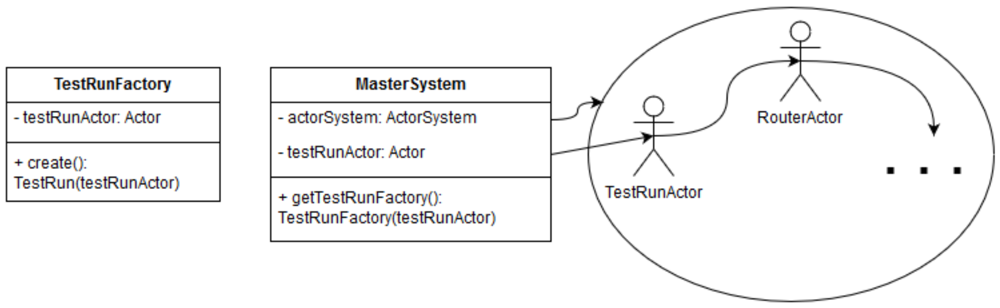
TestRunFactory
- testRunActor: Actor
+ create(): TestRun(testRunActor)
MasterSystem
- actorSystem: ActorSystem
- testRunActor: Actor
+ getTestRunFactory(): TestRunFactory(testRunActor)
RouterActor
TestRunActor
. . .

Figure 4.3: *Diagram showing how TestRunFactory and TestRun classes encapsulates the behavior of the actor system*

4.2 Inversion of Control

As a part of writing a well-structured application, we put emphasis on code reuse and testability. When using an Object-Oriented (OO) language this means decompositioning large classes into smaller classes with a single responsibility. This seemingly makes each individual class easier to test but classes will use and depend on one another. These dependencies will be expressed through composition or aggregation of the necessary classes which introduces a direct dependency from one class to another.

4.2.1 Achieving Testability

To actually make these decompositioned classes testable on their own, we need to remove the direct dependencies between classes. We achieve this by using Dependency Injection (DI); rather than letting a class provision its own dependencies, we provide these dependencies as input to the class. The dependencies are provided through interfaces that define the required functionality, which makes it possible to swap implementations e.g. with a mocked object when testing.

With the use of DI the classes are less coupled but it comes at the cost of more complex usage. When dependencies have to be provided as input to classes, it puts responsibility on the class consumer to resolve what the consumed class needs. This is not too much complexity for a single class but the dependencies likely have dependencies of their own. We see that we are no longer creating and consuming a single class but rather a hierarchy of classes. Listing 3 shows an example of this, but it is still a fairly flat hierarchy as only some of dependencies have dependencies of their own However, we observe that more complex hierarchies will significantly increase the complexity.

```
val svc = new ShippingService(new ProductLocator(),
    new PricingService(), new InventoryService(),
    new TrackingRepository(new ConfigProvider()),
    new Logger(new EmailLogger(new ConfigProvider())))
```

Listing 3: *Small dependency injection example*

We realize that it is indeed possible to do as in Listing 3 making the code reusable and testable, however, it is complicated to do so. Listing 3 has to be repeated each time this class configuration needs to be created, resulting in the need of copy-pasting that same snippet each time. Consequently, if we employ this solution for multiple classes, we risk achieving an opaque block of code that is frequently copied around.

Alternatively we move Listing 3 into a method of its own and call this method when we need an instance. This helps in avoiding the copy-pasting of opaque code blocks, but it is not very general as it requires a method per class as in Listing 4.

```
object MegaFac {
    def getShippingService(): ShippingService = ...
    def getProductLocator(): ProductLocator = ...
    def getPricingService(): PricingService = ...
    def getInventoryService(): InventoryService = ...
    def getTrackingRepository(): TrackingRepository = ...
    def getConfigProvider(): ConfigProvider = ...
    def getLogger(): Logger = new Logger(getEmailLogger())
    def getEmailLogger(): EmailLogger = ...
}
```

Listing 4: *Simple *container* object*

Separating the configuration and consumption of objects gets rid of the complicated construction from Listing 3 and it is the core principle behind the Inversion of Control (IoC) pattern; extracting control and configuration from the consumer, allowing the consumer to focus on consuming the class.

IoC frameworks provide separation of configuration and consumption through the use of containers which encapsulate a particular configuration. Like we did in Listing 4, IoC frameworks provide containers for specifying and encapsulating the configuration i.e. which implementations will be used. The framework then provides facilities for constructing object graphs from a configuration.

4.2.2 Google Guice

For Kuiga Box we are using an IoC framework by Google called Guice[32] (pronounced ”juice”). Guice allows us to create and combine configurations, allowing us to split configurations into entities called *modules* and combine these as needed. We may desire to swap different parts of the configuration for different program use cases or for testing purposes. This provides flexibility in situations where we may need to support different components in the program's environment while neatly encapsulating the configuration details.

```scala
class WorkerModule extends AbstractModule with GuiceAkkaActorRefProvider {
  override def configure(): Unit = {
    install(ThrowingProviderBinder.forModule(this))
    bind(classOf[WorkerAdapter]).to(classOf[DockerWorkerAdapter])
    bind(classOf[Actor]).annotatedWith(Names.named(WorkerActor.name))
      .to(classOf[WorkerActor])
    bind(classOf[Actor]).annotatedWith(Names.named(WorkerGuardianActor.name))
      .to(classOf[WorkerGuardianActor])
    bind(classOf[ActorFactory])
    bind(classOf[WorkerSystem]).to(classOf[WorkerSystemImpl]).asEagerSingleton()
  }

  @Provides
  @Named(WorkerGuardianActor.name)
  def provideWorkerGuardian(system: ActorSystem): ActorRef =
    provideActorRef(system, WorkerGuardianActor.name)

  @CheckedProvides(classOf[DockerWrapperProvider])
  @throws(classOf[DockerCertificateException])
  @throws(classOf[InterruptedException])
  @throws(classOf[DockerException])
  def provideDockerWrapper(dockerWrapper: DockerWrapper): DockerWrapper = {
    dockerWrapper
  }
}
```

Listing 5: *Guice module with WorkerSystem configuration*

Listing 5 shows an example of how Google Guice is used in Kuiga Box. The code excerpt shows the configuration concerned with the actor system encapsulation on the worker. It binds various classes, configuring mappings from an interface to an implementation and potentially binding an implementation to itself. The latter is a technical detail of Guice as the framework must know how to construct all objects in a graph.

There are no limitations to what we can or cannot bind and a single module need not contain bindings for the entire graph, it is only when combining one or more modules into a 'factory', called an *injector* in Guice, that the modules are consumed and constraints are checked when using this injector to create object graphs. This use is shown in listing 6; we create an injector by combining 3 modules, the 'WorkerModule' from listing 5 and two others.
We can construct object graphs by calling the *getInstance(class)* method on the injector, requesting instances of the root objects in the object graphs we need. Guice will then take care of constructing the requested class, as well as any classes required by the root class.

```
Injector injector = Guice.createInjector(
        new ConfigModule(ConfigModule.getConfig()),
        new AkkaModule(),
        new WorkerModule()
);

WorkerSystem system = injector.getInstance(WorkerSystem.class);
DockerWrapperProvider provider =
    injector.getInstance(DockerWrapperProvider.class);
```

Listing 6: *Guice injector usage*

Guice has many options for customizing how the configuration is interpreted and several facilities for providing dependencies. The simplest is through the constructors and as we are using DI this becomes very natural. In order for Guice to understand that it should inject dependencies, the constructor must be annotated with *@Inject()* as in Listing 7.

```
class DockerWorkerAdapter @Inject()
    (dockerWrapper: DockerWrapper) extends WorkerAdapter {
}
```

Listing 7: *Guice Inject annotation*

4.2.3 Summary

We use Google Guice to great effect to abstract from configuration details. This allows to more easily unit test classes as we easily swap one or more components to facilitate this testing. The use of IoC also facilitates easy configuration of the software for different use cases, allowing us to support multiple component combinations with little effort.

4.3 Kuiga Plexer

The implementation of the Kuiga Plexer is two-fold, first the implementation of the binary that captures and logs data units from the virtual instances and secondly the network configuration that allows the data units to travel from the virtual instances to the Kuiga Plexer. The network configuration, that will be presented, is rather reliant on Docker and its network configuration, but the network configuration method employed by Docker should be applicable with other virtualization methods as well.

The Docker network configuration supports communication between the host and containers by creating Docker networks, that containers can be added to. Docker uses network bridging to connect the container networks with the host network. Bridging is defined as the creation of a single logical network from from two or more networks [33].

Docker connects these networks to the host using virtual Ethernet (vEth) interfaces. The vEth constructs are created in pairs, and they are commonly used to connect containers to hosts. Docker creates these vEth endpoints during the container creation process. One of the vEth endpoints are placed inside the Docker container while the other end of the *tube* is placed by Docker on the container host. Docker then binds the vEth endpoint residing on the host to the docker0 interface which Docker bridges with the host's eth0 interface. Figure 4.4 illustrates this setup.

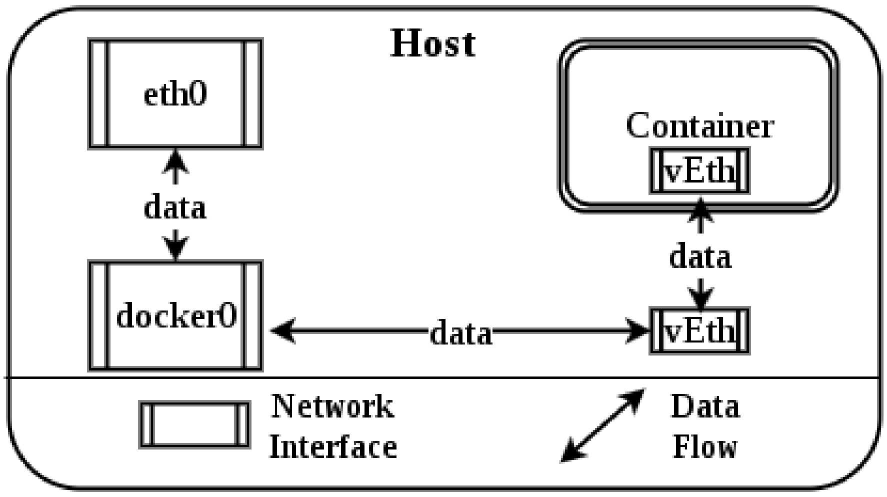

Figure 4.4: *The Docker network configuration*

Docker supports creation of custom networks with their own bridges, like docker0, each with their own subnet. We use this feature to create two container networks namely the networks *Radio* and *Control*. We add the *Radio* interface as the default route inside the containers such that all Internet traffic is routed through this interface. We dedicate the *Control* interface to connections from the host to the container such as SSH connections.

We assign the *Radio* network to handle traffic to and from the SDR subsystem. This traffic is of particular interest to us from a testing and debugging perspective, as we want to log the data sent through this interface. We do this by capturing packets on the *Radio* interface for then to log them in a file. Figure 4.5 shows the configuration of the network setup.

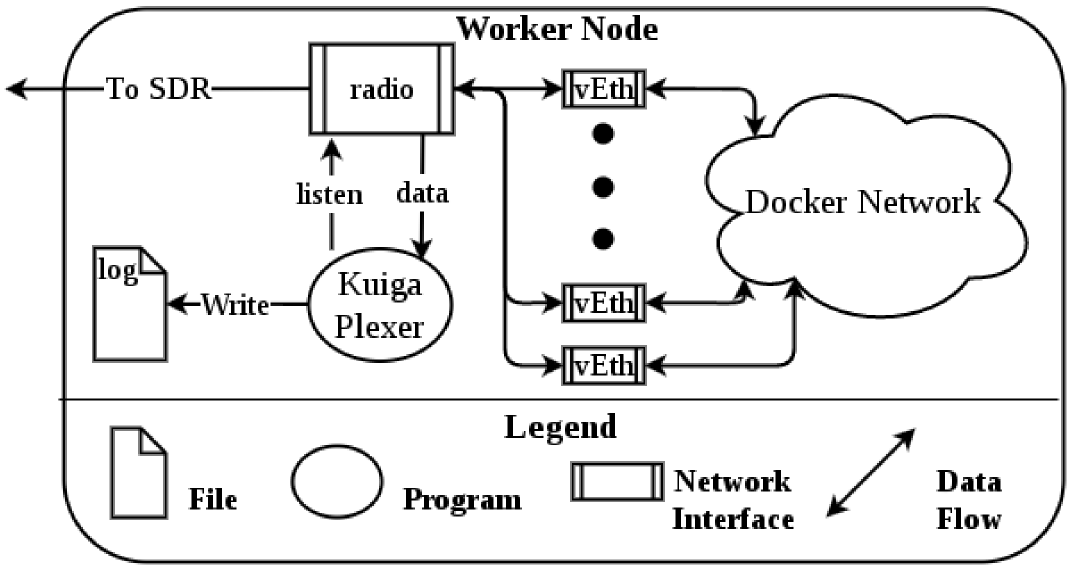

Figure 4.5: *The logging and radio network setup configuration*

The data units Kuiga Plexer receives are frames which are used by layer 2 OSI stack protocols. We intentionally chose to capture frames since Docker applies an address translation (Network Address Translation (NAT)) to the layer 3 (IP protocol) packets. The NAT removes our ability to discern from which container each package is originating. Capturing frames also has the added benefit of letting us retain some flexibility since we can always unpack higher level data units in accordance to what the SDR subsystem requires regarding data units.

The Kuiga Plexer itself is quite simple as all it does is create a raw ethernet socket and bind it to the of the *Radio* interface.

We use the Tins [2] library, which provides the functionality to capture packets. We use the library to capture packets from the *Radio* interface.

4.4 Image Store

As described in Section 3.4, the Image Store is the component that stores base images and runnable images. The Kuiga Master must be able to retrieve these base images from the

[2]Tins library: `http://libtins.github.io/`

store and upload runnable images to it. The Kuiga Worker nodes only needs to download runnable images. The following sections investigate which technologies can be used for the Image Store and how the Image Store has been implemented.

From the design of the Image Store we see that the implementation must support the following features

- Must be accessible from the master and all worker nodes
- Must support persistently storing image data under a identifier provided to the Image Store
- Must support retrieval of image data when provided the identifier

Examining these features we recognize that the set of features is similar to the features supported by object storage technologies such as Amazon Simple Storage Service (Amazon S3). Object storage technologies exhibit the following characteristics

- Objects are identified by a user-provided key
- Objects are opaque and the contents are unknown to the object storage
- Objects have user-provided metadata associated
- Objects are grouped into containers/buckets identified by user-provided keys
- The object storage supports storage, retrieval and deletion of objects
- The object storage is designed to be consumed directly by applications

While this set of features is more extensive compared to the features required of the Image Store we still find that the concept of object storage is useful. Using an object storage technology allows us to abstract from the details of how the Image Store data should be stored on a block device or file system. Additionally, having the object storage technologies designed to interact with applications should simplify the development of the Image Store component.

4.4.1 Object Storage Technologies

For object storage implementations we identify two types:

- Embedded object storage such as *Google LevelDB* [34] i.e. as a library integrated into an existing application used through an API
- Standalone object storage systems i.e. running in a self-contained application used in a client-server architecture

The embedded type is not desirable to use as we must support access from several applications running on several physical devices. The embedded type cannot do this on its own

so we must implement functionality on top of the embedded object storage to make it accessible from all nodes.
The standalone type of object storage is exactly what we need. The client-server architecture is perfect for the multiple clients we must support, it supports the intended workflow of passing image references from the Kuiga Master to the Kuiga Workers, allowing the individual Kuiga Worker to resolve the reference to obtain the image data.

Based this we choose to look at two technologies, Ceph[35] and SeaweedFS[36]. We have chosen to further examine these two technologies based on an initial study of the technologies available. We believe that these two technologies represent the competent implementations of two different approaches.
We examine the technologies with an emphasis on the complexity of configuring, consuming and maintaining as well as the maturity of the technology and the quality of the documentation.

Ceph

Ceph is a distributed object store which also supports exposing itself as a block device and a file system, however only the object storage functionality is relevant for the Image Store. Ceph is open-source and is free of cost to use, it is maintained and actively developed by Red Hat, the owner of the popular commercial linux distribution, Red Hat Enterprise Linux (RHEL).

Ceph is a mature technology with a large community and numerous large-scale commercial deployments. The maturity is reflected in the excellent documentation which is easily navigated and searched.
Ceph supports a wide range of uses and configurations which makes it able to support new uses, allowing greater maintainability for the Image Store making use of Ceph. However this generality of Ceph also negatively affects maintainability as the generality makes the system more complex which increases the threshold of knowledge required to perform maintenance.
This same generality also affects the complexity of configuring the system, however this is mitigated by excellent documentation as well as official Docker images. These factors serve to make Ceph quick to deploy with a reasonable default configuration.

As mentioned Ceph can be consumed through its object storage interface, as a block device or as a file system. Building on the object storage interface Ceph also provide clones of the OpenStack Swift Representational State Transfer (REST) API and the Amazon S3 API which allows Ceph to be swapped into systems where either OpenStack Swift or Amazon S3 is used.
What this means for us is that we can use client libraries for either of these, with Amazon S3 receiving official support from Amazon for a wide range of languages[37], including Java and consequently Scala.
This makes it very simple to consume Ceph.

SeaweedFS

SeaweedFS has a different approach to the concept of storing *key, binary data* pairs with some interesting compromises. We include it because of these compromises as we find them potentially desirable.

SeaweedFS is not quite an object store based on the characterics we outlined in 4.4.1.
It does not support user-provided keys, user-provided metadata and the concept of containers.
These features are not strictly required by the Image Store and their absence allows SeaweedFS to be significantly simpler than traditional object storage technologies.

This reduced complexity simplifies deployment and with very little configuration possible, SeaweedFS can be deployed in a few minutes. This also comes into play for maintainability as it lowers the threshold of knowledge required to perform maintenance, but conversely the loss of generality negatively impacts maintainability due to the reduced flexibility making changes and additions more difficult.
The lack of support for user-provided keys increases the complexity of the system consuming SeaweedFS as the consuming system must separately handle mapping from its chosen key to the key provided by SeaweedFS.
Consuming SeaweedFS is more complicated than consuming Ceph as it requires several more steps for both storage and retrival of objects, as well as requiring the use of SeaweedFS's custom interface which is generally unknown due to the limited number of deployments of SeaweedFS.

The limited number of deployments is also part of the cause and an effect of the low maturity of the SeaweedFS project.
The project is maintained and developed by a single person in their spare time and has a very small community. This is also reflected by the state of the documentation as it is not comprehensive and part of it are outdated

Selecting an Object Storage technology

We decide to use Ceph to provide the data storage for the Image Store component.

Ceph wins in terms of maturity of the software and documentation and it is simpler to consume. These advantages combined with the large community, which makes it easier to get help, are the reason for our choice.
We recognize the merits of SeaweedFS, especially the simpler design makes the behavior more predictable, however in its current state and considering the low maturity, we opt not to use SeaweedFS and rather use Ceph.

4.4.2 Integrating Ceph

As described Ceph supports being consumed through a clone of the Amazon S3 REST API. We opt to integrate Ceph into the Image Store component through this interface because of the quality and active development of the official client SDK provided by Amazon.
As this SDK is comprehensive, well documented and mature it simplifies the integration of Ceph. While the native object storage client for Ceph is also mature and well documented it requires the presence of external components in the environment which the client is run in. To make the Image Store as convenient and modular as possible we have decided to make use of Amazon Web Services (AWS) SDK.

4.4.3 Supporting Image Store Functionality

When Kuiga Master needs to create a runnable image it has to get the base image first. Kuiga Master acquires the base image it builds the runnable image by combining the base image with the AUT as described in Section 4.5. The runnable image must then be stored by the Image Store. This is where the *uploadImage()* function comes into the picture. Prior to this, the image has been checked for invalid naming as we impose some restrictions on the naming. The implementation of *uploadImage()* is shown in Listing 8. First we make the name, which defines the image on the store, in the right format, to have a way to identify the image with the right OS and architecture. Then we create a container on the object storage, so we have a place to store the image. We then use *s3Client.putObject()* to store the image in Ceph.

As the Amazon S3 imposes access control we need to allow the Kuiga Workers to access the stored image. We do this by adding *PublicRead* to the Access Control List (ACL), an alternative, and more secure approach, would have been to create a different set of credentials for the Kuiga Workers and add read-only access for these credentials to the ACL of the object. Once the image has been made available on the object store, we return the URL pointing to the image. This URL can then be provided to the Kuiga Workers to allow downloading the image.

4.5 Building Docker Images and Containers

This section intends to explain the process of creating a test run, from building the Docker image on Kuiga Master to creating Docker containers on the worker nodes from this image. This process follows the steps described in Section 3.4 on how runnable images are created on Kuiga Master and then downloaded to Kuiga Worker.

We use a open-source Docker Client library [3] on both the Kuiga Master and Kuiga Worker

[3]Docker Client on GitHub: `https://github.com/spotify/docker-client`

```
private def uploadImage(image: Image, inputStream: InputStream): Option[URL] = {
  val imageNameAndArch = s"${image.name.toLowerCase}.${image.architecture}"
  val bucket = s3Client.createBucket(imageStorePath + image.osName.toLowerCase)
  val imageFile = createFile(inputStream, imageNameAndArch)

  try( {
    if (imageFile.isDefined) {
      s3Client.putObject(bucket.getName, imageNameAndArch, imageFile.get)
      s3Client.setObjectAcl(bucket.getName, imageNameAndArch,
      CannedAccessControlList.PublicRead)
    }
  } catch {
    //Error handling
    }
  }
  return Some(getImageURL(image))
}
```

Listing 8: *Implementation of the* uploadImage() *function*

to communicate with the Docker client API. The library provides all the necessary Docker API calls as Java methods such as start, stop and remove container.

Please note, there is some overlapping terminology between Docker and the terms we have defined in this paper. The terms may be closely related but still have a slight difference in their definition e.g. both domains define the terms image and base image, which in the Docker context is a *Docker images*, while the terms are broader in this paper, where they cover both actual *Docker images* but also images used in other types of virtualization.

The Kuiga Master uses Docker to create runnable images (in this case *Docker images*), which it uploads to the Image Store. Docker images are created from a *Dockerfile*, a text file containing instructions on how Docker should build the image, and a context, the set of files in the same directory as the *Dockerfile*. When Kuiga Master builds a Docker image, the context contains the AUT and a start up script that, when the Docker container is started on the Kuiga Worker, performs some network related configuration and launches the AUT.

Listing 9 shows a snippet of the code that creates Docker images (runnable images). The method makes use of a *DockerWrapper* object, which contains methods that call the Docker client API with the provided parameters. The *buildImage()* method is responsible for building the Docker image (runnable image). As the first parameter it takes a path to the directory containing the *Dockerfile* and context that should be built. The second parameter is simply the name of the image. If the building process is successful, the newly created runnable image is saved to the Image Store. The call to the *saveImage()* method returns an *InputStream* of a image tarball created by the Docker client. It is this tarball that is saved to the Image Store, where it later can be downloaded by a Kuiga Worker and loaded into the worker's Docker client. Lastly the Kuiga Master removes the image from its Docker client, as it is of no use to the Kuiga Master; the runnable images are all available on the Image Store.

```
def createDockerImage(): Unit = {
...
 val dockerImageId = DockerWrapper.buildImage(dockerPath, imageName)

 if (dockerImageId.isDefined) {
   ...
   imageInputStream = DockerWrapper.saveImage(dockerImageId.get)
   DockerWrapper.removeImage(dockerImageId.get)
 }
}
```

Listing 9: *createDockerImage() method from the ImageFactoryImpl class. The method is responsible for creating Docker images on Kuiga Master*

When the Kuiga Master is done creating and uploading the tarball of a Docker image, it sends a *CreateTestRun* message to a Kuiga Worker. This message contains information about the image and test run that should be executed on the worker node. Among the image information is a download URL for the tarball on the Image Store. Listing 10 shows the *loadImage()* method, which firstly calls the *downloadFileImage()* method to download and store the runnable image from the Image Store. As previously mentioned, the runnable image is saved as a tarball of a Docker image and this tarball can easily be loaded into the Docker client, where it can be used to create new Docker containers. The tarball is saved locally and should be deleted when it has been loaded into the Docker client (or if it fails).

```
private def loadImage(fileUrl: String, fileName: String): Unit = {
  val tempPath = DownloadHandler.downloadFileImage(fileUrl, fileName)
  try {
    dockerWrapper.loadImage(new BufferedInputStream(
        new FileInputStream(tempPath)))
  } finally {
    Files.deleteIfExists(Paths.get(tempPath))
  }
}
```

Listing 10: *loadImage() method of the DockerWorkerAdapter class. The method handles download of image from the Image Store and loading it into the Kuiga Worker's Docker client*

The last step in the creation of a test run, is to create the requested number of containers (virtual instances). In Docker every image and container has a unique id, so creating a new container is simply a matter of calling a method, with the id of the image to create the container from. Listing 11 presents the *createContainers()* method, which creates an amount of containers from a specified image id. The method returns a collection of container ids, so the Kuiga Worker can keep track of which containers are related to a test run.

```
private def createContainers(amount: Int, imageId: String): Seq[String] = {
  var containers = Seq[String]()
  for(i <- 0 until amount) {
    containers = containers :+ dockerWrapper.createContainer(imageId)
  }
  return containers
}
```

Listing 11: *createContainers() method of the DockerWorkerAdapter class. The method handles creation of new containers from a specified Docker image id*

This section presents parts of the process on how test runs are created using Docker. Listing 10 and 11 present two methods from the *DockerWorkerAdapter* class, that extends the *WorkerAdapter* trait seen in Listing 12. The Kuiga Worker requires an adapter, that handles the creation of virtual instances. In this case Kuiga Worker uses the *DockerWorker-Adapter*, which handles the creation of virtual instances using Docker, but the adapter can be swapped with an adapter that for instance can create virtual instances using Xen or QEMU.

```
trait WorkerAdapter {
  def createTestRun(id: Int, amount: Int,
                    shouldStart: Boolean, fileUrl: String): Boolean
  def startTestRun(id: Int): Boolean
  def stopTestRun(id: Int, shouldDelete: Boolean): Boolean
  def deleteTestRun(id: Int): Boolean
  def testRunStatus(id: Int): String
}
```

Listing 12: *WorkerAdapter trait*

5 Tests

In this chapter, we will elaborate on how testing can be executed on Kuiga Box as a means of evaluating the implementation of Kuiga Box to its requirements. We do this by presenting how software verification is relevant to Kuiga Box and by presenting a performed load test.

5.1 Software Verification

As a part of any software development, verifying that that the software system works as intended is an important part of the process. This verification can be performed in a number of ways with a common method being tests.
These tests should be representative, comprehensive, and repeatable as these three characteristics are necessary if we are to reason about the about the correct functioning of the software system.
While these characteristics are sufficient for reasoning about a final version, the tests benefit from being easily performed for the purpose of using tests to verify the software as it is created. Testing during the process can be help to guide the development and catch problems before the problems manifest into larger, harder to solve issues.

Representativity is relevant as the tests represent a constructed environment and it is important that effort be put into properly mimicking the environment the software is intended to function within. Constructing a perfectly mimicked environment can be impractical, as a constructed test environment is often created to make it actually feasible to perform the test. This can be because of the cost or effort associated with creating and maintaining a real environment. This forces a trade-off on making the test representative and a solution is to narrow the focus of the test and go with an approximation that covers the facets of the real environment.

It is important that the tests are comprehensive so that the tests will cover all uses of the software system. Simply showing that the software system works correctly when utilised in one way is not sufficient to reason that the software function correctly. The tests must show that the software functions correctly for all intended uses and that it fails appropriately for inappropriate uses.

Ensuring that the tests are repeatable is necessary to make the tests and results tangible and to convey both. If the tests are not repeatable the results will merely be a claim by us, the tester, as to the merits of the software system under test. By ensuring that the tests are repeatable we can also use the tests to test for regression as we develop the software system. This is useful to make sure that mistakes are not repeated and that faults already removed are not reintroduced.

Allowing the tests to be easily performed avoids that the tests are skipped because they are deemed to costly to perform for a small number of changes. If the tests are only performed a larger amount of changes has accumulated, it increases the time that passes until an introduced fault is detected. This situation is undesirable as it decreases the value of gained from creating a comprehensive suite of tests.

5.1.1 Testing Kuiga Box

In order to test and verify Kuiga Box we can see that we must achieve several characteristics for the tests created. To accomplish this we employ a bottom-up approach using multiple methods of testing. This approach is inspired by the software development model, V-model, as presented by Sommerville [25, p. 60]. We find inspiration in the models systematic, bottom-up approach to testing and less from the complete development process. While the V-model is well known it aligns more closely with a plan-driven approach to software development, despite attemps to make the model more agile. We find this undesirable as the exploratory nature of the project demand that we are able to quickly shift the direction.

By first testing at a lower level, we can verify the correctness of the smallest unit, as we move up the errors are likely to be caused by the interactions introduced by the higher level. Starting the testing at the unit level enables us to write tests as the software system is being implemented. With this we can put more emphasis on continuously testing instead of relegating testing to the end of the project, which is an often critisized shortcoming of practices inspired by the V-model despite claims that the model can support an iterative development process. In this manner we can gradually increase our confidence in the correct functioning of Kuiga Box.

Starting at the lowest level we utilise unit testing through the use of ScalaTest[38]. We create test fixtures to test individual classes and methods on these classes. As described in Section 4.2 Dependency Injection (DI) allows us to separate the class from its dependencies by injecting simulated dependencies. This enables us to test the class on its own, however it poses an issue for the representativity of the test. The simulated dependencies may be overly simple and risk not accurately simulating a 'real' object. Despite this possible divergence unit testing is still attractive as the programmed nature of the unit tests mean they are repeatable and easily performed. Unit tests are also fairly quick to write however some care has to be taken to ensure that the suite of tests for a method express a comprehensive test of the method.

Ensuring that unit tests are comprehensive is very difficult as unit tests only allow us to reason about the absence of the specific faults covered by tests. This combined with the

questionable representativity of unit tests is why we must move to a higher level. After testing the individual unit as comprehensively as imaginable, we need to verify that the units function together correctly. For this we look to multiple levels of integration testing.
Initially we integrate a few classes, using simulated classes for the remaining dependencies. As we encounter and remove faults from the software system we gradually integrate more classes until we have verified the inner workings of individual system components.

From the unit to the component level we continue creating and automating our integration tests as we did for our unit tests. When testing the integration of several components we should strive for automated tests but the tests may be complex to automate, involve external components or interact with the software system's environment. For Kuiga Box these complications are even more numerous due to the distributed nature of the system.

```
class MasterAndWorker$Test extends FunSuite
    with BeforeAndAfterEach with BeforeAndAfterAll {
  val injector = Guice.createInjector(
    new MasterTestModule(),
    new WorkerTestModule())

  var masterSystem: MasterSystem = _
  var workerSystem1: WorkerSystem = _
  var workerSystem2: WorkerSystem = _
  var workerSystem3: WorkerSystem = _

  override def beforeAll(): Unit = {
    workerSystem1 = injector.getInstance(classOf[WorkerSystem])
    workerSystem2 = injector.getInstance(classOf[WorkerSystem])
    workerSystem3 = injector.getInstance(classOf[WorkerSystem])
    masterSystem = injector.getInstance(classOf[MasterSystem])
  }
}
```

Listing 13: *Automated Master and Worker integration testing*

Listing 13 shows the setup of integration testing between the Kuiga Master and Kuiga Worker components. These tests utilise the same ScalaTest framework as is used for unit testing and the tests are equally independent of the system configuration.
These automated integration tests abstract from the details of communications and the distributed components are all present on the same system. We can see that this simplification make the tests unsuitable for testing how the system is affected by external factors and are thus not suitable for testing the software system's fault tolerance against imperfect communication.
Testing Kuiga Box in a representative environment is more attractive to do manually due to the complexity of configuring this environment.

With this scheme of testing we present an approach to verifying Kuiga Box and gain confidence in the correctness of the software system. The testing scheme is mostly directed towards testing correct functionality of individual components and correct interactions. For the interactions between the distributed components the approach suffers from the less rigorous manual testing and the complexity of testing all possible state transitions with ”unit testing”-style integration tests while accounting for the environment affecting the test.
To increase our confidence in Kuiga Box the interactions between components, which escape the process, can be modelled and model checkers be used to verify that the system functions and fails correctly when factoring in influences by external factors.

5.2 Load Test

We decide to conduct load testing on Kuiga Worker and Kuiga Plexer components. The MoSCoW analysis in Section 1.3, states that Kuiga Box should be able to virtualize at least 100 devices. We want to, with this in mind, determine the boundary of how many virtual devices Kuiga Box can handle on a single worker node. This is where the load test comes into the picture since the load test is a process in which you as the developer put an increasing amount of demand on a system, to determine how the system reacts to different levels of load conditions. In our case we want to understand certain key metrics of the system performance while running a varying amount of Docker containers. We define the overall object of the load test to be: *How many virtualized containers can the current version of Kuiga Box handle?*

We chose to perform load testing on Kuiga Worker and Kuiga Plexer since these are the parts that currently involve the issue of scalability. We see this in Section 3.2 where the worker node is shown to be able to be scaled. The interest of load testing with regards to scalability is expressed in the possibility of performing multiples of the same load test but with varying amount of workers. The Kuiga Master will in the future warrant tests of it seeing as how it is intended to be able to manage varying amounts of worker nodes, but it is excluded for now.

As we want to reduce the amount of factors effecting the test, we decide to split the test into two smaller parts, one testing the worker node and its virtualization capability, and another testing the capabilities of the Kuiga Plexer. Hence the load test is divided into two tests with different objectives:

1. Characterize the workload of containers the worker node can handle, before the performance deteriorates.

2. Define the package throughput Kuiga Plexer is able to capture.

5.2.1 Test 1: Worker Node Load Test

In the first test, we want to see how many containers a single worker node can handle. For this, there are three relevant parameters to keep in mind:

- The amount of containers running on the worker node which we define as the load
- The resource demand of the executing AUT
- The degree of synchronization between the containers' AUT deadlines

In this test, we want to increase the load of running containers while keeping the AUT as an unchanging parameter, hence all containers are executing the same AUT between the various loads. We also aim to synchronize the AUTs start, such that they start executing at the same time. This enforces a worst case scenario, as all the containers wake at the same time and execute their code or at least until they drift apart. We allow the AUTs to execute lasting three minutes after being awakened. This provides an approximate of 180 data measurements from each container.

The test application mimics an application that sleeps in-between periods of activity in which it may transfer data using a wireless connection. We argue that although Kuiga Box will support a lot of diverse applications with various behaviours, we find that this behavior is typical of a battery conserving IoT applications, explained in detail in Section 2.1

We propose the pseudo code for such an AUT in Listing 14. The AUT is meant to emulate an IoT application, that is active in a period of one second for doing a few simple instructions, in our case, it is writing to *stdout*.

At Line 6, the AUT is set to sleep until the *startTimeMs* is reached, which synchronizes the containers such that they wake up at the same time. The AUT starts with calculating the sleep until its next period at Line 12, and print its data to *stdout*

```
void main(var startTimeMs) {
    var counter <- 1
    var periodMs <- 1000

    var firstSleepMs <- startTimeMs - GetCurrentTimeMs()
    Sleep(firstSleepMs) // sleep sync

    while(true) {
        var currentTime <- GetCurrentTimeMs()
        var sleepTimeMS <- startTimeMs + (counter * periodMs) -
                           currentTime
        Sleep(sleepTimeMS) // sleep period

        var wakeupTime <- GetCurrentTimeMs()
        Print(wakeupTime, sleepTimeMS, startTimeMs)
        counter++
    }
}
```

Listing 14: *Pseudo code for the AUT of the first test*

Select Performance Metric

We want to see when the AUT exceeds its expected deadlines, when we step-wise increase the load of containers running the AUT. The performance metric used for this test is: The deadline jitter of the AUTs at a certain amount of containers. The formula for calculating this for a single container's deadline can be seen in Equation 5.1.

$$\text{pMetric} = |\,\delta_{wakeUp} - (\delta_{start} + (n * \delta_{period}) - \delta_{now})\,| \tag{5.1}$$

δ_{wakeUp} is the current time the container awakes from its sleep. It is in the Unix epoch time[1] format in ms.

δ_{period} is the execution period in ms which the AUT is executing within.

δ_{start} the start time used by all AUTs, to synchronize their start of the test. It is in the format of Unix epoch time in ms.

δ_{now} the current Unix epoch time in ms of the AUT when writing to *stdout*.

n is an integer representing which deadline is calculated, ranging from 0 to 180 (which is the number of allowed execution seconds of the AUT after δ_{start}).

[1]Also known as UNIX time, which is seconds since the 1 January 1970

Data Collection

We use the Docker logging interface to collect the test data which provides a downloadable log file, containing the *stdout* for each container. We use the data from all the containers of each test case, meaning some test case have more data than others. As we are expecting each container to behave in the same way, the number of logs used should not play a huge role.

Expected Result

We expect that the jitter increases as the number of containers increases since each container is competing for the same resources. We naturally assume to see the lowest amount of jitter is recorded while running only one container.

Test Execution

We execute the test with a baseline of a single container running the AUT and a set of 10 test cases with an incrementation of 50 running containers for each test case.

We start a single test case at a time, and for each of these all the containers are created and started with a future start time, where the AUTs should start executing the test code. we stop the tests, when the test has run for 180 seconds from the start time, and download their log files. After downloading the files, the containers used for the test are deleted, as a means of cleaning up before the execution of the next test case.

We execute the tests on a laptop with the following hardware specification:

- RAM: 2x 2GB PC3-10600 CL9 - Kingston SNY1333D3S9DR8/2G
- CPU: Intel i3 350M Quart-core (3m cache, 2.26GHz) - Stock settings
- Disk: A used 500GB 2.5' Western Digital Blue - WD5000BEVT

5.2.2 Test 2: Kuiga Plexer Load Test

The purpose of this test is to determine how many packets Kuiga Plexer *misses* as a function of how many entities are sending traffic as well as the frequency at which they are sending data units. The entities sending data are simple programs that are instructed to send packets to an interface with a given frequency. We intentionally avoid using our virtual instances as they are in this case irrelevant to the test.

Please note that we did not perform the test due to time constraints but we will still present the rest of the test design for clarity and to let future developers perform the test themselves.

Test Setup

The test setup requires a minimum of two computers, where one is running the Kuiga Plexer component and others are running the packet sender programs. The number of packet sender programs should match the different amount of virtual instances, the final solution supports to get the representative results. Each packet sender program would ideally be executing on its own dedicated computer, to not interfere with each other i.e. use up each other shared resources, but this is not feasible in our case since the number of packet sender programs should correspond to the number of virtual instances Kuiga Plexer should support.

Further, to perform the test some network configuration is also required. The Kuiga Plexer needs to be capturing packets from a dedicated interface that is used by no other part of the computer hosting the Kuiga Plexer. The packets sent to the Kuiga Plexer host has to be routed from the receiving interface to the dedicated interface.

Select Performance Metric

We are interested in the amount of packets the Kuiga Plexer is missing at different levels of load. A packet is considered missing when a sent packet is not logged by Kuiga Plexer, while it is capturing and logging packets. We need to know the amount of packets the Kuiga Plexer successfully captured and the amount of packets sent in total from the packet sender programs, in order to calculate the amount of missed packets. The *missed* metric is calculated by the equation shown below 5.2.

$$missed = captured - \sum_{i=1}^{n} sent_i \tag{5.2}$$

n is the number of packet sender programs that send packets to the Kuiga Plexer.

$sent$ is the amount of packets sent from a packet sender.

$captured$ is the amount of packets that the packet sender captured.

5.2.3 Data Analysis

As mentioned in Section 5.2.2, we are not conducting the test regarding Kuiga Plexer, hence this section will only reveal the results of the load test executed on the worker node.

We illustrate the test results of the deadline jitter as boxplots in Figure 5.1. In the plot, we have chosen to show the median value, as it is a more robust statistic measurement compared to the mean measurement, as the mean result tends to be affected by skewness. Skewness is a statistical term where a few data point is having a high influence on the data set. We experienced a few high outliers, which could affect the mean calculation, making it less representative of the majority of the data set. It can be seen in Figure 5.1, that the maximum value is higher than the median value, which would skew the jitter to have a higher *ms* value.

In Section 2.4.2 it is elaborated that we want to be able to manage at least 100 virtual instances. In Figure 5.1 we observe that at a load of 100 containers running the AUT, described in Section 5.2.1, 75%(50% from the box and 25% from the lower whiskers) of the deadlines has a jitter between 0 to 6 *ms*, with a median value of 2 *ms*. Compared this to a single container running the AUT there is some extra overhead when the amount of containers running is increased.

We can further observe that the overhead is almost linear in regards to the increasing amount of load in Figure 5.1. The fact that the jitter is almost linearly increasing and none of the containers crashes or provide dirty data throughout the tests can reflect that we have not used a resource demanding AUT.

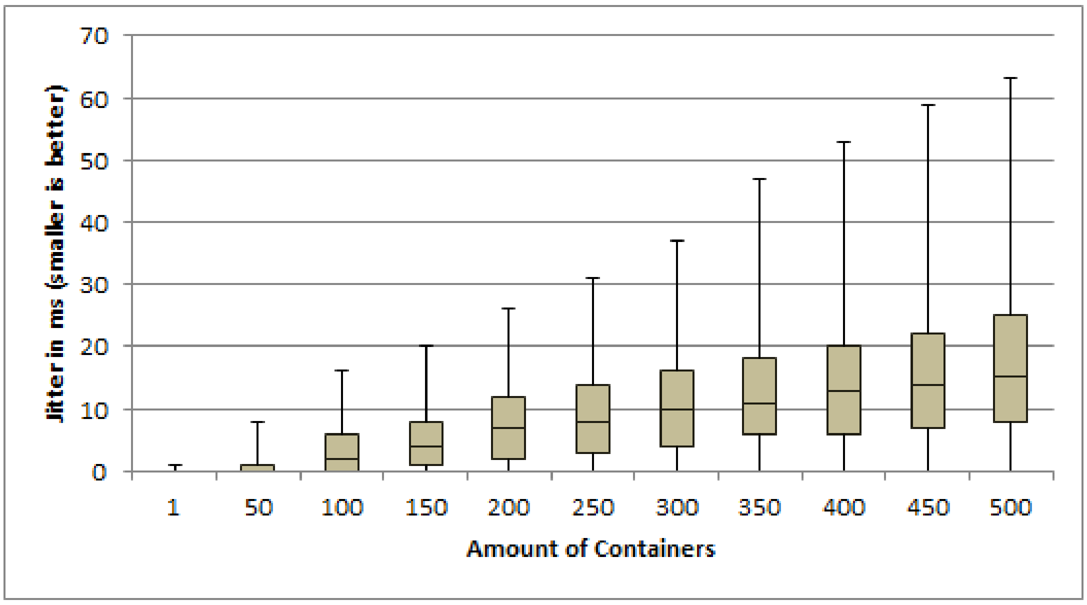

Figure 5.1: *The test result of the load test of a single worker as a boxplot. The whiskers represent maximum and minimum.*

The test result cannot give us a definitive answer of how many containers Kuiga Box can handle, as the result only reflects how the particular AUT performs, whereas a more CPU extensive AUT could potentially provide drastically different jitter results. It is also not known to us which requirement the user can have for their tests, meaning for some tests the small jitter from, for instance, 100 containers could be insignificant whereas for others it could invalidate the tests. The test result can, however, tell us that for our given worker node and our given AUT, we can at least virtualize up to 500 devices at the cost of higher *ms* deadline misses. The test can also be used as a baseline for a single worker which can be compared with the test outcome from a distributed worker system for Kuiga Box.

6 Discussion

We use this chapter to reflect on the parts of Kuiga Box we find interesting to discuss after the development process. We will discuss the uses of actor model in Kuiga Box, the shortcoming of the test run abstraction and work delimitation used for this project.

6.1 Extensive Actor Model

We describe, in Section 4.1.2, how we implement a series of classes that encapsulate the actor model and thereby hide the internal representation of the actor system. We do this as a means of exposing features in a classical object-oriented way which we expect improves the readability of the code. However, we experience little to no improvement in readability from doing this, and it is as such, arguable whether or not the effort is worth it.

In retrospect, removing this encapsulation and instead concentrating more on using the actor model more freely will not decrease readability or writability of Kuiga Box's code. Moreover, it is also possible that this will improve the development process, because instead of spending time maintaining this encapsulation, we may instead spend it improving the actor model implementation.

6.2 The Test Run Abstraction

As we mention in Section 3.10, we create a *test-run* abstraction. We use the abstraction throughout the system in both the master and worker nodes and we find that it works well to simplify the use of the system from a user's perspective.

In retrospect, the *test-run* abstraction introduces cohesion-related issues regarding the implementation of Kuiga Box, because there is no separation of test runs and virtual instances. We argue that it will improve cohesion and separation of concern significantly if a virtual instance has its own encapsulation. Moreover, the functionality required by Kuiga Box will be hard to implement if we do not remove the abstraction.

The issue is best portrayed using a scenario: assume that our Kuiga Box setup consists of one master node and two worker nodes. We create a *test-run* and specify the workload to be 50 virtual instances. We distribute this equally among the two workers, such that each worker node will have a workload of 25 instances. We discover an issue here since the *test-run* abstraction used by the master node is in many ways identical to what each worker node uses as encapsulation. This fact forces the master node to create two separate test runs that each constitutes a workload of 25 instances. The problem is related to the issue of cohesion because it should not be a *test-run* that Kuiga Master sends to each worker node: It should be a workload; something that a test run contains.

Moreover, it is troublesome to expose errors to the user, when one of the virtual instances terminates before time, as either the whole test run is executing as planned or it is *dirty*.

A possible solution to this, is to design and implement Kuiga Box to manage its internal resources in a model that fits the developer's mental map of the system, and in extension have an extra layer between the internals and the user, in which the developer friendly model is translated to a user friendly model. For this, we could still use the *test-run* abstraction for the user, and implement the test run such that it has the reference to its virtual instances, and it is these instances which are transported and manipulated throughout the system.

6.3 Broad Delimitation

When we examine Kuiga Box in retrospect, we recognize that we have a broad requirements delimitation in Section 1.1.1, considering the various components it consists of.

This delimitation is, however, intentional. We, instead of pursuing individual fields in this area, adopt a broad project perspective, where the challenge lies in connecting all the necessary fields of work.

We categorize the primary fields as:

- Distributed Systems
- Virtualization
- Network and Communication

We know from experience that each these fields are comprehensive enough to dedicate an entire semester to e.g. another project may spend a semester concentrating on improving the virtualization capabilities of Kuiga Box.

We do, however, not necessarily mean that Kuiga Box should have been delimited differently in our case, as it works great as a proof-of-concept system. The current state of Kuiga Box represents the outcome of combining and integrating all three of the mentioned fields through proper analysis and design. We do, however, recognize a side-effect of this approach which is that some of the components in Kuiga Box are simplistic compared to the potential these fields have individually e.g. pursuing virtualization can lead to many different ways of virtualizing IoT devices and applications.

7 Conclusion

We consider the problem statement presented in Section 1.3:

> *How do we design and implement a system, capable of executing and hosting potentially thousands of virtual devices, each running a user provided application, for the purpose of testing communication between devices and a base station?*

This question formed the starting point of our analysis on the subject of virtualization as shown in Section 1.2 and 2.2. We perform the same analysis on the area of scalability and distributed computing in Section 1.2.4 and 2.3.

Along with the requirements in Section 2.4 we can stipulate some high-level goals for the system to be designed and built, based on the critical notes gained from the investigations:

1. It can virtualize applications
2. It can scale horizontally
3. It can direct and handle virtualized instance network traffic
4. It can perform well under high workloads

We determine whether these goals have been accomplished by reflecting on the status of Kuiga Box's implementation concerning the MoSCoW requirements in Section 2.4.2 as well as the tests in Chapter 5.

Can Kuiga Box virtualize applications? The current version of Kuiga Box can virtualize applications compiled with the RIOT OS targeting an x86 architecture given that the worker node's hardware supports this architecture natively. We consider this goal completed.

Can Kuiga Box scale horizontally? We can, by design, scale horizontally through delegation of workload to multiple worker nodes. Kuiga Box supports this scaling to some degree. However, the implementation is not complete in a way that guarantees multiple worker nodes will not cause incorrect behavior in any component of Kuiga Box. We will, for this reason, consider this requirement partially completed.

Can Kuiga Box direct and handle the network traffic? Considering the MoSCoW in Section 2.4.2, we see that the implementation of the network traffic handling can be divided into two phases: First the traffic goes from the virtual instances to the Kuiga Plexer and secondly the traffic from Kuiga Plexer to the SDR. We implement the first phase and the second phase is not yet implemented, primarily because the SDR is not available to us yet. We, therefore, regard this goal as being partially completed.

Is Kuiga Box's performance sufficient? We formulated a *must have* requirement from the MoSCoW in Table 2.2 of being able to manage at least 100 virtual instances. We can conclude from the results from Section 5 that Kuiga Box can virtualize the required 100 virtual instances with a reasonable performance. We for these reasons consider the performance sufficient.

In conclusion We have implemented core features; that lets us create, start and terminate virtual instances. The framework for communication between Kuiga Box nodes has been implemented using the actor model which allows for straight-forward horizontal scaling due to its extensible nature. We find that, although we did not fulfill all the requirements, in their entirety, we are very satisfied with our progress.

We will use the discussion Chapter 6 to reflect and discuss how we can improve the system.

8 Future Work

We use this chapter to present possible improvements to Kuiga Box. The improvements include changes to our web API, how we implement the Kuiga Plexer as well as the inclusion of failure handling in Kuiga Box.

8.1 Kuiga Master REST API

In Section 2.4.2 we find the latest MoSCoW for Kuiga Box and see mentions of an API, however in the early phases of development, the table included points of explicitly implementing a REST API. This REST API was intended to be the API with which the Kuiga Box user controls all the things that managing test runs entails.

We currently structure our front-end around a web API as the means of interacting with Kuiga Box. This API, however, is unstructured in the sense that it does not necessarily follow any architectural styles such as REST. We are using the current API for development purposes, such as debugging, but many of the same features are also intended to be supported by a future REST API.

REST, first described by Roy Thomas Fielding [39], defines an architecture which specifies how resources are defined and how they are addressed using HTTP. These resources specify the state which is manipulated by HTTP methods such as, GET, PUT, POST, and DELETE. The architecture is, however, also described as being stateless in the sense that the REST API does not retain its state outside of the defined resources. This fact implies that a correctly implemented REST API does not retain user sessions in between request. This fact indicates that synchronization of user sessions between multiple hosts implementing the API is not required. This property improves the horizontal scalability of the API since multiple hosts implementing the same API are not reliant on each other. The fact that the REST API is easier to scale makes it very attractive for our use-case as we prioritize the scalability of our solution.

To follow the REST architecture requires that we model our API around resources e.g. *testrun, node, log*. The *testrun* resource is intended to have HTTP methods that support creating, deleting and querying a test run status. The *node* resource will be used to check the status of, for instance, worker nodes and running tests. This status information should include information such as the amount of connected workers, or whether the resources are insufficient on a given node. The *log* resources are as the name suggest intended to retrieve logs from various test runs.

We define a subset of the possible HTTP methods to interact with these resources, as well as their parameters, in Table 8.1

URI	parameters	description
CREATE:/kuigabox.com/testrun	architecture, amount	creates test run
GET:/kuigabox.com/worker/1	testRunId	get resource status
GET:/kuigabox.com/log/1	testRunId	get testrun log

Table 8.1: *Kuiga Master REST methods*

8.2 Kuiga Plexer

The Kuiga Box system does not currently interact with the SDR system. We, however in the future, intend for the virtual instances to communicate with the SDR system by interfacing with User Equipment (UE) exposed by the SDR system. This connection is meant to facilitate the delivery of network traffic to and from the SDR system. Shown in Figure 8.1 is a visualization of the setup, note that we have removed the logging process which was part of the original design in Section 3.6.

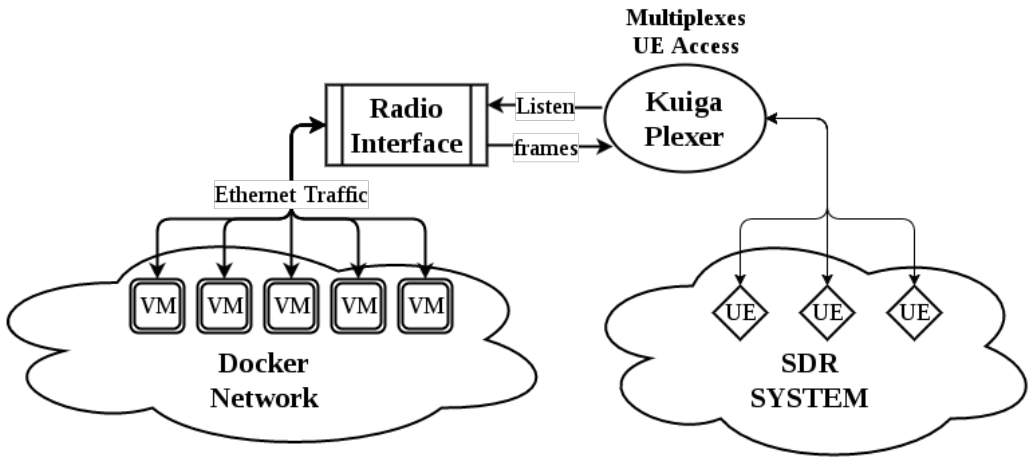

Figure 8.1: *Kuiga Plexer restricts and grants access to UE and the software radio SDR*

We can connect the VMs to the UE in two different ways, first it can match how it is configured in Figure 8.1 which also matches the initial design as seen in Section 3.1. This initial design is based upon the limitation of only having access to a limited amount of UE. This constraint requires that the Kuiga Plexer must restrict and grant UE access to the VMs, in other words, it multiplexes access to the UE. We found that this design also needs the Kuiga Plexer to manage a mapping between the address(IP) of a virtual instance and the UE on which it has sent traffic. This mapping between an internal address and an UE is required to facilitate communications in both ways between the UE and the VMs.

We learned later, through talks with group 16gr950, of another possible design solution in which the SDR exposes as many UE as there are virtual devices. In that case, the prior job of the Kuiga Plexer is no longer required as each virtual instance has access to its own UE. This sort of setup would instead require extensive routing configurations that instruct the host's network stack to forward network traffic between the VMs and their dedicated UE. In this case, the job of the Kuiga Plexer would be to create this network setup with tools such as *iptables*, a tool used to create network routing rules, instead of multiplexing access to a limited amount of UE.

8.3 Failure Handling

We use Section 2.4.1 to touch on the non-functional dependability requirement in which we highlight that resilience, or rather failure handling, is relevant to discuss with regards to a future versions of Kuiga Box.

We deal briefly with the failure handling subject in Section 3.8 where we describe the aspects of the communication between Kuiga Box's nodes. We realize that the decision of making the actor model handle the communication between nodes requires that the actor model also handles the failure handling between nodes. This choice is intentional and one of the key reason we chose to use the actor model in the first place. We will, therefore, require that Kuiga Box in future versions implements a failure handling scheme.

We use Akka, in Kuiga Box's current implementation of the actor model, which we describe in Section 4.1. Akka operates, by default, on a "let-it-crash" approach which we in some sense can perceive as a non-defensive view on program errors that encourages a developer to expect that some parts of the system will eventually crash without being able to prevent it. As we have previously discussed, this suits a distributed system well. For instance, we can have one actor (placed on one node) that needs information from another actor (placed on another node), where the first actor is not able to contact the other node for unknown reasons. Then, in this situation, it is impossible for the first actor to know what have happened.

The let-it-crash approach encourages that if such a situation, should occur, that the program does not stop executing or try to rescue what is broken. However, it requires that the broken part of the system should be replaced, such that the rest of the system remains operational.

There are several methods of implementing a let-it-crash approach in Kuiga Box, and we will not investigate which method is most suitable here. For a more in-depth look at the let-it-crash approach see the following page on Akka's website: `http://doc.akka.io/docs/akka/1.3.1/java/fault-tolerance.html`

Bibliography

[1] ITU, “New itu standards define the internet of things and provide the blueprints for its development,” [Online]. Available: http://www.itu.int/ITU-T/newslog/New+ITU+Standards+Define+The+Internet+Of+Things+And+Provide+The+Blueprints+For+Its+Development.aspx (visited on 18/12/2016).

[2] Gartner. (Nov. 10, 2015). Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015, [Online]. Available: http://www.gartner.com/newsroom/id/3165317 (visited on 26/09/2016).

[3] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974, ISSN: 0001-0782. DOI: 10.1145/361011.361073. [Online]. Available: http://doi.acm.org/10.1145/361011.361073.

[4] R. Dittner and D. Rule, *The Best Damn Server Virtualization Book Period: Including Vmware, Xen, and Microsoft Virtual Server*. Syngress Publishing, 2007.

[5] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011, ISBN: 9780470887998.

[6] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, “Xtratum: A hypervisor for safety critical embedded systems,” in *11th Real-Time Linux Workshop*, Citeseer, 2009, pp. 263–272.

[7] C. Henley, *Hypervisors*. [Online]. Available: https://blogs.technet.microsoft.com/chenley/2011/02/09/hypervisors/ (visited on 17/12/2016).

[8] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, “A comparison of virtualization technologies for hpc,” in *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, IEEE, 2008, pp. 861–868. [Online]. Available: https://www.researchgate.net/publication/221191387_A_Comparison_of_Virtualization_Technologies_for_HPC.

[9] D. Marshall, “Understanding full virtualization, paravirtualization, and hardware assist,” *VMWare White Paper*, 2007. [Online]. Available: http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf.

[10] IoT Now. (Oct. 6, 2016). Power saving tips for battery-powered iot devices, [Online]. Available: www.iot-now.com/2016/10/06/53360-power-saving-tips-for-battery-powered-iot-devices (visited on 19/12/2016).

[11] E. Brown. (Sep. 26, 2016). Open source operating systems for iot, [Online]. Available: https://www.linux.com/news/open-source-operating-systems-iot (visited on 07/12/2016).

[12] FIT, *Operating systems*, FIT. [Online]. Available: https://www.iot-lab.info/operating-systems/ (visited on 03/11/2016).

[13] *Freertos.* [Online]. Available: http://www.freertos.org/ (visited on 03/11/2016).

[14] *Riot os: The friendly operating system for the internet of things.* [Online]. Available: http://riot-os.org/ (visited on 03/11/2016).

[15] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. Schmidt, "Riot os: towards an os for the internet of things," in *The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy, Apr. 2013. [Online]. Available: https://hal.inria.fr/hal-00945122.

[16] *Contiki: The open source os for the internet of things.* [Online]. Available: http://contiki-os.org/index.html (visited on 03/11/2016).

[17] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[18] Xen community. (2016). Xen project software overview, [Online]. Available: https://wiki.xen.org/wiki/Xen_Project_Software_Overview (visited on 04/11/2016).

[19] J. Turnbull, *The Docker Book: Containerization is the new virtualization.* James Turnbull, 2014, ISBN: 9780988820203. [Online]. Available: https://books.google.dk/books?id=4xQKBAAAQBAJ.

[20] *Docker overview*, Docker Inc. [Online]. Available: https://docs.docker.com/engine/understanding-docker/ (visited on 09/12/2016).

[21] *What is docker*, Docker Inc. [Online]. Available: https://www.docker.com/what-docker (visited on 03/11/2016).

[22] *Libvirt: The virtualization api.* [Online]. Available: https://libvirt.org/ (visited on 08/12/2016).

[23] "Virtual infrastructure managers used in production," English, in *Scheduling of Large-scale Virtualized Infrastructures*, F. Quesnel, Ed. 2014, ISBN: 978-1-84821-620-4. DOI: 10.1002/9781118790335.ch3. [Online]. Available: https://ebookcentral.proquest.com/lib/aalborguniv-ebooks/reader.action?ppg=47&docID=1765087&tm=1482072977898.

[24] *Open stack - project navigation*, 2016. [Online]. Available: https://www.openstack.org/software/project-navigator/ (visited on 12/12/2016).

[25] I. Sommerville, *Software engineering*, 10th ed. Boston, Mass. u.a: Pearson, Apr. 2015, ISBN: 1292096136. [Online]. Available: http://iansommerville.com/software-engineering-book/.

[26] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: Concepts and Design.* pearson education, 2005.

[27] Agile Business Consortium. (2008). Moscow prioritisation, [Online]. Available: https://www.agilebusiness.org/content/moscow-prioritisation-0 (visited on 17/12/2016).

[28] N. Briscoe, "Understanding the osi 7-layer model," Jul. 2000. [Online]. Available: https://www.os3.nl/_media/2014-2015/info/5_osi_model.pdf (visited on 17/12/2016).

[29] C. Hewitt, "Actor model of computation: Scalable robust information systems," *ArXiv preprint arXiv:1008.1459*, 2010.

[30] A. Stannard, *Akka.net: What is an actor?* Jan. 25, 2015. [Online]. Available: `https://petabridge.com/blog/akkadotnet-what-is-an-actor/` (visited on 19/12/2016).
[31] B. Thomsen, *The actor programming paradigm (the concurrent programming paradigm)*, University Lecture, 2016.
[32] *Google guice.* [Online]. Available: `https://github.com/google/guice` (visited on 19/12/2016).
[33] The Linux Documentation Project. (2016). What is a bridge? [Online]. Available: `http://www.tldp.org/HOWTO/BRIDGE-STP-HOWTO/what-is-a-bridge.html` (visited on 21/12/2016).
[34] *Google leveldb.* [Online]. Available: `https://github.com/google/leveldb` (visited on 19/12/2016).
[35] *Ceph.* [Online]. Available: `http://ceph.com/ceph-storage/object-storage/` (visited on 19/12/2016).
[36] *Seaweedfs.* [Online]. Available: `https://github.com/chrislusf/seaweedfs` (visited on 19/12/2016).
[37] *Aws sdks.* [Online]. Available: `https://aws.amazon.com/tools/#sdk` (visited on 19/12/2016).
[38] *Scalatest.* [Online]. Available: `http://www.scalatest.org/` (visited on 19/12/2016).
[39] Roy Thomas Fielding. (2000). Representational state transfer (rest), [Online]. Available: `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm` (visited on 20/12/2016).

Glossary

ACL Access Control List. 43

Amazon S3 Amazon Simple Storage Service. 40, 41, 43

AUT Application Under Test. 2, 16, 22, 23, 26–29, 43, 44, 51–53, 55

AWS Amazon Web Services. 43

DI Dependency Injection. 34, 35, 37, 48

GUI Graphical User Interface. 25, 26

industry contact Refers to our industry contact: Andrea Cattoni. 1–3, 7, 20

IoC Inversion of Control. 35–37

IoT Internet of Things. 1–3, 7, 9, 10, 14, 16, 19, 21, 23, 28, 29, 51, 58

Kuiga Master The master component of Kuiga Box. 41

Kuiga Worker The worker component of Kuiga Box. 41, 43

KVM Kernel Virtual Machine. 11

NAT Network Address Translation. 39

OO Object-Oriented. 34

OpenStack Swift A distributed object store from the OpenStack ecosystem. 41

OS Operating System. 2–14, 17–19, 22, 23, 43, 59

OSI Open Systems Interconnection. 25, 39

REST Representational State Transfer. 41, 43, 61, 62

RF Radio Frequency. 2, 3, 21, 23

RHEL Red Hat Enterprise Linux. 41

RPC Remote Procedure Call. 26

ScalaTest Multi-style unit testing framework for the Scala language. 48, 49

SDR Software-Defined Radio. 3, 17–24, 39, 59, 60, 62, 63

SeaweedFS A distributed key-blob storage technology. 42

UE User Equipment. 24, 25, 62, 63

vEth virtual Ethernet. 38

VM Virtual Machine. 3, 5–7, 11, 12, 62, 63