NTA

- Network Test Application -

Project Report

Group d807f17

Aalborg University
Department of Computer Science
Embedded Software Systems

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
NTA - Network Test Application

Theme:
Embedded Software Platforms

Project Period:
Spring Semester 2017

Project Group:
d807f17

Participants:
Anders Normann Poulsen
Gabriel Vasluianu

Supervisor:
Brian Nielsen

Copies: 1

Page Numbers: 58

Date of Completion:
1st June 2017

Abstract:

The IoT paradigm keeps evolving, leading to the development and growth of devices providing different functionalities. These devices are connected to the internet through different protocols. Devices placed in remote areas will have to use mobile networks to get connectivity. A typical setup would be a couple hundred of these devices, communicating with the same base station.
The problem is that the LTE standard has not been developed for this application, nor properly tested in such conditions. This project tries to approach this problem by focusing on a test application running on a virtualised test bed. The outcome would be a machine, able to virtualise a number of IoT devices, and replicate their communication. This could then be used to test the capabilities of the communication layer in an IoT system.

Preface

This report is written by the d807f17 group following the Embedded Software Systems master programme at Aalborg University. The subject of the report is the 8th semester project: NTA - Network Test Application. This resulted in developing a system that fulfils the study regulations.

The group is thankful to Brian Nielsen for guidance and advice throughout the project. Also, the group is grateful to Andrea Cattoni for the discussions and suggestions related to the project. The collaboration with the other students has helped throughout the development of NTA: sw704e16 (Software Engineering), 17gr1050 (Wireless Communication Systems) and Georgi Andonov (Computer Science).

Aalborg University, 1st June 2017

Anders Normann Poulsen
<apouls16@student.aau.dk>

Gabriel Vasluianu
gvaslu16@student.aau.dk>

Reading Guide

In this report, references are written using the Harvard method [last name(author or company),year(if available)]. Sources are listed alphabetically in the bibliography. Figures, Tables, Equations, and Listings (code snippets) are numbered after the chapter they are in, for example, the first figure in chapter two is called 2.1, the next 2.2 etc. All illustrations and figures are made by the group members unless otherwise stated. All data sheets and code can be found on the zip archive electronically uploaded with the report.
A prerequisite for reading this report is basic knowledge in the computer science field.

Glossary

CPU	Central Processing Unit
DMA	Direct memory access
DSI	Display Serial Interface
FPGA	Field-programmable gate array
GPIO	General-purpose input/output
GPU	Graphics Processing Unit
HDMI	High-Definition Multimedia Interface
IoT	Internet of Things
JVM	Java Virtual Machine
LTE	Long-Term Evolution (communication standard)
MTC	Machine Type Communication
PC	Personal Computer
RISC	Reduced instruction set computing
RAM	Random-access memory
SBC	Single Board Computer
sbt	Scala Build Tool
SDN	Software Defined Network
SDR	Software Defined Radio
SoC	System on a Chip
SUT	System under test
USB	Universal Serial Bus

Contents

CHAPTER 1

Introduction

Today's world is more interconnected the ever before, due to the progress of Internet technologies and rise of cheap, but powerful computing platforms. While the milestone of having one connected device for each person in the world has long been passed, we are looking at having at least 3-4 times as many devices as the world's population in the next decade[11].

This growth is driven by the IoT paradigm, which has gained traction, and got the attention of numerous companies operating in different fields. As expected, research is currently being conducted, focusing on security, energy capacity of the batteries operating the IoT devices and balancing the processing power required by these[29].

A recent study by Ericsson shows that the Danish market is just started to adopt the IoT, with 60% of the companies they interviewed having IoT initiatives ongoing[14]. This figure will increase with time, leading to more concrete projects deployed on the market. The implications for telecommunication companies are large, meaning they will have to further develop, expand and test their services.

Figure 1.1 shows the simplified architecture of an IoT system, with its different layers (from right to left): physical, communication, system, application. This project focuses on the communication layer. The traditional way of connecting the physical layer (the IoT devices) to the cloud is by having a wireless internet network. However, tech companies have interest into adopting the LTE standard for their networks due to its versatility.

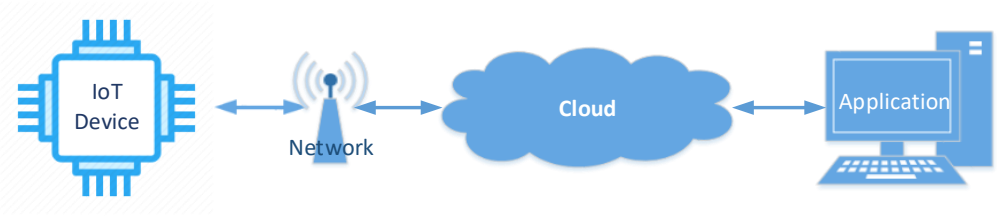

Figure 1.1: IoT architecture

LTE (Long-Term Evolution) is a wireless communication standard having *full mobility, high data rates and coverage, predictable latency, quality of service and ease of deployment*[4] as main advantages.

In order to make a clean transition to LTE, there are several things to consider. One such aspect is the security of LTE enabled IoT systems. Another concern is how well the base station will perform in an IoT context. Traditionally, LTE devices send and receive large datasets in short bursts (e.g. a smartphone loading a website), whereas IoT devices may tend towards sending and receive small datasets, but as continuous streams (when compared to smartphones). Our industry contact[1] shared the interest in researching and testing the capabilities of cellular base stations.

A previous project[3] has approached the task of creating a framework for testing the communication layer in an IoT setup. The main idea was to see how many devices a wireless cellular network can reliably serve. This can be achieved by virtualising the IoT devices on a machine, that would then transmit data on the network as if there were several isolated devices. By doing this, it is possible to find out what are the limitations of the communication layer. More information on this topic can be found in section 2.2 Description of Kuiga Box.

Keysight has further expressed their desire of having a testbed running on a compact yet powerful platform. This means that it should be possible to perform intensive tests at different locations. The aim of this project is to provide a tool that can stress test existing cellular networks, in particular LTE-networks.

[1] Andrea Cattoni - researcher at Keysight Technologies

CHAPTER 2

Analysis

This chapter presents the prerequisites of the project, as they have been discussed with the industry contact. Previous work on an IoT testbed is analysed, in order to see what can be learned from their endeavours. Based on this, the criteria for choosing a platform to work on are established, and different hardware are investigated. In the end, a choice will be made for one of the development platforms and the work plan will be established. The overall goals of the project will be discussed in the Problem statement.

2.1 Keysight and their expectations

Keysight[27] is a US company that manufactures test and measuring equipment for electronics and radio devices. They have an R&D department in Aalborg which has a close collaboration with the School of Information and Communication Technology (SICT) at Aalborg University. In the discussions the group had with Andrea Cattoni, he expressed the company's interest in developing a test platform for the IoT, particularly for doing experiments on the communication layer. He mentioned that the testbed should:

- emulate a large number of IoT devices. A realistic test setup would comprise of around 1000 devices
- be capable of being controlled. This means that the data that is being sent on the network will be analysed, and based on this, its characteristics can changed, in order to measure the preciseness of the system
- be portable. It should be possible to carry around the device to different test locations. A good approximation of the actual size would be a big suitcase.

Furthermore this aspect implies that the energy requirements of the system are small enough, so it can be battery operated

- require as few resources as possible. The algorithms for the test application should be optimised, so that the testbed can be run on relatively modest hardware

Based on these requirements, figure 2.1 presents a plausible scenario for the testbed. The IoT devices would be sending data to the SUT (System Under Test). In this case, the system is represented by a base station of an LTE network. The SUT's performances will be monitored, and by analysing the results, the IoT devices can be controlled throughout the course of the test.

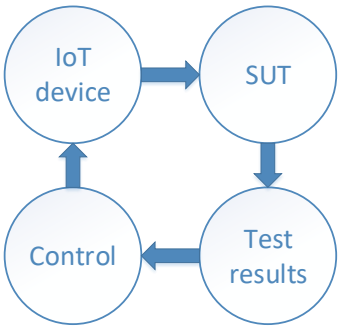

Figure 2.1: Test scenario

The challenge Keysight proposes is to investigate these requirements and assess how, and if they can be accomplished in a practical setup. Due to the fact that the system has to be small, portable and use low amounts of energy, the foundation for the testbed will be an embedded platform. Besides this, using an embedded platform is a precondition for this semester project, due to the study regulation.

2.2 Description of Kuiga Box

Kuiga Box is the product of a semester project for a group of 7th semester software engineering students (SW7). It attempts to provide a framework for virtualisation of IoT devices, to solve the problem proposed by Keysight (see section 2.1). To avoid reinventing the wheel, Kuiga Box makes no attempt to implement a new form of virtualisation, but instead uses Docker[12] as the engine for virtualisation. Figure 2.2 shows an overview of the software stack used for virtualisation. Of the problem Keysight proposed, Kuiga Box attempts to solve only a part of it. Obviously, to test an LTE network one needs to connect to it, and to do that, one needs an LTE radio and antenna. Traditionally in LTE networks, each device has its own

LTE radio and antenna. This, however would require a lot of LTE hardware for a virtual testbed. To alleviate this problem, a separate master thesis project is running concurrently to implement a software define radio (SDR) to allow virtual IoT devices to share the radio and antenna. To see more about this project, see section 2.5.

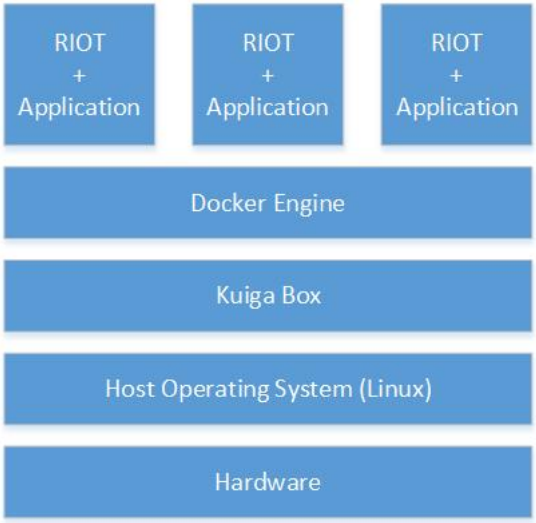

Figure 2.2: A diagram of the Kuiga Box stack

To help improve the scalability of Kuiga Box, and possibly also to fulfil SW7 study regulations, Kuiga Box has been implemented as a distributable platform. This means that several physical nodes can join together in testing the network. To accomplish this, Kuiga Box is split into four major components:

1. **Kuiga Master** - A central controller for Kuiga Workers. It is responsible for building the Docker images to be used for testing, and distributing work and controlling the individual, connected worker nodes.

2. **Kuiga Worker** - A local front-end for Docker on the individual nodes of the testbed cluster. They are responsible for directly interfacing with Docker, as well as setting up the networks to be used for the test.

3. **Kuiga Web** - A simple front-end for Kuiga Master.

4. **Image Store** - A storage area for Docker images. The image store is not an executable component, but instead a wrapper around an arbitrary file system.

The following figure represents the architecture of Kuiga Box and its components.

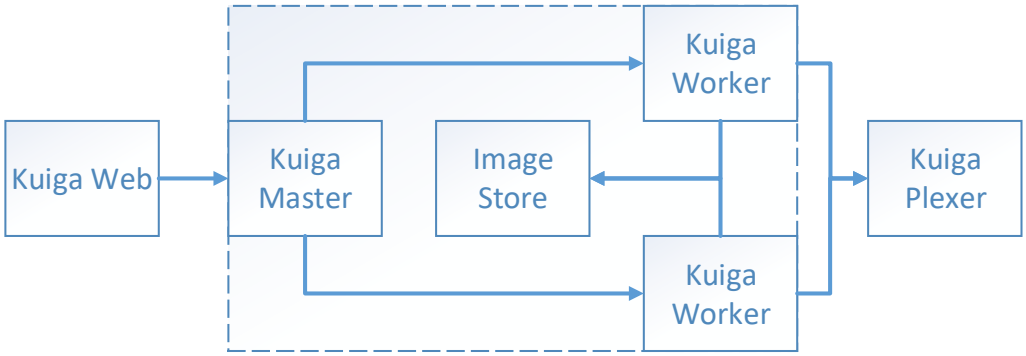

Figure 2.3: Kuiga Box overview

Figure 2.3 shows, in addition to the four components described above, Kuiga Plexer. This component was theorised by the SW7 group, but never implemented. It is intended to connect Kuiga Box with the SDR. This component is also sometimes referred to as the middleware. To see more about this see section 2.5.3.

Though Kuiga Master can be launched alone, it is also launched alongside Kuiga Web, and as such, in a running system only Kuiga Web and Kuiga Worker should be launched.

During the development of Kuiga Box, making it run on embedded hardware was not a priority. Therefore, the system has not been optimised for low resource environments.

2.3 Analysing resource consumption

With a focus on running Kuiga Box on embedded hardware, resource consumption is an obvious thing to investigate. As memory, processing power and storage are limited on embedded hardware, it is important to analyse how much of these Kuiga Box requires, so as to better understand what the minimum requirements for hardware are.

2.3.1 Processing

Embedded hardware is often categorised by having limited processing power. Although certain hardware, such as the Raspberry Pi 3, has become more powerful, it is still not as powerful as its PC siblings, for which Kuiga Box was originally developed.

As Kuiga Box itself is just a control interface for the virtual IoT devices used in the testbed, it has very little work to do during test runs. Therefore, when the system is a at rest, or actively testing, the processing power it needs is minimal, to the point where it has almost no impact on overall performance; it is only during the start up and shut down phase that Kuiga Box itself requires significant processing resources. This does not take into consideration the CPU time that the Docker images will use.

2.3.2 Storage

The Kuiga Box source files, and their compiled counterparts, including dependencies, require only about 100 MB of space. In addition to this, there is a 350 MB Docker image build for the tests. As Kuiga Box is developed to be distributed, both Kuiga Master and Kuiga Worker store a copy of all Docker images, regardless if they are running on the same system or not. In addition, while Kuiga Worker is loading Docker containers, the image is also stored as a temporary file. Therefore, during runtime, up to three copies of the Docker image will exist. This means that the minimum space required for the Docker images is in reality $350\text{MB} \cdot 3 = 1050$ MB. For this reason Kuiga Box and Docker must have as a minimum 1.5 GB of space to work with.

2.3.3 RAM

Kuiga Box runs on top of a Java Virtual Machine (JVM). Popular JVM implementations such as the Java HotSpot JVM[21] by Oracle and the OpenJDK JVM[22], are not implemented in a way to minimise memory usage. Instead, anticipating further object creation, they allocate memory ahead of time. This allows code to create simple objects without having to wait for the JVM to allocate more memory every time. This in turn improves the responsiveness and speed of the code. Unfortunately, it also leads to a larger memory overhead, which is naturally undesirable, especially on memory constrained systems.
Adding to the memory overhead from the JVM itself, is the choice of programming language for Kuiga Box. Kuiga Box is primarily programmed in Scala. Why the previous group chose this language is not documented, but we speculate, that it might have been because of Scala's ability to deliver fast prototypes, the ease in implementing an Actor-based application, or simply an academic curiosity, owing to it being a functional programming language with strong emphasis on object-oriented code. Regardless of the reason, choosing Scala over the more traditional Java, may too, contribute to a higher memory consumption, owing to the way functional programs will favour immutable objects over mutable objects, thus creating more objects during runtime. This may not be immediately visible when the system is at rest, but when the system is working, with new objects being cre-

ated more frequently, the peak memory consumption may rise, awaiting the next garbage collection.

Considering the choice of Scala, and therefore the use of a JVM, it is clear that the memory consumption of Kuiga Box may be an issue, and therefore needs some investigation. To start off with, the memory consumption with unmodified (relative to the work of the previous group) JVM parameters. These JVM parameters are `-Xms512m -Xmx1024m -Xss2m`, which set the initial size of the Java heap to 512 MB, its maximum size to 1024 MB and the thread stack size to 2 MB. Aside from this are the default JVM options to use the ParallelGC garbage collector, and to compress pointers.

		Kuiga Web			Kuiga Worker		
Memory		Allocated	Resting	Peak	Allocated	Resting	Peak
First run	Total	-	920 MB	920 MB	-	776 MB	776 MB
	Heap	1024 MB	243 MB	421 MB	872 MB	114 MB	375 MB
Default	Total	-	711 MB	711 MB	-	617 MB	617 MB
	Heap	678 MB	181 MB	340 MB	952 MB	243 MB	273 MB

Table 2.1: Memory consumption of Kuiga Box. First run refers to the first time Kuiga Box is launched. These measurements are taken in the same way described in section 2.3.3.1

Legend:

Allocated	Maximum allocated heap memory.
Resting	The memory used when the software is at rest (after setting up virtual IoT devices).
Peak	The maximum memory used by the software during one test run.

As can be seen in table 2.1, without any kind of tuning of the JVM, Kuiga Box uses a lot of combined memory: 1328 MB. This is clearly a lot, especially considering that it does not include the actual IoT virtualisation, but is only the overhead added by Kuiga Box. As can be seen, the first run uses more memory, because it must also compile Kuiga Box. The numbers for the first run are not important for regular operation, as this must be done only once, although they set a strict minimum, as there *must* be enough memory to complete the compilation process.
Examining the numbers for the second run with the default configurations some important points are revealed:

1. Most of the heap space allocated is never used.
2. Memory is never released back to the operating system.

Considering this, it becomes clear that there is plenty of room to tune the JVM with

the purpose of reducing the memory consumption.

To explore how to tune the JVM for minimal memory footprint, an experiment is set up.

2.3.3.1 Experiment

Objective
Determine which JVM tuning parameters help reduce the memory footprint of Kuiga Box, without interfering with the execution. As shown by experiments conducted by others[15], even changing the garbage collector can affect the memory consumption. Tests will also be conducted to see whether that is the case for Kuiga Box.

Setup
To make it easier to conduct the experiment, it will be run in a virtual machine set up on a PC. The virtual machine runs Fedora 25 Linux with Oracle Java HotSpot JVM version 1.8.131 installed. To avoid swapping, which will make it difficult to get accurate memory readouts, the virtual machine is fitted with 4 GB of memory; more than the embedded hardware this project will use. As this is a test of memory, the virtual machine is given a comfortable 4 CPU cores to work with. Raw memory usage will be read from Gnome System Monitor, which accurately shows how much RAM the processes are using. Heap memory usage will be read from VisualVM[31], which can show how heap usage evolves over time.
The tuning parameters that will be explored are:

- Reducing the heap size (initial and maximum)
- Reducing the maximum free heap ratio.
- Changing the garbage collector
- Letting the JVM unload classes

Both Kuiga Web and Kuiga Worker are being launched via `sbt` - Scala Build Tool.

Results
The results from the experiment are shown in table 2.2.

		Kuiga Web			Kuiga Worker		
Memory		Allocated	Resting	Peak	Allocated	Resting	Peak
Default	Total	-	711 MB	711 MB	-	617 MB	617 MB
	Heap	678 MB	181 MB	340 MB	952 MB	243 MB	273 MB
-Xms1m	Total	-	544 MB	544 MB	-	435 MB	435 MB
	Heap	311 MB	123 MB	191 MB	223 MB	89 MB	172 MB
-Xmx512m	Total	-	536 MB	536 MB	-	470 MB	470 MB
	Heap	512 MB	196 MB	196 MB	512 MB	124 MB	167 MB
-Xms256m	Total	-	476 MB	476 MB	-	378 MB	378 MB
-Xmx256m	Heap	256 MB	164 MB	169 MB	256 MB	109 MB	119 MB
-Xms128m	Total	-	396 MB	396 MB	-	358 MB	358 MB
-Xmx128m	Heap	128 MB	94 MB	112 MB	128 MB	74 MB	95 MB
-Xms64m	Total	-	X	X	-	X	X
-Xmx64m	Heap	X	X	X	X	X	X
-Xms1m	Total	-	X	X	-	X	X
-Xmx128m	Heap	X	X	X	X	X	X
-Xms1m -Min/ MaxFreeHeapRatio=5/10	Total	-	430 MB	430 MB	-	330 MB	330 MB
	Heap	179 MB	141 MB	150 MB	104 MB	71 MB	82 MB
-Xms1m -Xmx128m -Min/MaxFree HeapRatio=5/10	Total	-	X	X	-	330 MB	330 MB
	Heap	X	X	X	104 MB	71 MB	82 MB
-Xms1m -Xmx128m -Min/MaxFree HeapRatio=0/10	Total	-	X	X	-	317 MB	320 MB
	Heap	X	X	X	102 MB	74 MB	78 MB
-XX:+Use SerialGC	Total	-	565 MB	565 MB	-	477 MB	477 MB
	Heap	512 MB	242 MB	280 MB	512 MB	110 MB	210 MB
-XX:+UseSerialGC -Xms1m	Total	-	456 MB	456 MB	-	346 MB	346 MB
	Heap	270 MB	132 MB	193 MB	174 MB	85 MB	116 MB
-XX:+UseSerialGC -Xms1m -Xmx128m	Total	-	393 MB	393 MB	-	322 MB	322 MB
	Heap	128 MB	92 MB	124 MB	128 MB	95 MB	103 MB
-Xms1m -Xmx128m -Xss2m -XX:+CMSClassUnloading Enabled -XX:+Use ConcMarkSweepGC	Total	-	X	X	-	X	X
	Heap	X	X	X	X	X	X

Table 2.2: Memory consumption of Kuiga Box, shown both as overall memory used, and heap used.

Legend:

Allocated	Maximum allocated heap memory.
Resting	The memory used when the software is at rest (after setting up virtual IoT devices).
Peak	The maximum memory used by the software during one test run.
X	Crashed or too slow to run

It is observed during the experiment, that the JVM does very little garbage collection if the heap is not near the limit of the space allowed. Therefore, not limiting the size of the heap, results in the memory usage of Kuiga Box rise drastically. This means, that without some aggressive limits to the heap size, Kuiga Box will occupy more memory than necessary.

The results of the experiment also show, that a higher initial heap size ultimately increase the memory usage of Kuiga Box. While no obvious reason for this has been found, it can possibly be because the operating system is preallocating physical memory to accommodate the entire heap, as opposed to only the used heap, to save time when the application later allocates memory.
The results also show, that reducing the initial heap is not without consequences; reducing the initial heap without changing the maximum heap, can lead to out of memory errors, as well as `sbt` endlessly trying to rediscover dependencies. This can be seen by the fact that Kuiga Box can run with the settings `-Xms128m -Xmx128m`, yet not with `-Xms1m -Xms128m`.

Lastly, it can also be seen, that changing the garbage collector, from the default ParallelGC to SerialGC, has a positive impact on the overall memory consumption of Kuiga Box. This may be because the serial garbage collector is simpler than the parallel, and hence requires fewer resources itself. It may also be because SerialGC is more optimised for applications that require less memory[16], and it may therefore have an advantage in garbage collecting low-memory applications.
The ConcMarkSweep garbage collector, which has the ability to unload unused classes, was also tested. In principle, it should help to reduce the memory footprint, as the JVM loads many classes from the dependencies of Kuiga Box during initialisation which may not be needed during runtime. In practice however, as the JVM tries to avoid garbage collecting if it is not near the maximum heap size, it has very little effect. This owes to that fact that, one of the peaks in memory usage happens during initialisation. Thus, reducing the maximum heap size in an attempt to trigger garbage collection during runtime, also hinders application initialisation from completing.

Implications
Through this experiment, it is now possible to determine how to optimise the JVM parameters, such that the memory footprint of Kuiga Box is as small as possible without negatively affecting performance in a significant way. It can be seen from the results, that Kuiga Box uses the least resources when it is run with the JVM parameters `-Xms1m -Xmx128m -Xss2m -XX:+UseSerialGC`, and it will therefore be these parameters that will be used for Kuiga Box during this project.
From the results, it is also possible to determine the minimum required memory to run Kuiga Box: $393 + 322 = 715$ MB. Compared to the original memory consumption of 711 MB for Kuiga Web and 617 MB for Kuiga Worker, this reduces the memory consumption with 45% and 48% respectively, or a total of 46%.

What the raw data from this experiment does not show, but what analysis of, and experience with, Kuiga Box have shown, is that once Kuiga Box is done setting up the virtual IoT devices, and while it waits to shut them down, it has nothing to do. Hence, Kuiga Box is not needed in the majority of the runtime of the tests conducted. This means that after the virtual IoT devices are set up by Kuiga Box, it will not harm the execution of the test if Kuiga Box is placed in swap by the underlying operating system. This largely negates the overhead introduced by Kuiga Box, and makes it possible to dedicate more RAM to the virtual IoT devices.

2.4 Hardware

In order to choose an embedded platform, the group took a look at the newest development platforms available on the market. Based on the expectations from Keysight, and on the initial analysis of the Kuiga Box tool, the following list of criteria has been formulated:

- Performance. The board should have a 64-bit CPU in order to run Docker, since it supports only the `x86_64` and `armhf` architectures[13]. It should also have at least 1 GB of RAM.
- Development support. Popular and well documented boards.
- Should be able to run a Linux distribution; this is a requirement of Docker.
- Availability. Mature projects, with hardware distributed in Europe, preferably Denmark.
- Price. Low enough so that the department[1] can afford it.
- Size. Small enough to make the testbed portable.
- Power consumption. Low enough to allow the testbed to be powered by batteries.

The factors that weight the most at this point in time are: performance, development support and support for Linux. Availability is also important, but it should be possible to source any of the boards from Europe, leading to a short shipping time.

[1] Aalborg University - Department of Computer Science

2.4.1 Initial hardware options

The following list consists of the boards initially researched to be used as the main development platform. The table in Appendix A contains details about each board, based on the established criteria.

1. **Parallella Board** is a board that aims at *democratizing access to parallel computing by offering affordable open source hardware platforms and development tools*[23].
2. **Raspberry Pi 3** is one of the most popular boards in this list. The great adoption of Raspberry Pi, together with the more powerful hardware of newer iterations, has lead several major software solutions to official support Raspberry Pi, such as Docker[26] and GitLab[32].
3. **Odroid C2** is advertised as a board having more advantages and better performance than the Raspberry Pi[30]. However, it is less popular and less supported, despite the fact that it is cheaper than Raspberry Pi.
4. **LattePanda** is a board that differences from the others through the fact that it can run full Windows 10. It also features a coprocessor suitable for less compute-intensive tasks[18].
5. **Pine A64** is the cheapest board in this list, the reason being that in can only be sourced from China[24].
6. **Intel Compute Stick** is *a device the size of a pack of gum, that can transform any HDMI display into a complete computer*[6]. This is by far one of the most innovative products from the list, designed for ease of usage.
7. **Other options:** XPedite6401[8], Tessel 2[28], Arduino Yún[2]. These boards were investigated due to their popularity in the hobbyist niche. However, they do not provide the performance required, and for this reason will not be included in the table in Appendix A.

As seen in this section, there are a variety of boards to choose from, for an embedded application. The hardware producers have shown their creativity and expertise in developing solutions for all kind of applications, with regards to power consumption and cost effectiveness.

Taking into consideration the requirements for the project (in section 2.1), the group decided to further investigate the Parallella board and the Raspberry Pi 3. This way, it would be possible to see how they work in detail, if there has been conducted any objective research of their performances, and then take a final decision on which board to use.

2.4.2 Parallella

Parallella is build on a many-core architecture showcasing parallel computing with the use of its 16 core Epiphany coprocessor. Parallella has a dual-core ARM Cortex-A9 main processor, which uses the ARMv7-A architecture (32-bit). The great advantage is that there are a lot of distributions built for this architecture.

The ARM processor is coupled with an FPGA, and the coprocessor, as seen in figure 2.4.

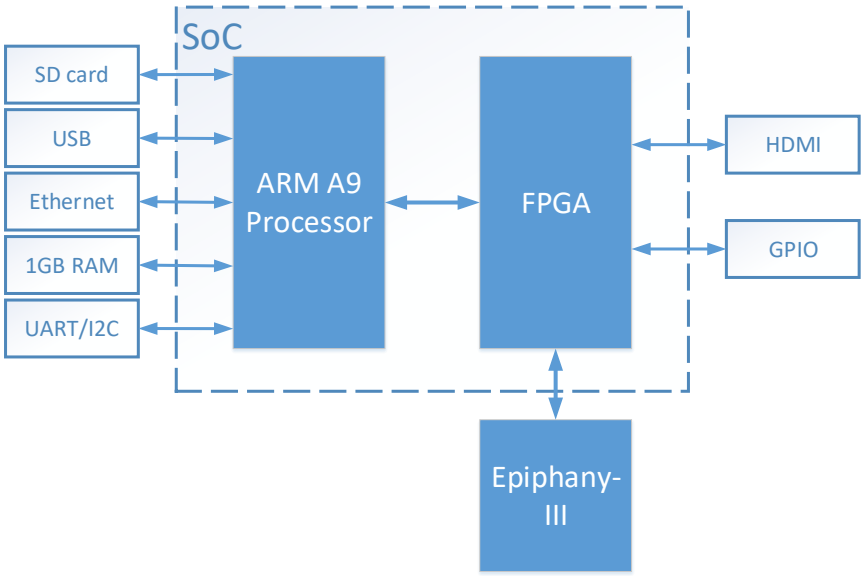

Figure 2.4: Parallella Block Diagram [9]

The coprocessor is an Epiphany-III[1] and has 16 cores. This is achieved by having a mesh network, containing 16 nodes as seen in figure 2.5. Each node has a RISC core (Reduced Instruction Set Computer), a DMA (Direct Memory Access) engine, local memory and a network interface. The cores are called "eCores", and due to the instruction set they use, languages such as C and C++ can be used effectively to program applications for them. The toolchain for developing for the eCores is open-sourced and licensed as free software[5]. According to the manufacturer[2] the cores run at 1 GHz. However, Paralella states that based on their measurements, the cores only achieve an operating frequency of 600 MHz and may require active cooling.

[2]Adapteva, Inc.

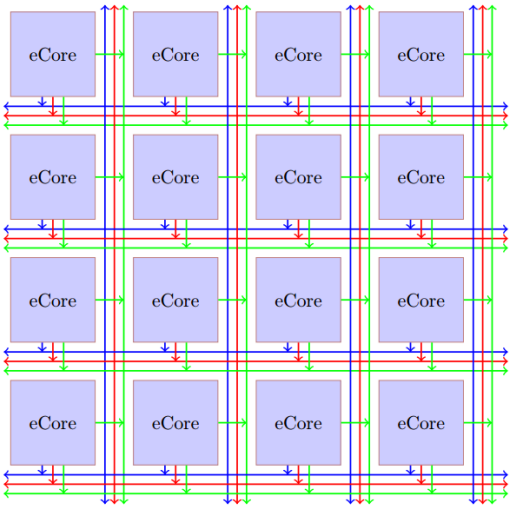

Figure 2.5: The mesh network containing 16 eCores. The coloured lines represents read/write communication lines[9]

The purpose of the FPGA (Field-Programmable Gate Array) is to provide an interface between the ARM processor and the Epiphany-III coprocesor. Besides this, it also provides HDMI (High Definition Multimedia Interface) and a number of GPIO (General Purpose Input/Output) pins.

There are typically two approaches, when programming the board:

- use the ARM processor to run the main program, and let the eCores do some of the heavier computations
- use the eCores to run an optimised version of the program to be executed, having the ARM CPU controlling the process

The issue to consider here is the cost of memory transfers between the ARM processor and the eCores. This problem can further be examined, in order to choose the most efficient solution (use of local or external memory, use of DMA, etc.). However, generally speaking, by going with the first approach, an algorithm would be easy to implement, having certain functions run on the coprocessor cores. This might be executing slow, due to the memory transfer bottleneck. In the second situation, the algorithm would execute fast, but it would have to be developed for the coprocessor, to take advantage of the 16 cores. This can become a time-consuming and rather difficult task due to the small data the eCores can work with[9].

2.4.3 Raspberry Pi 3

Raspberry Pi is arguably the most popular SBC (Single Board Computer), and can be regarded as the project that started the whole SBC trend. It is built around the Broadcom BCM2837 SoC (System on a Chip), which comprises of:

- a quad-core ARM Cortex A53. This processor supports both 32-bit and 64-bit (ARMv8-A architecture) instruction sets. This architecture is officially supported by Debian, who have constantly released distributions for the Raspberry Pi, including their latest: Raspbian Jessie[10].
- a VideoCoreIV GPU (Graphics Processing Unit)
- 1 GB of RAM

Figure 2.6 represents the board's block diagram.

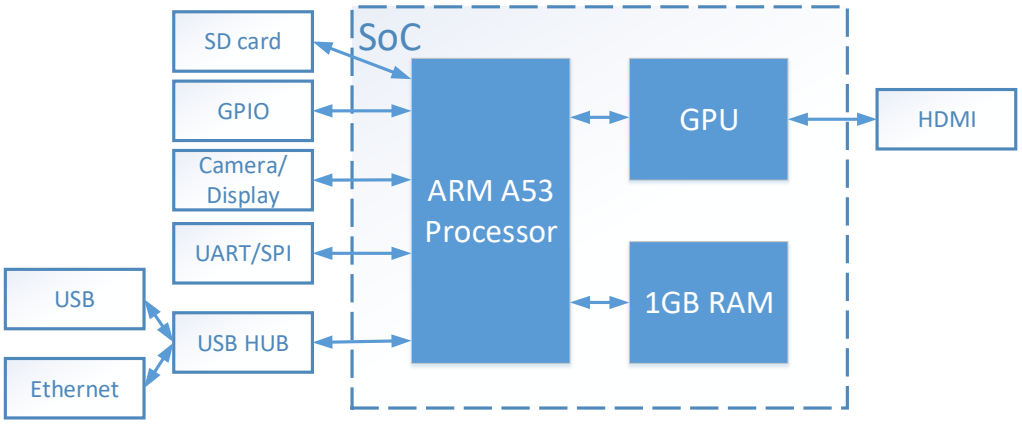

Figure 2.6: Raspberry Pi 3 block diagram[7]

The Raspberry boards can be set up into computing clusters, in order to achieve performances comparable to high-end computers. One specific study[20] showed that a cluster made up of 32 Raspberry Pi 2[3] was more then three times faster at running a password cracking algorithm[4], than a high-end laptop computer[5]. Furthermore, the price of the Raspberry Pi setup was less than half the price of the computer, which proves the outstanding capabilities of SBCs.

[3]This is the previous version of the Raspberry Pi, not the one presented as a choice in this section. Anyway, the results of this experiments are still relevant for the final decision

[4]John the Ripper

[5]Macbook Pro from 2015, with an estimated price of 4000 USD

This scenario was replicated with the help of 8 Parallella boards. These performed better than the Raspberry Pi 2 cluster, executing the same algorithm almost six times faster than the reference computer. The price for the setup was again, half of the computer's.

Taking into consideration that Raspberry Pi 3 increased its CPU speed by 300 MHz, and upgraded its architecture to 64-bit, the performance boost should be quite noticeable. The manufacturer claims that Raspberry Pi 3 is 50% faster than its predecessor, used in the experiment. Among the other things that were upgraded were the GPU and RAM, which got their clock speeds increased[34]. The board has built-in Wi-Fi and Bluetooth.

2.4.4 Decision

The final decision is to go for the Raspberry Pi 3. The popularity of the board played an important factor in this choice. The resources should be enough for the system, with the only point of concern being the memory of 1 GB. However, as seen in section 2.3 Analysing resource consumption, it should be possible to fit the Kuiga Box tool in the given RAM.

2.5 Complete test setup

Based on the information gathered in the first phases of the analysis, the overall test setup can be defined. Its component parts can be seen in figure 2.7. The project's stakeholders are:

- Keysight, the company proposing the project. Andrea Cattoni is responsible for the project.
- The *d807f17* group (Embedded Software Systems), coordinated by Brian Nielsen. The group works on virtualising the IoT devices.
- Georgi Andonov (Computer Science), working for Keysight on the middleware.
- The *17gr1050* group (Wireless Communication Systems), working on the SDR.

2.5.1 Virtualised IoT devices

The goal of NTA is to have a number IoT devices virtualised on a Raspberry Pi 3 with the use of the Kuiga Box tool. Then, they will interface with the middleware to send packets of data. This traffic should be realistic and controllable to properly test the network.

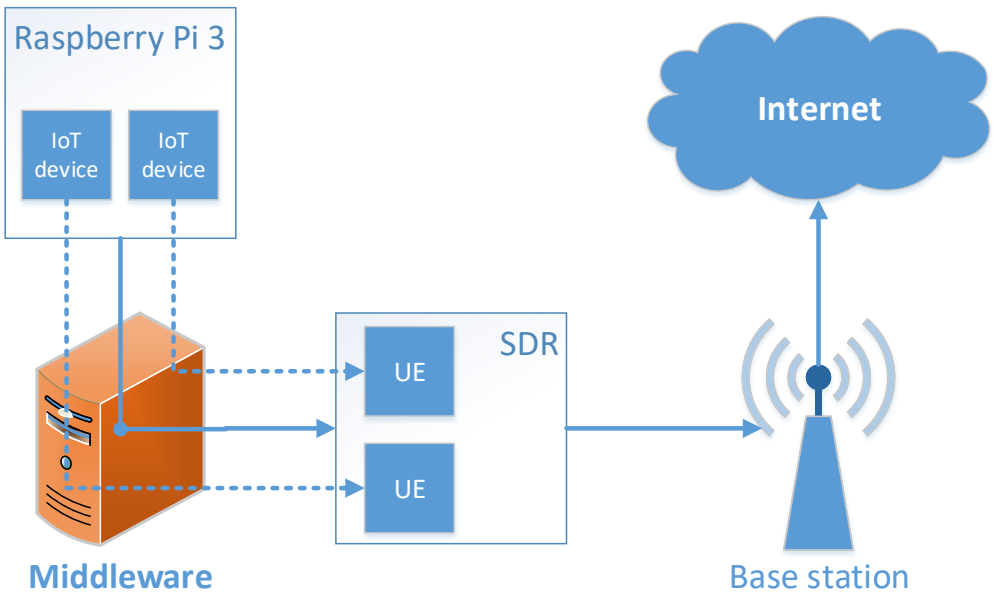

Figure 2.7: Practical test setup. The dotted lines between the virtualised IoT devices and the UE (User Equipment) represent the hypothetical connection between them, achieved by the Middleware

2.5.2 SDR

The SDR has the purpose of emulating the communication with the base station. The *17gr1050* group describes their project as:

> *[...] a "massiveMTC device", being a device that is to emulate a massive amount of MTC (machine type communication) devices for testing purposes. The massiveMTC device is an implementation of the communication protocol to be emulated for a lot of virtualised devices. This in itself can evaluate random access procedures (the access reservation protocol) of LTE. However, it can not alone produce network traffic at the eNodeB[6] and core network, as it would result from real devices accessing the network for data transmission. For this, the application layer behaviour must be added on top of the massiveMTC implementation - which is the purpose of the NTA [...]*

The massiveMTC device is composed of a computer, and an SDR board. The SDR board converts the frequency on which a packet is being sent. By using multiple frequencies, it appears as if data is being sent from different devices. The SDR board is an Ettus Research USRP B210[25]. The role of the computer is to perform LTE protocol layer functionality.

[6]Element of an LTE Radio Access Network

2.5.3 Middleware

Furthermore, the *17gr1050* group describes the middleware as:

> *[...] having the purpose of interfacing the massiveMTC device with the NTA, such that information packages [sic] can be passed between virtualised devices in NTA and its associated protocol stack in the massiveMTC device.*

2.5.4 Discussion

The middleware and the development of the massiveMTC device are two separate, ongoing projects. Their potential outcome is still unknown at this point in time. The groups communicate and keep each other updated, while consulting Keysight for advice and further requirements. As described in section 2.5.1, NTA's goal is limited and separated from the other two projects. For this reason, the rest of the documentation will focus on achieving this goal, by solving the problem described in Problem statement.

CHAPTER 3

Problem statement

The group will pursue to build a system that virtualises a number of IoT devices, running on an embedded platform, with the use of the Kuiga Box tool. This project definition is based on the current tendencies in the IoT world and on the demands from our industry contact. The system would be used to test the communication layer in an IoT context, primarily the LTE standard and its implementations, in order to find out how well it handles the load of numerous IoT devices.

This objective gives rise to the following problem statement:

How can a portable testbed for wireless cellular communication be built on a Raspberry Pi 3?

In relation to the topics investigated in Analysis, the following subquestions will be further looked into, in the following chapters:

- *How can Kuiga Box be migrated to an embedded platform taking into consideration the limited resources available there?*
- *What characteristics should the test application have?*
- *What facilities outside the embedded platform are needed to conduct the test?*
- *How should the system be controlled?*

Taking into consideration that this is a semester project, having limited time scheduled, the final result will not be a complete, fully-tested system. NTA has the main goal of showcasing the possibility of using embedded hardware, and is part of a series of continuous iterations towards a final testing setup.

CHAPTER 4

System Design

In this chapter the project is further analysed. Diagrams are given and notions are described, setting up the premises for implementing NTA. The topics discussed here also present relevant perspectives of how the project might evolve.

4.1 Methodology

Based on the problem and the subquestions formulated in the Problem statement the following methodology will be applied to design and build NTA:

- Adapt Kuiga Box to run on a Raspberry Pi 3
- Tweak its settings to run with limited resources
- Analyse and design the test application around the requirements stated by the industry contact
- Add some facilities to monitor and control the test application
- Load test the final system

4.2 System architecture

NTA is being built for Raspberry Pi using the Kuiga Box tool to manage Docker containers. These containers run a Docker image which represents a virtual IoT devices. The devices are supposed to send data packets using Ethernet. Figure 4.1 shows the external means of monitoring and controlling NTA. These are:

- the receiver of the data transmitted by the virtualised IoT devices. This is a mock of the middleware, SDR and antenna (from figure 2.7) replacing them in the development set up.
- the dashboard, providing a control interface for the how the data is being transmitted. Keysight's plan is to have a nice web interface to control the functionality of the test application. More information about this can be found in section 4.5.2. The data sink simply functions as a place to receive the traffic generated by NTA, in order to provide telemetry (packet loss, protocol, etc.).

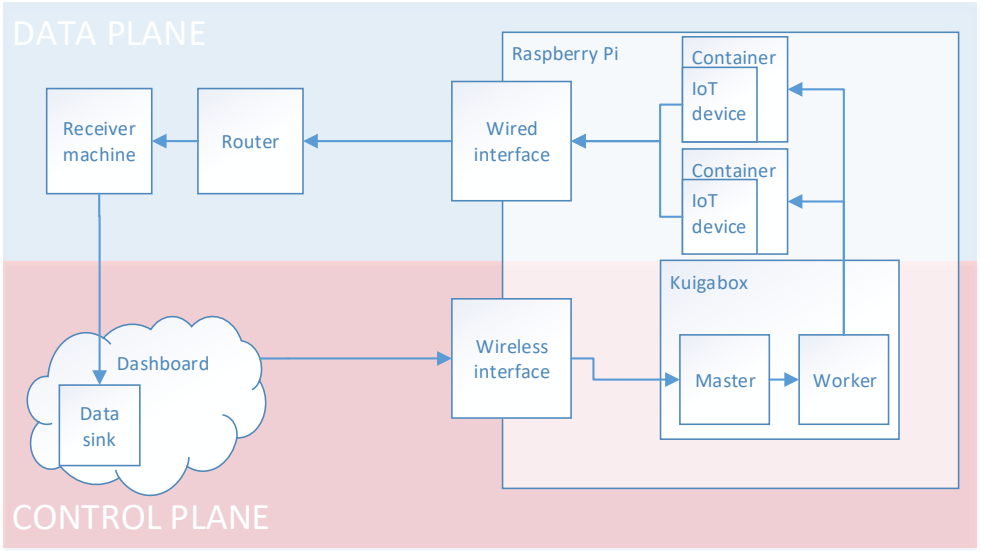

Figure 4.1: System architecture. The SDR and antenna are replaced with a laptop and router to emulate data and control taking separate paths in the final setup.

4.2.1 Physical setup

In the scenario in figure 4.1, the data and control planes are physically separated by using the two network adapters on the Raspberry Pi 3 (wired and wireless). The only reason behind this, is that it would be easier to implement the system this way. There is nothing preventing the whole system from both transmitting the data and controlling the application on one interface and using routing to separate their paths.

4.3 IoT devices

The IoT devices are virtualised as Docker containers. Their main goal is to send data packets to the base station. The right part of figure 4.2 shows the hierarchy of a container. The OS running inside the container has to be compatible with the host OS. In NTA's case the base image for building the Docker image (residing inside the container) is the same as the host: Raspbian. The application running in the containers is built on top of RIOT OS. The application is binary blob, built for the ARM platform, that is being placed inside the image, when Docker builds it. This means that the application can be used as a starting point for the container, when this is run.

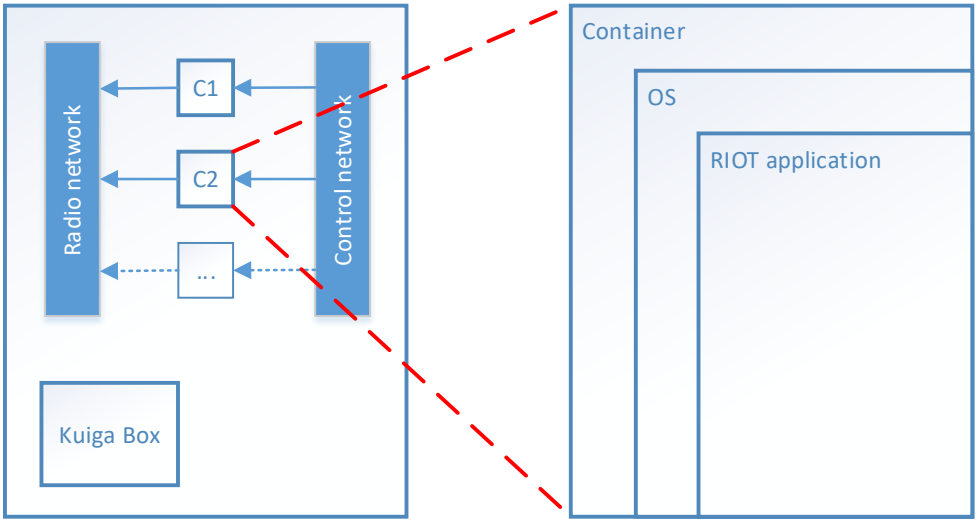

Figure 4.2: Close up of the architecture of a container. C1, C2 represent Docker containers. The networks they interface with are explained in section 4.4

4.3.1 Virtualisation in general

Virtualisation is a concept used throughout this report, and for this reason it is worth getting a deeper understanding of what it is, and how NTA can be built around it.
Virtualisation is the concept of creating a virtual computer environment, on top of a host machine. These virtual guest machines have access to the pool of resources of the host machine (CPU, RAM, storage).
Traditional virtualisation methods, *full-* and *para virtualisation*, virtualise an entire software stack for the guest operating system. This means that everything from the BIOS through the kernel, all the way to the applications are virtualised. This

allows for great flexibility, as the guests are generally independent from the host, but requires more resources, as both the software stack of the host, as well as one for all the guests are running.

More recently, a new method of virtualisation has emerged: *Operating-system-level virtualisation*. Here, everything from the kernel and below is shared with the host operating system. Compared to traditional virtualisation, this greatly reduces the virtualisation overhead, and greatly increases the scalability; more can be virtualised with the same hardware. Since the kernel is shared, it does add a restriction: the guest must be able to run with the host's kernel. This means that it is impossible to, for example, run a Windows guest on a Linux host, as Windows cannot run with the Linux kernel, and vice versa. It is, however, possible to virtualise one Linux distribution on top of another Linux distribution, for example, a Debian guest on a Red Hat Enterprise Linux (RHEL) host, as they are built around the same Linux kernel.

To reduce the resource overhead of Kuiga Box, it employs Docker, an Operating-system-level engine for virtualisation.

4.3.2 Virtualisation vs Emulation

In addition to distinguishing between different types of virtualisation, it is also important to distinguish between virtualisation and emulation. Compared to emulation, *virtualisation* means to take a real system, in the case of this project a real IoT device, and run the software in a virtual environment, complete with virtual sensors, virtual actuators, etc. In contrast, *emulation* means to run a piece of software, that to an outside observer looks and behaves like a real device. In the case of this project, where the system under test (SUT) is not the software, but the LTE base station, it is not important what the emulated IoT device does with the data it receives, or how it generates its own data. Instead, what is important is any data being transmitted back and forth between the IoT device and its backend service.

It is generelly the case, that the closer the testbed is to reality, the less scalable it is. This is the reason why the SW7 group chose to use Docker to virtualise IoT devices, rather than using an emulator, even though using Docker introduces the requirement, that the software must be runnable as a Linux process. For the same reason NTA will focus only on emulated IoT devices. This decision is based on several reasons.

1. It is not important for the SUT how traffic is generated, nor what the receiver does with it.

2. Using virtual IoT devices also requires emulating or virtualising sensors, actuators, etc.

3. As virtual IoT devices try to observe and manipulate the world around them when they are not transmitting data, they will use more resources

4. Virtual IoT devices will generate more realistic traffic than emulated IoT devices.

Because of point 2, developing realistic IoT software to virtualise takes longer, as not only the IoT software needs to be implemented, but also virtual or emulated sensors, actuators, etc. Because of point 3, using emulated IoT devices will require fewer resources, and it is therefore possible to run more IoT devices on the same hardware. Henceforth, the term virtualisation will be used in place of emulation.

4.4 Test application

The test application will run in Docker containers. Its main purpose is to send packets over the network, in a way similar to real IoT devices. To closely emulate a real IoT device, the test application should send packets that are similar to those real IoT devices send. For the purpose of testing the network, the actual contents of the packets are not important. Instead, the test application will send packets with the following typical characteristics:

- Size: 200 Bytes
- Periodicity: 1 second
- Protocol: UDP

Additionally, per request of Keysight, the test application is able to change the characteristics of the network packets during runtime. What Keysight wishes to be able to control is:

- Size
- Periodicity: from 100 milliseconds to 10 minutes
- Protocol: Either TCP or UDP
- Number of retransmissions (TCP only)

When Kuiga Box sets up Docker containers for a test run, it creates two separate virtual network through Docker. One network, `radio`, is intended to be connected to the middleware, and further on to the SDR and the antenna. This network is therefore intended as the data plane for tests. The other network, `control`, is

intended to be used to control the test applications.
Through the `control` network created by Kuiga Box, instructions on how to change the network packet characteristics are issued to the instances of the test application during runtime; each instance of the test application listens on the `control` network for these instructions. When one such instruction is received, the characteristics of all following packets are altered.

4.5 Control interface

As mentioned in section 4.4, through instructions issued over the `control` network, the test application can change the characteristics of the packets sent in the future. It is the responsibility of the control interface to issue these instructions to the test applications.

4.5.1 Command line interface

The simplest control interface is the command line tool aptly named `broadcaster`. Based on command line arguments, it broadcasts instructions to the running instances of the test application over the `control` network.

4.5.2 REST API

Keysight has suggested implementing a REST API to control the characteristics of the network packets through the internet. The web interface is intended to centralise the control of test runs, all throughout their life cycle.
Given the presence of the command line interface, the simplest way to implement the REST API for controlling active test runs, is to implement a simple REST server to route incoming requests with their parameters to the active instances of the test application through the command line application mentioned in section 4.5.1.

4.5.2.1 GUI interface

The motive for Keysight's wish to implement a REST API comes from the fact that, Keysight has a of group students in Turkey working on a graphical user interface to control test runs. This group uses REST as their way to interface with other systems. Unfortunately, no more information about the Turkish group or their work is known to this group.

4.6 Host monitor

The host monitor has the purpose of supplying information about the Raspberry Pi. In some cases, this can be achieved by using a small display, since the Rasp-

berry Pi has a DSI (Display Serial Interface) which can interact with touch screen displays. The motivation for having a host monitor, is primarily getting the IP address of the machine, to be used when connecting to it through SSH. Since NTA intends to use the Raspberry Pi without any screen and keyboard an alternative way has to be used, for getting information about the host.

An idea could be to get this information through Bluetooth. This way, the Raspberry Pi only needs to be paired once with another device, in order to transmit the information. This approach will be further explained in the following chapter.

4.7 Summary

The discussions in the previous chapters led to formulating a design for the testbed. However, due to the limited time allocated for the project, not all of these features will be implemented. The group will focus on:

- migrating Kuiga Box to the Raspberry Pi
- adapting the test application to transmit data packets having the characteristics defined in section 4.4
- developing a control program running on the host (Raspberry Pi), instead of having the Dashboard from figure 4.1 running in the cloud
- implementing a receiver program on the receiver machine seen in figure 4.1

CHAPTER 5

System Implementation

This chapter covers the functionalities implemented in the system. Diagrams and code snippets are included to make it easier to understand how the process was carried out.

5.1 System overview

Figure 5.1 represents the part of the system that is going to be implemented by NTA, based on the project's design, summarised in section 4.7. The Raspberry Pi is running the containers with the *test application*. The application sends data packets to a machine, which runs a *receiver* program. The motivation for doing this is that the packages can be seen as they arrive at the data sink represented in the left side of the figure. The *receiver* program monitors the packets characteristics (size, periodicity, protocol). These characteristics can be controlled using the *broadcaster* interface, which is running on the Raspberry Pi.

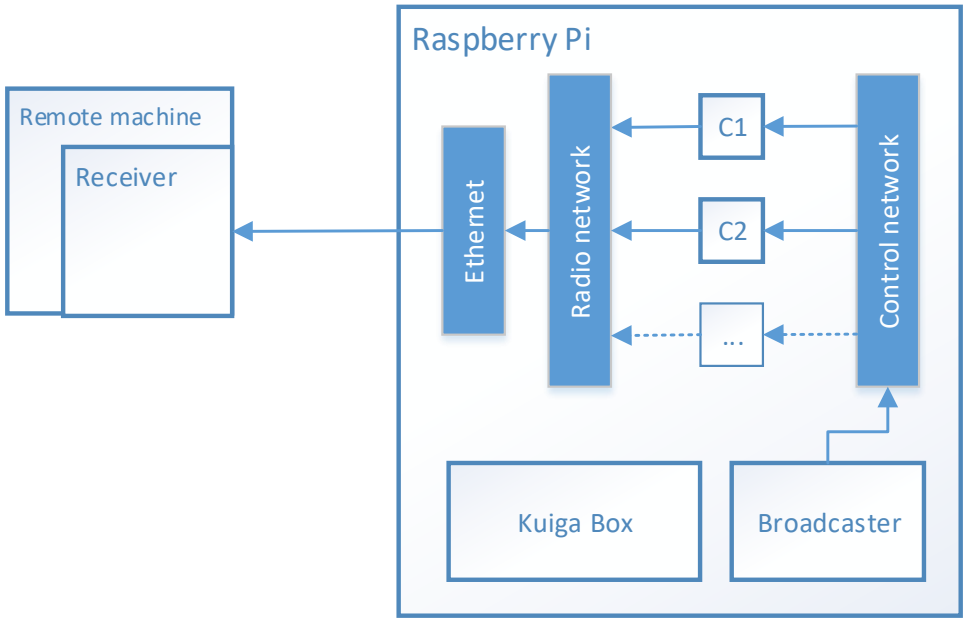

Figure 5.1: System overview. C1, C2 represent Docker containers. The remote machine is the data sink for the packets the test application sends

5.2 Hardware

The platform used for containing NTA is the Raspberry Pi 3. The board is enclosed in a plastic case, as seen in figure 5.2. As the intention is to have the board running without a screen and keyboard (headless system), the operating system written on the SD card is Raspbian Jessie Lite[1].

5.2.1 Basic host monitor

As the Raspberry Pi is running without a screen or keyboard attached, it is near impossible to find the IP address needed to connect to it remotely through SSH. Although it is possible to scan the network looking for the Raspberry Pi, it is impractical large networks. To overcome this challenge a bash script has been installed as a `cron job` to send the IP of the Raspberry Pi by Bluetooth (using the `rfcomm` protocol). A Bluetooth terminal, running for example on a smartphone then receives the IP addresses of the active network interfaces. This is a very basic monitor, serving the simple needs of NTA. In the future this can be further improved with the ideas suggested in section 6.1.

[1]Minimal image based on Debian Jessie - April 2017 version

Figure 5.2: Raspberry Pi 3

5.3 Deployment of Kuiga Box

The next step was to deploy the optimised version of Kuiga Box on the Raspberry Pi. Several steps have been taken to prepare the environment for it, as well as suiting the containers to the RIOT application. Things that had to be done:

- compile the JFFI[2] library for Raspberry Pi
- find the proper image to be used as base image for the Docker build process
- use a proper Embedded C development library for Raspberry for the test application
- increase the size of the swap

A more detailed explanation of these can be found in Appendix B.

5.3.1 Changes to Kuiga Box

Unfortunately, the version of Kuiga Box handed down from the SW7 group was non-functional. This was in part because of missing robustness on the side of the code, leading it to not function on new platforms. The main problem, though, was its dependence on Amazon S3.
The original version uses Amazon S3 as the storage medium for the Kuiga Box image store. After the conclusion of the SW7 project, the account used for S3 was likewise terminated. Without a storage medium, Kuiga Box obviously did not work.
For simplicity, and to keep in step with the idea of a portable testbed, the S3

[2]Java Foreign Function Interface

storage implementation is replaced with an implementation based on the local storage. Both the S3 image store and the local image store are implementations of the `ImageStore` trait[3], see listing 5.1.

```
1 trait ImageStore {
2         def uploadBaseImage(image: Image, inputStream: InputStream
              ): Option[URL]
3         def uploadCustomImage(image: Image, inputStream:
              InputStream): Option[URL]
4         def deleteImage(image: Image)
5         def getImageURL(image: Image): URL
6         def getAllImages(): ListBuffer[Image]
7         def getAllBaseImages(): ListBuffer[Image]
8         def getAllCustomImages(): ListBuffer[Image]
9 }
```

Listing 5.1: Definition of ImageStore trait

As can be seen from this trait, images are accessed via URL. The implemented local image store works directly on files. To serve images across modules inside Kuiga, the file path is converted to a URL, in compliance with the trait. Naturally, this means that the local image store does not work across a network. This therefore also removes the possibility to distribute work across several worker nodes.
If the local store must later be modified to again allow distribution of work, a standard server for serving files over HTTP or similar may be employed.

In addition to implementing a new image store, several smaller changes are also made to the code, to improve robustness. For example, the original Kuiga Worker looks for a an executable file named `middleware` to launch once all Docker containers are started. The purpose of this executable is to receive data from the containers and route it onwards. This approach to routing data to the SDR seems to have been based around a misunderstanding of what the SDR expects. This approach is no longer being used. As a result, this executable file is no longer present. Without it, however, the original version of Kuiga Worker throws an exception which is not caught when the file is not found, ultimately resulting in the worker no longer responding to incoming requests from Kuiga Master. While the code that looks for and tries to execute the file named `middleware` is still present and being executed, it is now guarded by a try/catch to remain running, even when no such file exists.
Other small changes include increasing the timeout times for the actor system Akka, to compensate for the slower Raspberry Pi (relative to the PC's it was originally developed on). As well, Kuiga Worker determines which platform it is run

[3]Traits are similar to Java Interfaces

on through a hardcoded value. To convince the worker it is running on an ARM CPU, this hardcoded value is changed to reflect the Raspbian platform.

5.3.2 Optimisations

The optimisations of Kuiga Box have focused only on reducing the memory footprint. Aside from using the memory restrictions found in the experiment described in section 2.3.3.1, `-Xms1m -Xmx128m -Xss2m -XX:+UseSerialGC`, the memory footprint of Kuiga Box has been further reduced by packaging Kuiga Web and Kuiga Worker in such ways, that `sbt` is no longer necessary. Rather, they are being run directly as Java applications. To see how they are packaged, see section 5.7.

5.4 Test application

The test application is implemented as a RIOT application in C. This means, that it is compiled together with RIOT OS, resulting in a single ELF image representing both RIOT OS and the application. For NTA, this ELF image is executed as what RIOT calls a *native* board: a native process running inside Linux that emulates bare metal management, such as process control, memory management, etc., using Linux system calls.
For faster development, it is based on the simple test application developed by the SW7 group to demo Kuiga Box. Though the NTA test application is heavily modified, it uses the same approach to networking: BSD sockets.
Rather than executing the ELF image directly, Kuiga Box, through Docker, launches the `start_client` script, which renames the two network interface created earlier by Kuiga Box to `radio` and `control`; their original names are arbitrarily generated names given by Docker. Additionally, the `start_client` script sets up certain environment variables used later by the RIOT application. This script is inherited from the SW7 group, and very few changes have been made to it.
On start up of the ELF image, the application initialises two BSD sockets: one used to send data over the `radio` network, and one used to listen for instructions on the `control` network. Once these sockets are initialised, the application creates two RIOT threads that transmit and listen for packets, respectively.

5.4.1 Socket initialisation

During start up, the test application reads certain network parameters passed to is from Kuiga Box through environment variables. These parameters are:

- `CONTROL_DEV`: The name of the control network interface
- `CONTROL_PREFIX`: The prefix for the control network subnet

- RADIO_PREFIX: The prefix for the radio network subnet
- PORT: The port to use when sending packets. The next port, PORT+1, is used when listening for instructions on the control network.
- CONTAINER_ID: The ID of the Docker container the specific instance of the test application is running inside.

5.4.1.1 Control socket

The socket for the control network is created as an IPv4 UDP socket. It is bound to listen to addresses covered by the mask CONTROL_PREFIX.255.255 on port PORT +1 on the interface named control. CONTROL_PREFIX is set by Kuiga Worker, and is subnet used for the control network created in Docker by Kuiga Worker. The value of CONTROL_PREFIX is hardcoded in Kuiga Worker to be 10.193.

5.4.1.2 Radio socket

The socket for the radio network is initially created as an IPv4 UDP socket, though it can later be recreated as an IPv4 TCP socket. The description of the destination is also set up at this time, to use the port denoted by PORT and to a hardcoded IP address.

5.4.2 Control

Handling of the control network resides in a thread by itself. This is because listening on a socket is a blocking operation, and because it is the easiest way to implement a reliable server. The control thread listens on the control network for UDP packets with instructions from the broadcaster application (see section 5.5). A packet sent on the control network must have the following format:

```
[type]:[value]
```

[type] must be a string encoded decimal number denoting which parameter of the network communication that must change, while the content of value depends on the parameter. The parameters of network communication that can be changed are protocol, periodicity and packet size. The decimal ID values denoting parameters are determined in the parameter enum, see listing 5.2. The purpose of encapsulating this in an enum, is to standardise the numeric values in a portable way; the enum is located in a separate file, and the very same file is also used by broadcaster

For PROTOCOL-type instructions, the value must be a string tcp or udp[4]. For PERIODICITY-type instructions, the value must be a string encoded decimal with

[4]Both are case sensitive

```
typedef enum parameter {
    PROTOCOL = 1,
    PERIODICITY = 2,
    PACKET_SIZE = 3,
} parameter_t;
```

Listing 5.2: `parameter` enum

a value greater or equal to 100. This value is the packet period in milliseconds. For `PACKET_SIZE`-type instructions, the value must be a string encoded decimal with a value between 3 and 65507[5], both included. This value is the size of the packets to be sent in bytes.
If the type is not recognised, or if the value falls outside of the bounds accepted for the defined types, then nothing is changed.

Due to the way TCP sockets themselves handle retransmission, changing the number of retransmissions for TCP connections, otherwise considered in the design, is not implemented.

5.4.3 Radio data

Generating data for the radio network is handled in a separate RIOT thread. Using the function `xtimer_usleep(int)` from the RIOT module, `xtimer` controls the periodicity of data packets sent. To minimise drift, the time it took to send the last packet is subtracted from the sleep time. Listing 5.3 shows this. `xtimer_usleep(int)` parks the RIOT thread for a given number of microsec-

```
unsigned long long start, end;

while(1) {
        start = xtimer_now_usec64();
        sendPacket();
        end = xtimer_now_usec64();
        xtimer_usleep((periodicity * 1000) - (unsigned int)(end -
            start));
}
```

Listing 5.3: The code responsible for maintaining the given periodicity. Note `periodicity` is an `unsigned int` of milliseconds.

onds. Because RIOT is aimed at single core boards, it does not support execution of multiple threads in parallel, even when running as a native Linux process. It is

[5]65507 bytes is the maximum size of a IPv4 UDP packet

therefore important that, as the `xtimer` functions do, the function used for waiting puts the thread to sleep.
The packets to be sent are constructed with an initial character <. To this is appended as many times as the packet size will allow, the entire ID of the Docker container (64 bytes). In the remaining space as much of the ID that will fit is copied into the packet, followed by `>\n`. A packet may look like:

```
<3be04e19f016f041f25f32557c5d174675028a42efc31269
4a4c0d4d86c8442a3be04e19f016f041f25f32557c5d17467>
```

Here, the packet size is 100 bytes, and the container ID is `3be04e19f016f041f25f32557c5d174675028a42efc312694a4c0d4d86c8442a`. As it can be seen, the entire container ID only fits into the packet once. After this only the first 33 characters of the container ID will fit. This sets the lower bounds on the packet size to 3 bytes, while the upper bound is the UDP upper bound 65507 bytes.
If the application is to send UDP packets, as is default, they are sent as is. If however, the application has been instructed to send TCP packets, for every packet sent a new TCP socket is created and connected to the receiver after which the packet is sent. Once send, the socket, and thereby also the connection, is closed.
The reason a new socket is created for every packet is, that a TCP socket cannot be reused once a connection is closed, which the receiver may do at will.

5.5 Broadcaster

`broadcaster` is a C-based command line utility made to issue instructions to running test applications. It uses switches to determine which instructions to send to the test applications. Tables 5.1 shows all available switches and their meaning. If any switches are passed, `broadcaster` broadcasts instructions to all run-

Switch	Explanation
-p <value>	Instructs test applications to change protocol to <value>. <value> must be `tcp` or `udp`
-P <value>	Instructs test applications to change the period to <value> milliseconds
-s <value>	Instructs test applications to change the packet size to <value> bytes
-h	Prints help text

Table 5.1: `broadcaster` switches

ning test applications through the control network. Once all switches are parsed, `broadcaster` creates a UDP broadcast socket and for each passed switch, broadcasts an instruction to the 10.193.0.0/16 address range, which is subnet used by

the control network. The broadcast is sent on port 5557, which is equivalent to `PORT+1` in the test applications (see section 5.4.1).
The instructions are sent in same the format

```
[type]:[value]
```

An example of an instruction is `3:100`. This instruction asks the test application to change the packet size to 100 bytes. This is the same format as seen in section 5.4.1. The value of type comes from a copy of the `parameter` enum also used in the test applications, see listing 5.2.

At the moment, any instructions sent with `broadcaster` are received by all running test applications. If a more fine grained control of which instances should receive the instruction is necessary, it may be implemented by making `broadcaster` use unicast or multicast to target individual or groups of running test applications. This, however, would require that `broadcaster` gain a knowledge of which IP address the individual instance has. Alternatively, the format of the instructions may be extended to include container IDs. While the instructions would continue to be transmitted as broadcasts, it would be up to the individual test applications to compare their container ID to the one included in the instruction, and if so change its parameters. This, naturally, would require changes to both `broadcaster` and the test application.

5.6 Receiver

The receiver will run on a different machine, and receive the data packets being sent by the test application. This was developed in C++.

The receiver works by opening two sockets (TCP and UDP), and listens on them. It then prints the size of the packets it eventually receives along with the packet itself. The two listeners run on two concurrent threads.

When started, the receiver takes a port number as an argument. The socket is created, and the host information is put in a struct. This includes the port number to be listened on, and is used to identify the socket. The socket is then bound to this struct, as to *assign a name to the socket*, as seen in listing 5.4 (for the UDP socket): The creation of the UDP and TCP sockets is similar. However, they differ in the way they function. UDP sockets are connectionless, while for TCP sockets a connection needs to be established first, and they are recreated after their connections are closed.

```
22 socketFD = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
23
24 memset(&sa, 0, sizeof sa);
25 sa.sin_family = AF_INET;
26 sa.sin_port = htons(port);
27 sa.sin_addr.s_addr = htonl(INADDR_ANY);
28
29 bind(socketFD, (struct sockaddr*) &sa, sizeof sa);
```

Listing 5.4: Initialisation and binding of UDP socket

5.7 Packaging NTA

NTA consists of a number of compressed files, scripts, programs and services. In order to provide a ready to use package, the system can be installed using the provided `Makefile`. What this achieves is:

- Compiling the test application, the `broadcaster` and the `receiver`
- Making the `broadcaster` globally executable
- Extracting the archives containing Kuiga Web and Kuiga Worker to `/opt/`
- Copying the platform specific `Dockerfile` to a folder in Kuiga Web to be used during the image build process
- Installing two `systemd` services called `nta_web` and `nta_worker` which run the Kuiga Web and Worker with the optimised JVM options

The Kuiga Web archive is packaged with Lightbend Activator[19], a framework used by Kuiga Web. Packaging can be done by running the command `activator dist`. This will create a ready-made zip archive with all the files necessary to run Kuiga Web.
The Kuiga Worker archive is packaged with IntelliJ IDEA by JetBrains[17]. IntelliJ IDEA allows a Scala build task, as defined by the project files, to be packaged as an artifact. Because of code signed dependencies, Kuiga Worker cannot be packaged as a standalone JAR files. Instead, based on the manifest, the dependencies of Kuiga Worker are loaded from the current working directory. When IntelliJ creates the artifact, it copies all JAR dependencies to the same folder as the artifact. This folder is manually packaged in a zip file.
The motivation for packaging these two, is to take the burden of compilation away from the Raspberry Pi, as well as making deployment easier.

5.8 Experiment

In order to test the capabilities of the system the following experiment was conducted.

Objective
The focus of the experiment is showing how many virtual IoT devices the system is capable of running simultaneously. This is compared to the original Kuiga Box to see which improvement the work for this project has yielded. Since most of the development of NTA was done on a virtual machine, with similar memory and performance as the Raspberry Pi 3, this experiment will also show how the system performs on the embedded platform.

Setup
The original Kuiga Box is run through `sbt` with the original JVM parameters: `-Xms512m -Xmx1024m -Xss2m`. To test the system developed during this project, NTA is deployed on the Raspberry Pi, as described in 5.7. This is being run with the following JVM parameters: `-Xms1m -Xmx128m -Xss2m -XX:+UseSerialGC`.

After each test, the system is rebooted and the Docker environment is cleaned by running the script `docker_clean.sh`. This stops and removes all containers, and deletes images and their copies.
A `swap` file on the Raspberry Pi is set to 1 GB. During the experiment different `swap` setups are also experimented with: using `zram` (compressed in-memory swap) and having a large partition on a USB flash drive (more details about these in Appendix B).

The results are compared to NTA running in a virtual machine, set up with Debian 8, 1 vCPU, 1 GB RAM and 1 GB `swap`, to, as closely as possible, mimic the performance of a Raspberry Pi 3.

Results
The initial test runs conducted on the Raspberry Pi show a limit of 40 running virtual IoT devices. The system can safely create and run these virtualised devices, where the test application sends packets with the typical characteristics mentioned in section 4.4. These were tested for both UDP and TCP protocols.

When trying to handle more devices, the system unpredictably fails. In order to test different approaches for increasing the number of active devices, the subsequent test runs were set up for 50 virtualised devices.

The test runs based on 50 virtualised devices had the following behaviour when failing:

- Set up test run via the web interface
- Kuiga Master builds the image, and successfully copies it to the image store
- Kuiga Master provides the image to the worker
- Kuiga Worker unpacks the image and starts creating the containers based on it
- 50 containers are created
- Using the web interface, the test run is started, meaning that the containers begin running the test application one by one
- During the starting phase, somewhere between the 41th and 50th container, the system fails. At this point, 350 MB of RAM are being used, and 0 MB swap.
- The running containers receive a `SIGTERM` signal and terminate with a `143` exit code (seen in Docker logs)
- The remaining containers, that had yet to be started, keep their state: created but not running
- The environment is cleaned up with and the test run ended, by running `docker_clean.sh`

`SIGTERM` is a signal received from either the kernel or another application to terminate a process. In the presented scenario, it can be an indication of the OS running out of memory, but it can also mean something else. The hypotheses the group has for the cause of the `SIGTERM` signal are:

1. Not enough total memory (RAM + swap)
2. Too slow swap file, causing the OS to panic and close the containers
3. Something preventing Java from being swapped
4. A problem with the kernel
5. A problem with Docker

The first two assumptions are tested out by creating a large swap partition (4 GB) on a USB flash drive. This is given a higher priority than the original swap. In total, the system therefore has 5 GB of swap, and will swap to the faster USB flash drive before the slower micro SD card. The behaviour of the test runs is the same.

In case the continued failure is because the USB based swap is still too slow, the second hypothesis is also tested by adding 500 MB of zram. Since this is a form of in-memory swap, it is much faster, at a slight cost in CPU time. Again, its priority

is higher that that of the original swap. The system now has 1.5 GB swap in total. With this setup, the behaviour is the same.

Another approach is to let the test runs settle in by initiating two of them, at different times (approximately 10 minutes between them). The assumption is that this will give the kernel enough time to swap out more from RAM. The first test run starts 25 containers. The second test run is then created with 25 containers, and they are then started. After the 15 first containers of the second test run are started, the system fails, and both the containers from the first and the second test run are terminated.

The third assumption was tested by forcing Kuiga Web and Kuiga Worker to swap. This was achieved by running a program allocating large chunks of memory. When the system ran out of RAM it began using swap (5.5 GB). The resident size of the two processes (Kuiga Web and Kuiga Worker) was reduced by a factor of 10, in comparison to when the system was not using swap. This is a good indicator that the two processes can actually be swapped, and Java is not the culprit of the system failure.

The forth and fifth hypotheses need to be further investigated. A test run is set up, having 50 containers, under the assumption that most of them will be terminated. The focus is on one of the containers, analysing the system calls it makes, and the signals it receives. The first container started by Kuiga Worker was used for this purpose. `strace` (a simple debugging tool) was attached to the container, and its actions were monitored. Indeed, it could be seen that when the system failed, the container received a `SIGTERM` signal. This was transmitted from PID 0, a process started by the kernel itself. Usually, the `swapper` or the `scheduler` have process ID 0 and are responsible for paging[33]. The behaviour of the swapper has been investigated throughout the other hypotheses, and does not seem to be the cause of the failure. The scheduler decides which threads run on the CPU. The group had a theory that this could be overwhelmed by the large number of processes Kuiga Worker starts (four for each container). However, this theory was quickly disproven, by looking at the maximum number of processes (`pid_max`) and threads (`threads-max`) the OS can handle.

```
pid_max 32768
threads-max 15434
```

The system has 598 user-threads and 99 kernel-threads when running 40 containers. Further investigations are needed for understanding the cause of not being able to run more than 40 containers.

Similar tests were run on the Debian 8 virtual machine. In this environment 150 containers can run, when sending the default data packets. This results in using 900 MB RAM and 350 MB swap. When running this many containers, the network bandwidth capabilities need to be considered. 150 containers sending 50 kB packets over UDP will use approximately 50 Mbps bandwidth. If NTA will be further expanded, the 100 Mbps Ethernet of the Raspberry Pi will prove to be a physical limit for the maximum number of IoT devices the system can host.

Finally, Kuiga Box was run on the Raspberry Pi, with its original parameters through `sbt`, as it was originally run. The group had experienced the original Kuiga Box running on a virtual machine with limited resources, but wanted to see how the original system would perform on the embedded platform. When the Worker and the Web interface are being loaded, the system runs out of RAM and begins swapping around 100 MB of data. It proved to be difficult to have a valid test run. The Akka actor system fails to maintain connections between the Kuiga Master and the Worker. This means that the Docker image can be built, but the worker is unable to receive the command to start creating containers based on the image. The only trial that managed to start up containers had the following behaviour:

- The image is successfully built, and 50 containers are created
- The test run is started
- After starting 25 containers, the Akka actor fails
- Kuiga Worker continues starting the remaining containers, until it reaches 50 running containers
- Shortly after, all the containers are terminated (`SIGTERM` signal)

This last test confirms that the lack of memory is not good for Kuiga Box. While the optimised version manages to maintain medium sized tests, the original version cannot cope with the lack of resources.

Discussion
The tests conducted in this experiment show the capabilities of the system. These were also beneficial for further improving the project as a whole, small changes being done to the test application and the control interface during these tests. Also, different shortcomings have been noticed, which could be addressed in the future. These are mentioned in the Perspectives section of the Conclusion.

5.9 Features of the system

In the end of the implementation phase, NTA has the following features:

- **Kuiga Box can be successfully run on a Raspberry Pi 3**
  At the moment, Kuiga Box is no longer able to be distributed among several nodes. Additionally, Kuiga Worker still uses a hardcoded value to determine the CPU architecture of the host.
- **The Raspberry Pi board is used to virtualise 40 IoT devices**
  It may be possible to extend this number by tuning the kernel, but how to do this has not yet been discovered.
- **The virtualised IoT devices run a *test application* sending data packets**
  The characteristics of the packets sent can be changed at runtime. Currently, the IP address of the receiver is hardcoded in the test application. Additionally, TCP retransmission cannot be changed during runtime.
- **The *broadcaster* is used to change the characteristics of the data packets**
  It is currently being run on the host machine, with no way of launching it remotely, except through SSH. Additionally, the broadcaster is not able to target individual running test application.
- **The data packets are seen arriving at the destination, which is a remote machine running the *receiver* program**
  The receiver is still simple is its implementation, and cannot yet be used for gathering extensive telemetry; it merely echoes incoming packets.

CHAPTER 6

Conclusion

NTA is capable of virtualising IoT devices which run a test application generating traffic that can be controlled. The system runs on a Raspberry Pi 3, and was tested to find the number of virtualised IoT devices it can handle.

The questions formulated in the Problem statement have been addressed throughout this report:

How can Kuiga Box be migrated to an embedded platform taking into consideration the limited resources available there?
Kuiga Box is able to run on an embedded platform, in this case a Raspberry Pi 3, by tweaking the settings of the JVM. In addition, small changes were needed, in order to remove the external dependencies of the original system.

What characteristics should the test application have?
The test application should be able to transmit the specific kind of traffic required for testing LTE-networks. The characteristics of the data packets were communicated by Keysight. The test application should also be able to change these characteristics at runtime.

What facilities outside the embedded platform are needed to conduct the test?
In the practical test setup Keysight wants to have, there needs to be an SDR to send the packets with the LTE protocol. The packets coming from the IoT devices will be directed to the SDR, by some middleware. In the test setup NTA used, data is not routed through a mock middleware, but instead sent directly over the internet to the receiver. The receiver is a small data sink implemented to not only acknowledge packets, but also to prove they are received. In the test setup, the receiver is

located offsite, to ensure data travels over the internet.

How should the system be controlled?
The test runs are controlled through the web interface integrated in the original Kuiga Box. Running test applications are controlled by a program broadcasting changes to the data packets characteristics, on one of the networks the virtualised IoT devices are connected to.

Status on the Middleware and SDR
The other two projects related to NTA have also made progress. The middleware ended up being a VPN server, where the nodes with Kuiga Worker connect as VPN clients. Traffic coming from the virtualised IoT devices is transmitted over VPN to the middleware and is forwarded to the private network the SDR is on. This is received by a router on the SDR-side, which forwards it to the User Equipment (UE). The SDR group tested their project and achieved having five UE devices transmitting data over Telenor's network.

6.1 Perspectives

As can be seen in section 5.9, there are still many improvements to be made on NTA before it is suitable for field work. The group suggests looking further into the following:

- To achieve Keysight's goal of 1000 virtual IoT devices with Raspberry Pis, or similar hardware, it is clearly necessary to once again turn to a distributed setup. As such, working on making Kuiga Box work distributed once again is suggested.
- Experiments on x86 virtual machines, show that it is possible to run more than 40 containers, if Kuiga Box is swapped out. It would be worthwhile to investigate further, why it is currently not possible on the Raspberry Pi.
- Start separating the routing of the data and control plane, in accordance with the design in section 4.2. The motivation for this is to avoid influencing the test runs with the control instructions.
- Use the experience gathered when developing the receiver and the broadcaster to design the dashboard, mentioned in section 4.2, for controlling the testbed.
- Look into the aspect of virtualising IoT devices running different OS'es.
- To make controlling NTA easier, a REST interface for Kuiga Box, as well as the broadcaster should be implemented.

- Make it possible to change the IP address of the receiver in the test application alongside packet characteristics.
- Extend the functionality of the host monitor mentioned in 4.6 and 5.2.1. Additional features may include transmitting the platform characteristics of the host, a regular heartbeat to the dashboard mentioned section 4.2, state of IoT devices, state of NTA services, etc.
- When Kuiga Box sets up new test runs, the Docker image is completely rebuild from the base image. All of the time, this is redundant work. The group suggests modifying Kuiga Box in a way, that it can reuse the Docker images it builds.

Bibliography

[1] Inc. Adapteva. *E16G301 EPIPHANYTM 16-CORE MICROPROCESSOR - Datasheet*. http://www.adapteva.com/docs/e16g301_datasheet.pdf. 2013.

[2] Arduino AG. *Arduino Yún LininoOS*. https://www.arduino.cc/en/Main/ArduinoBoardYun. 2017.

[3] Casper Møller Bartholomæussen et al. *Kuiga Box: A Test Bed for Large Scale Testing of Virtualized IoT Devices*. Dec. 2016.

[4] Andrew Bindelglass. *Tech Companies Test LTE IoT Network*. http://next-generation-communications.tmcnet.com/topics/nextgen-voice/articles/429882-tech-companies-test-lte-iot-network.htm. 2017.

[5] Ben Cordero. *Parallella vs the Pi*. https://blog.condi.me/parallella-vs-the-pi/. 2015.

[6] Intel Corporation. *Intel Compute Stick*. http://www.intel.com/content/www/us/en/compute-stick/intel-compute-stick.html. 2017.

[7] Efa at English Wikipedia. *Raspbian Pi 3 Board Layout*. https://commons.wikimedia.org/wiki/File:RaspberryPi_3B.svg.

[8] Inc Extreme Engineering Solutions. *XPedite6401*. https://www.xes-inc.com/products/processor-mezzanines/xpedite6401/. 2017.

[9] Ł. Faber and K. Boryczko. "Efficient parallel execution of genetic algorithms on Epiphany manycore processor". In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2016, pp. 865–872.

[10] Raspberry Pi Foundation. *Raspbian*. https://www.raspberrypi.org/downloads/raspbian/.

[11] Jayavardhana Gubbi et al. *Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions*. https://arxiv.org/ftp/arxiv/papers/1207/1207.0203.pdf. 2013.

[12] Docker Inc. *Docker*. https://www.docker.com/.

[13] Docker Inc. *Get Docker for Ubuntu*. https://docs.docker.com/engine/installation/linux/ubuntu/.

[14] Ericsson Inc. *Every. Thing. Connected. - A study of the adoption of 'Internet of Things' among Danish companies*. http://digital.di.dk/SiteCollectionDocuments/Analyser/IoT_Report_onlineversion.pdf. 2015.

[15] *Java HotSpot VM Memory Footprint Optimization*. http://javagc.blogspot.dk/2013/03/java-hotspot-vm-footprint-optimization.html. 2013.

[16] *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*. http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html.

[17] JetBrains. *IntelliJ IDEA*. https://www.jetbrains.com/idea/.

[18] Lattepanda. *LattePanda 2G/32GB*. http://www.lattepanda.com/product-details/?pid=1d. 2017.

[19] Lightbend. *Lightbend Activator*. https://www.lightbend.com/activator/download.

[20] S. J. Matthews, R. W. Blaine, and A. F. Brantly. "Evaluating single board computer clusters for cyber operations". In: *2016 International Conference on Cyber Conflict (CyCon U.S.)* 2016, pp. 1–8. DOI: 10.1109/CYCONUS.2016.7836622.

[21] Oracle. *Java HotSpot VM*. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.

[22] Oracle. *OpenJDK*. http://openjdk.java.net/. 2017.

[23] Parallella. *About The Parallella Project*. URL: https://www.parallella.org/about/.

[24] PINE64. *PINE A64+ 2GB BOARD*. https://www.pine64.org/?product=pine-a64-board-2gb. 2017.

[25] A National Instruments Company ttus Research. *USRP B210 (Board Only)*. https://www.ettus.com/product/details/UB210-KIT.

[26] Matt Richardson. *Docker comes to Raspberry Pi*. Aug. 2016. URL: https://www.raspberrypi.org/blog/docker-comes-to-raspberry-pi/.

[27] Keysight Technologies. *Keysight*. http://www.keysight.com/.

[28] Tessel. *Tessel 2*. https://tessel.io/b. 2017.

[29] W. Trappe, R. Howard, and R. S. Moore. "Low-Energy Security: Limits and Opportunities in the Internet of Things". In: *IEEE Security Privacy* 13.1 (2015), pp. 14–21. ISSN: 1540-7993. DOI: 10.1109/MSP.2015.7.

[30] ODroid UK. *Odroid C2 - 64-bit quad-core Single Board Computer*. http://www.odroid.co.uk/hardkernel-odroid-c2-board. 2017.

[31] *VisualVM*. https://visualvm.github.io/index.html.

[32] Job van der Voort. *GitLab on Raspberry Pi 2!* Apr. 2012. URL: https://about.gitlab.com/2015/04/21/gitlab-on-raspberry-pi-2/.

[33] Wikipedia. *Process identifier — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Process%20identifier&oldid=782568316. [Online; accessed 30-May-2017]. 2017.

[34] Andrew Williams. *Raspberry Pi 3 vs Pi 2*. http://www.trustedreviews.com/opinions/raspberry-pi-3-vs-pi-2. 2016.

APPENDIX A

Overview of the development boards investigated

The following table consists of the development boards that the group looked into, during the analysis phase. Information is given for each board based on the criteria established at the beginning of section 2.4.

Board	Performance	OS support	Availability & Price	Size & Power consumption
Parallella	dual-core ARMv7 1GHz CPU 16-core Epiphany Coprocessor 1GB RAM	Linux	The department has it ∼1100 dkk from RS components	Credit card size 5W (typical)
Raspberry Pi 3	quad-core ARMv8 1.2GHz CPU 1GB RAM	Linux	Shipment from Denmark ∼600dkk from raspberrypi.dk	Credit card size 1.5W (idle) 6.7W (peak usage)
Odroid C2	quad-core ARMv8 1.5 GHz CPU 2GB RAM	Linux	Shipment from the UK ∼500dkk from odroid.co.uk	Credit card size 1.8W (idle) 4W (peak usage)
LattePanda	quad-core Atom 1.84 GHz CPU ATmega32u4 coprocessor 2GB RAM	Windows	Shipment from Denmark ∼730dkk from RS components	88 x 70 mm 10W (maximum)
Pine A64	quad-core ARMv8 1.2GHz CPU 2GB RAM	Linux	Shipment from China ∼200dkk from pine64.org	129 x 79 mm 1.6W(idle) 4.2W(peak usage)
Intel Compute Stick	quad-core Atom 1.84 GHz CPU 2GB RAM	Windows	Shipment from Denmark ∼1200 from bilka.dk	123 x 38 mm 15W (maximum)

APPENDIX B

Raspberry Pi setup tips

This appendix covers some ARM platform specific steps that needed to be taken in order to run Kuiga Box. The following list describes some of the problems encountered, and the solution to them, when trying to run the original version of Kuiga Box on the Raspberry Pi:

1. The docker-client library used to make Docker API calls fails when building an image. This is due to the incompatible JFFI (Java Foreign Function Interface) library provided with Kuiga Box. The solution is to build the JFFI library for ARM Linux. More information here:

   `https://github.com/spotify/docker-client/issues/477`

2. When building a Docker image, to be run in a container, Docker needs to get a *base image* to use. The *base image* is a snapshot of an OS filesystem, typically the host OS. For this reason, the Dockerfile used for building images for the Raspberry Pi uses `jsurf/rpi-raspbian` instead of the original `debian` *base image*.

3. The RIOT application requires some embedded C libraries to run properly in the containers. These are installed when building the Docker image. However, it is important to choose the ones compiled for ARM. The group went for the stable version of `libc6-dev`. Other versions that could be used (these have not been tested) can be found here:

   `https://packages.debian.org/search?keywords=libc6-dev`

4. In order to load test the system, the size of `swap` was increased to 1GB. The assumption is that after Kuiga Master build the Docker image, and after Kuiga Worker starts the containers, they will be paged out to `swap`. This way, the containers would use RAM when running. The `swap` was altered using the *dphys-swapfile* approach, explained here:

   `https://raspberrypi.stackexchange.com/questions/70/how-to-set-up-swap-space`

During the test phase, different options for `swap` had been tested. These were:

- *zram*, which is a way of compressing swap, and have it operate on RAM. This was achieved using a script that can be found here:

  `https://gist.github.com/sultanqasim/79799883c6b81c710e36a38008dfa374`

- Having a large (4GB) swap partition on an USB flashdrive. This can sometimes be beneficial, as an USB2 device connected to the Raspberry Pi can be faster that the microSD card. In order to do this, the USB flashdrive was formatted as `ext4` and the following guide has been followed to make it swap:

  `https://askubuntu.com/questions/173676/how-to-make-a-usb-stick-swap-disk`

- Tweaking with the swappiness value. This can have a value between 0 and 100, which relates to RAM usage and how fast the programs will be swapped out. For example:

  `sysctl vm.swappiness=100` which sets it to 100