



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SAMI NÄPPÄ
CONTINUOUS INTEGRATION IN RURAL LTE BASE STATION
SOLUTION

Master of Science Thesis

Examiner: prof. Hannu-Matti Järvinen
Examiner and topic approved on
31 January 2018

ABSTRACT

Sami Näppä: CONTINUOUS INTEGRATION IN RURAL LTE BASE STATION SOLUTION

Tampere University of technology

Master of Science Thesis, 46 pages

May 2018

Master's Degree Programme in Information Technology

Major: Software Engineering

Examiner: Professor Hannu-Matti Järvinen

Keywords: continuous integration, LTE, 4G, Robot framework, Gitlab

Mobile network coverage in rural areas are usually a challenging topic for mobile network operators. Setting up a remote base station and operating it might not be a wise investment, since the number of customers served in the rural areas is usually low. Kuha is a name for a rural base station solution that is designed to tackle the issues by designing a business model that decreases the costs for the operators to extend their already existing network. This rural area connectivity is achieved by utilizing already existing hardware and software to some extent.

In order to achieve this type of non-traditional connectivity, part of the software running on the base station has to be reimplemented. Since part of the software is reimplemented, it also needs to be tested. Due to the fact that this reimplemented software is very profound software component within the base station, the continuous integration is a wise choice compared to continuous delivery or continuous deployment.

In this thesis we will explore the evolution of the mobile network technologies, and how Kuha is addressing the issues that the rural areas introduce. As a practical part of this thesis, a continuous integration system was created. Theoretical part introduces different types of testing and also how continuous integration fits to the traditional and modern software development methodologies.

As a result of the work done, a continuous integration system was established. This system has been in use since the project started and will be under further development even after this thesis. The system, at the time of writing this thesis, supports a software compilation and integration testing. What it comes to the future of the system, a move towards continuous delivery or continuous deployment will be a next step to take.

TIIVISTELMÄ

Sami Näppä: Maaseuduille tarkoitetun LTE-tukiasemaratkaisun jatkuva integraatio

Tampereen teknillinen yliopisto

Diplomityö, 46 sivua

Toukokuu 2018

Tietotekniikan diplomi-insinöörin tutkinto-ohjelma

Pääaine: Ohjelmistotuotanto

Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: jatkuva integraatio, LTE, 4G, Robot framework, Gitlab

Matkapuhelinverkkojen kattavuus maaseuduilla on yleensä keho, sillä operaattorit eivät ole halukkaita laajentamaan verkkojaan alueille, jossa mahdollisia asiakkaita on vain vähän. Tukiaseman pystytys ja operoiminen maaseuduilla saattaa tuoda enemmän kustannuksia kuin tuloja operaattoreille. Kuha on tukiasemaratkaisu, joka on kehitetty tätä ongelmaa silmällä pitäen. Kuhan tarjoama ratkaisu alentaa käyttöönotto- sekä operoimiskustannuksia, joita syrjäiset asennuspaikat saattavat tuoda. Kuha osittain hyödyntää olemassa olevaa laitteistoa ja ohjelmistoja.

Jotta tällainen epätyypillinen liitettävyys olisi mahdollista saavuttaa, joudutaan eräs tukiaseman ohjelmistokomponentista kirjoittamaan osittain uusiksi. Kun uutta ohjelmistoa luodaan, tulisi se myös testata. Jatkuva integraatio Kuhan tapauksessa on looginen ensimmäinen askel kohti automaatiota, sillä uudelleen kirjoitettu ohjelmistokomponentti on hyvin oleellinen osa koko tukiaseman toiminnallisuutta.

Tämän diplomityön kirjallisuusosuus selvittää matkapuhelinverkkojen kehitystä, sekä minkälaisen ratkaisun Kuha tarjoaa ongelmaan jonka syrjäiset seudut aiheuttavat. Käytännön osuus diplomityössä oli jatkuvan integraation järjestelmän toteuttaminen. Kirjallisuusosuudessa esitellään myös miten erityyppiset ohjelmistotestausmenetelmät sopivat yhteen perinteisten ja modernien ohjelmistokehitysmenetelmien kanssa.

Diplomityössä kehitetty jatkuvan integraation järjestelmä on ollut projektin käytössä projektin alusta asti. Järjestelmä tulee olemaan myös käytössä, sekä jatkuvassa kehityksessä tämän diplomityön jälkeenkin. Tulevaisuuden kehityssuuntana järjestelmää tullaan kehittämään kohti jatkuvaa toimitusta tai jatkuvaa käyttöönottoa.

PREFACE

This thesis was written between the fall of 2017 and spring of 2018. At the same time, I was also developing the software that this continuous integration system ultimately tests. This meant that I was sometimes too concentrated in the software development, rather than in the thesis itself.

I would like to thank Petri Ervasti for giving me the opportunity to do this thesis. I would also like to thank my colleagues for sharing me their infinite knowledge and wisdom.

Vantaa, 18.5.2018

Sami Näppä

CONTENTS

1.	INTRODUCTION	1
2.	LTE BASE STATION SOLUTION FOR RURAL AREAS.....	2
2.1	From 3G to 4G and to the upcoming 5G.....	2
2.2	Narrowband-IoT.....	5
2.3	Kuha base station solution.....	7
2.4	Flexi Zone Micro.....	10
3.	TESTING AND PRACTICES IN SOFTWARE DEVELOPMENT	11
3.1	Agile methodologies	11
3.2	Software testing.....	13
3.3	Continuous integration	15
3.4	Gitlab.....	17
3.4.1	Pipelines	17
3.4.2	Merge requests	19
3.4.3	Issue tracking	20
3.4.4	Gitlab runners.....	20
4.	CONTINUOUS INTEGRATION IN KUHA.....	22
4.1	Test environment.....	22
4.1.1	Base station	23
4.1.2	Power distribution unit.....	25
4.1.3	User equipment	26
4.1.4	Smokebox.....	27
4.1.5	OpenStack cloud instances.....	28
4.2	Continuous integration pipeline	29
4.2.1	Compilation.....	29
4.2.2	Test stage.....	31
4.2.3	Baseline knife request creation	32
4.3	Integration tests	33
4.3.1	Robot framework	33
4.3.2	Test scenarios.....	35
4.3.3	Software architecture in integration tests.....	37
4.4	Supporting software for the continuous integration system.....	40
4.4.1	WFT sniffer.....	40
4.4.2	Baseline listing.....	42
4.5	Further development of the system	43
5.	CONCLUSIONS.....	46

LIST OF FIGURES

Figure 1.	<i>LTE network architecture. Copied from [1].....</i>	<i>3</i>
Figure 2.	<i>LTE coverage of DNA's network in Lapland.....</i>	<i>4</i>
Figure 3.	<i>Different NB-IoT carrier deployments. Copied from [7].....</i>	<i>6</i>
Figure 4.	<i>Simple instructions are provided with the Kuha base station.....</i>	<i>8</i>
Figure 5.	<i>Kuha solution architecture.....</i>	<i>9</i>
Figure 6.	<i>Waterfall model. Copied from [9].....</i>	<i>11</i>
Figure 7.	<i>Sprint process. Copied from [11].....</i>	<i>12</i>
Figure 8.	<i>Burn-down chart.....</i>	<i>13</i>
Figure 9.	<i>Black-box and white-box testing. Copied from [16].....</i>	<i>14</i>
Figure 10.	<i>Continuous integration workflow. Copied from [19].....</i>	<i>16</i>
Figure 11.	<i>Graphical interface illustrating continuous integration output. Copied from [22].....</i>	<i>17</i>
Figure 12.	<i>Overall view of the pipelines. Modified from [22].....</i>	<i>18</i>
Figure 13.	<i>One possible workflow when utilizing merge requests.....</i>	<i>20</i>
Figure 14.	<i>Test environment connectivity diagram.....</i>	<i>22</i>
Figure 15.	<i>Contents of the radio frequency box.....</i>	<i>23</i>
Figure 16.	<i>Flexi Zone Micro in the laboratory environment.....</i>	<i>24</i>
Figure 17.	<i>SSH interface of the PDU showing the statuses of the sockets.....</i>	<i>25</i>
Figure 18.	<i>Power distribution unit.....</i>	<i>26</i>
Figure 19.	<i>User equipment.....</i>	<i>27</i>
Figure 20.	<i>OpenStack graphical user interface displaying created instances.....</i>	<i>28</i>
Figure 21.	<i>Overall picture of the build pipeline.....</i>	<i>29</i>
Figure 22.	<i>Comparison between traditional virtual machines and Docker containers. Modified from [26].....</i>	<i>30</i>
Figure 23.	<i>Modular nature of the Robot framework. Copied from [29].....</i>	<i>33</i>
Figure 24.	<i>Log created by Robot framework.....</i>	<i>37</i>
Figure 25.	<i>Software architecture in testing environment.....</i>	<i>38</i>
Figure 26.	<i>WFT sniffer activity diagram.....</i>	<i>41</i>
Figure 27.	<i>Screenshot from the scheduled WFT sniffer executions.....</i>	<i>42</i>
Figure 28.	<i>Website listing all the knife baselines.....</i>	<i>42</i>
Figure 29.	<i>Baseline listing software deployments.....</i>	<i>43</i>
Figure 30.	<i>Radiators.....</i>	<i>44</i>

LIST OF SYMBOLS AND ABBREVIATIONS

ADB	Android Debug Bridge
ATDD	Acceptance Test-Driven Development
BPSK	Binary Phase Shift Keying
CAPEX	Capital Expenditure
CI	Continuous Integration
eNB	eNodeB, Evolved Node B
EPC	Evolved Packet Core
FDD	Frequency-Division Duplex
FZM	Flexi Zone Micro
GPRS	General Packet Radio Service
HetNet	Heterogenous Network
IoT	Internet of Things
LAN	Local Area Network
LTE	Long Term Evolution
MIMO	Multiple-Input and Multiple-Output
NB-IoT	Narrowband-IoT
OFDM	Orthogonal Frequency Division Multiplexing
OPEX	Operational Expenses
OSS	Operations support system
PDU	Power Distribution Unit
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RF	Radio Frequency
SMS	Short Message Service
SUT	System Under Test
TDD	Time-Division Duplex
UE	User Equipment
VoLTE	Voice over LTE
WFT	Workflow Tool
WLAN	Wireless LAN

1. INTRODUCTION

The continuous integration system introduced in this thesis was implemented during the software development of a rural base station solution. The aim for this continuous integration system was to support the development of the new software created for the pre-existing base station solution. Since the base station hardware already exists, the testing part of the continuous integration would not be testing all the parts of the base station because it has already undergone testing during development. Therefore, the integration tests would be concentrating more on the use cases that this new base station solution provides.

Kuha is the name for a base station solution that enables rural communities and households to have LTE mobile coverage. Kuha base stations are designed to be easy to setup without prior telecommunications experience. This offers a way for the mobile operators to decrease the costs of setting up new mobile base station sites since the base station would be shipped to customer, who would then perform the base station setup. Rural locations have previously been challenging for mobile operators since the costs of running a new base station exceed the benefits that it brings.

This thesis consists of a theoretical part which dives into the theory of testing in software projects which supports the practical testing environment that was established. Also, very brief introduction to mobile network technology and how that affected the motivation behind Kuha base station solution.

This thesis is constructive and the goal for this thesis is to create a continuous integration system by utilizing the Gitlab's built-in continuous integration functionality. The goal for this continuous integration system is to be utilized to test the Kuha base station solution. This system should be highly automated and be easily extended into even further automation, such as continuous delivery or continuous deployment. As a part of the continuous integration system, also integration test cases will be created. The goal for this continuous integration system is that it provides actual value to the software development team.

Chapter 2 will be introducing the evolution of mobile networks and how the solution called Kuha fits into that evolution. Chapter 3 will be discussing about the software development methodologies as well as how the development practices such as continuous integration can be used in cooperation with those methodologies. Chapter 4 will then introduce the continuous integration system that was developed as the practical part of this thesis.

2. LTE BASE STATION SOLUTION FOR RURAL AREAS

This chapter will be discussing briefly the evolution of mobile networks and some of the obstacles that have been introduced by this evolution. Section 2.3 introduces a mobile network solution called Kuha and how Kuha is planning to overcome some of these obstacles. While the business solution and software behind Kuha is discussed in Section 2.3, the hardware side will be discussed in Section 2.4.

2.1 From 3G to 4G and to the upcoming 5G

The high usage of data transfer has been the driving factor for the evolution from 3G based UMTS networks to 4G-based LTE networks while the driving factor in the previous generations was the rising usage of voice calls and short message service (SMS). The amount of data transferred in LTE-based mobile networks has been increasing year over year, being almost tripled within year 2009. This type of usage switch from voice to data traffic has made new kind of requirements for the mobile operators, forcing them to spend a considerable amount of their revenue to increase the capability of their networks. From the network planning point of view, adopting the LTE networks meant that some necessary precautions had to be made since LTE did not provide circuit-switched voice capabilities like the previous generations used to provide. At first this meant that for voice calls, the 2G and 3G networks had to be utilized, although LTE provides a voice over LTE (VoLTE) capability [1]. Nowadays VoLTE has been launched in more than 125 countries and was expected to exceed 650 million VoLTE-ready subscribers by the end of 2017 while LTE subscriptions are estimated to be around 2,6 billion. It has been estimated that around the year 2023 the VoLTE-ready subscription amount would match the LTE subscription amount [2].

Moving away from circuit-switched to packet-switched network provided advantages in terms of end user applications not requiring to be used in a specific network. One common use case would be to provide smooth, continuous usage of the application when user was moving between cellular network and Wi-Fi. Common trend in the mobile phone development has been to support this kind of uninterruptible network switch. Since upgrading into a new network generation requires a significant amount of time in terms of the spread of new mobile devices supporting the new generation as well as the installation of the new base stations, the networks are deployed in parallel. This means that multiple network generations are deployed into the same locations as the existing ones. While LTE provides theoretical peak data rates from 100 Mbit/s up to 326,4 Mbit/s, the actual data rates are usually lower due to the limitations in the core network transport as well as the spectrum. [1].

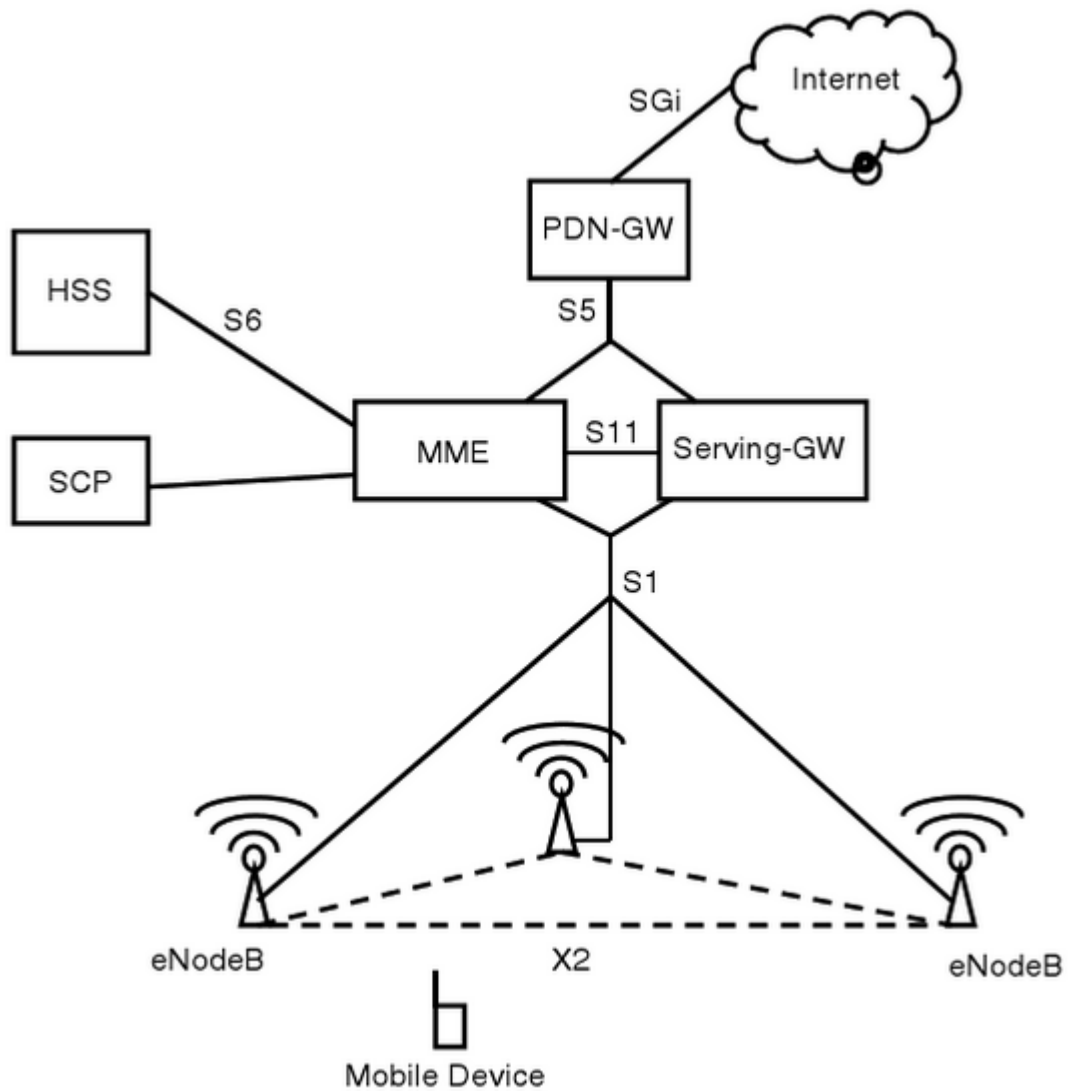


Figure 1. LTE network architecture. Copied from [1]

Figure 1 illustrates the LTE network architecture. The term used for base station in LTE is eNodeB (eNB). The S1 link connecting the base stations to the core network is usually established via an Ethernet. Depending on the use case, the S1 traffic can be transported via either a dedicated link or via public Internet. In highly populated areas, such as cities, the cells are distributed with a site distance between 500m and 2km. As the demand rises, operators may decrease the cell size and add additional hardware into the area in order to continue providing highspeed connectivity. It is also possible for the operators to deploy additional frequency bands within one cell, if they hold a license to radiate in such frequencies. Higher frequency bands are used in high density, low coverage areas to provide higher speeds and lower frequencies in rural areas to provide coverage. To further increase the capacity, temporary sites can also be used to handle cases where high demand is only needed for limited time, such as in case of sports events [1]. The increasing demand for data traffic in the modern cellular networks has been the factor for developing heterogenous networks (HetNets). Traditionally the network architecture was designed by relying on several full-scale base stations, called macro cells, that were deployed to

such locations where they would serve proportional number of customers. In HetNet, small base stations, called small cells, are used to complement traditional macro cells by extending the coverage to places where the network signal used to be weak, such as indoors. Small cells also provide lower transmit powers from as low as 100mW to 2W while the macro cells were anything between 5W. and 50W [3]. Small cell deployment introduces new problems to network operators since deploying high amounts of base stations into a single location may cause interference between the base stations. Traditionally, the network planning in cellular networks has been done by using problem formulations including coverage planning, power optimization and channel assignment. However, these planning models are not suitable for HetNets due to their focus on traditional cellular network design as they only considered the placement of the new macro cells. Due to this, new models for cellular network planning had to implemented, where also the possible inter-cell interference had to be considered [4].

From a financial point of view, the operators need to cover the costs of acquiring licenses for spectrum and for buying and installing base station and the infrastructure. These kinds of expenses are called capital expenditure (CAPEX). Running and maintaining a network also creates expenses that are called operational expenses (OPEX), which are rental costs for installation sites and power used by the base stations. In city like environment where the potential user amount is high, acquiring additional infrastructure to provide connectivity is easily justified. However, in rural areas, fast Internet access either via a fixed line or wireless connection is still rare in certain regions. The low number of potential customers in such areas is making operator reluctant to make investments.

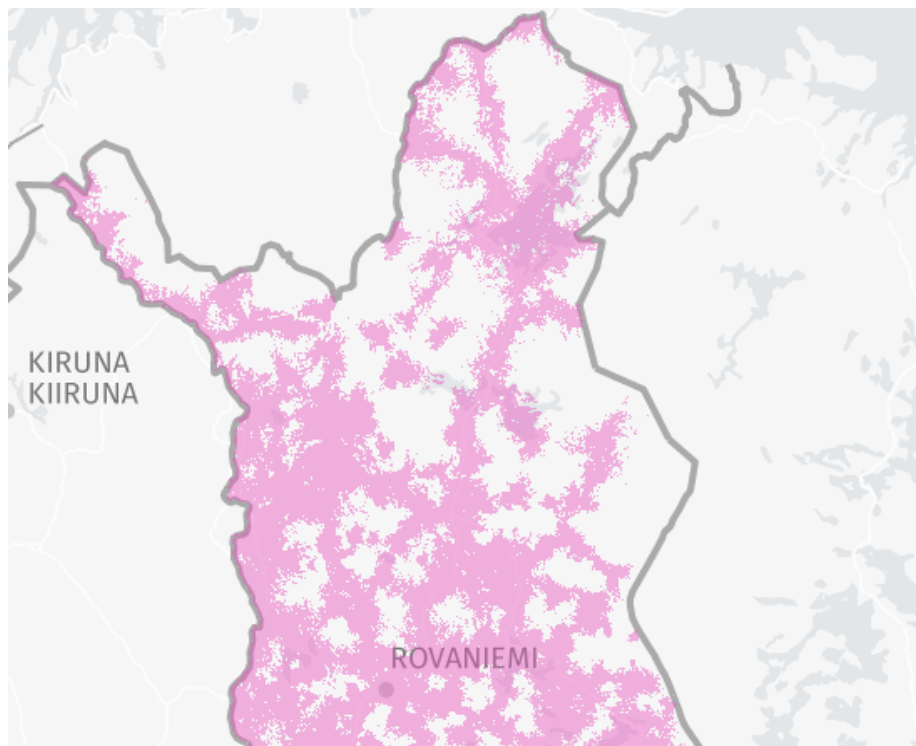


Figure 2. LTE coverage of DNA's network in Lapland

In Figure 2 a screenshot is taken from coverage tool that Finnish mobile operator DNA has provided. In the screenshot, the area filled with pink color represents an area where there is LTE coverage. Ericsson has estimated that a base station covering an area of 12 square kilometers is financially viable in areas with population densities as low as 15 people per square kilometers [1].

At the same time as new innovations and products are made, also new requirements regarding wireless connectivity are brought up by them. Moving on from LTE to 5G means that these requirements are taken into account during the design process. While LTE was designed to provide highspeed Internet access, the new products might not have the need for highspeed data transfer since they might only transmit low amounts of data occasionally. Often these same devices might also work on battery, meaning that the power used for wireless transmission must be kept at minimum. Internet of things (IoT) is one prominent area where this is a fact that has to be considered during the design process of the device. IoT will be discussed more in depth in the following section. 5G considers these highly varying requirements that the ever-evolving subscriber base has. 5G provides support for the low latency transmission that, for example, the machine-to-machine communication requires as well as the power-efficient, high latency simple data transmission that the IoT devices require. In order for the 5G to achieve even faster data transmission than what LTE provides, peak data rates being 10-20Gbps, certain measures have to be done. One of these measures is to use spectrum that is above 3GHz. Typically, LTE uses spectrum that is well below 3GHz and bandwidth of 20MHz in case carrier aggregation is not used. In 5G, the spectrum might go as high as 100GHz while the bandwidth is several hundred megahertz. Very high frequency is also a limiting factor in terms of cell coverage since the signal attenuation is very high, so the transmission distances will be shorter. Another prominent feature is beamforming, where the signal energy is directed to the direction where the receiving device is. Beamforming is a way to compensate for the signal attenuation that happens in the higher frequencies, thus allowing the range of the signal to be extended [1].

2.2 Narrowband-IoT

Internet of things (IoT) is a term used for describing a network where previously Internet-incapable everyday objects or devices, “things”, are now connected to the Internet. These devices would have capability to sense and actuate the surrounding environment to some extent. Also, some of these devices contain programmability capabilities. The idea behind the device interconnectivity would be to provide cooperation between different systems, hence providing new services for the customers. Studies have revealed that by the year of 2020, more than 30 billion devices would be connected to the Internet. Despite the amount of data sent by one IoT device is not high, due to the fact there are millions of these devices, the amounts of data generated will be substantial [5]. In the US, grain farms are moving towards increased automation where drones would be utilized to monitor crops,

informing the farmer about possible pest infestation. In the ground level, the devices monitor humidity and trigger watering if required [6].

During the design process of the IoT device, one of the most prominent design questions is how to connect the device to the Internet. Depending on the operational environment of the product, the connection can be either wired or wireless. Connectivity method is even more critical in the case device is operating battery-powered, since the power used for data transmission will be one of the factor to consider when maximizing the lifetime of the device. If the device is located indoors, for example in shopping malls or apartments, it can utilize the existing wireless LAN (WLAN). Other possibility is to utilize the cellular network in such places where the WLAN coverage is not provided such as outdoors. For low data rates and low hardware costs, General Packet Radio Service (GPRS) could be utilized, but operators are already upgrading their networks towards LTE. Connecting an IoT device to LTE network introduces new problems. The signaling between the device and the network can be very power-hungry for IoT devices operating on battery. To address this issue, the 3GPP standards group has developed the Narrowband IoT standard (NB-IoT) [6].

LTE uses Orthogonal Frequency Division Multiplexing (OFDM) as the modulation technique, where the data is transmitted by using multiple 180kHz wide narrowbands. Typically, the bandwidth for the LTE cell is something between 1,25 and 20MHz. Using narrowbands provides a way for the LTE to scale up or down depending on the amount of the available spectrum. Scaling is done by increasing or decreasing the amount of used narrowbands. NB-IoT standard describes that one or multiple NB-IoT carriers use these existing narrowbands, but due to the differences in, for example, the modulation, they would be specifically designed for higher power efficiency. Figure 3 illustrates different ways to deploy a NB-IoT carrier within the already used spectrum.

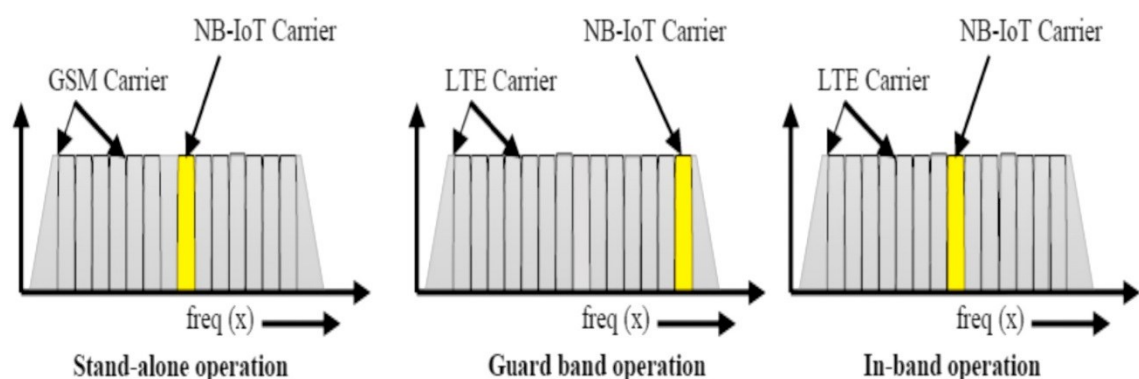


Figure 3. Different NB-IoT carrier deployments. Copied from [7]

NB-IoT standard provides a simplified air interface. It does not support channel measurements and, reporting and the potential raw data rate in practice is only around 200kbit/s. However, for many IoT devices these kind of data rates are usually sufficient. Despite the low throughput of the NB-IoT, it has been designed to serve up to 50 000 devices within

one cell. Network simulations made by 3GPP have also shown that a single NB-IoT channel would be capable of supporting this number of devices in a scenario where devices sent 105 bytes of data per transmission. NB-IoT also uses OFDM, but for modulation it utilizes Binary Phase Shift Keying modulation (BPSK) or Quadrature Phase Shift Keying (QPSK). These modulation techniques can transmit 1 and 2 bits respectively per transmission step compared to the normal narrowbands transmitting up to 8 bits per transmission step with Quadrature Amplitude Modulation (QAM) when the radio conditions are ideal. This lowers the requirements for the radio processing in the devices using NB-IoT channel [1].

2.3 Kuha base station solution

In traditional macro base station-based approach, the base station hardware is only part of the costs. New base station sites may require investments in the infrastructure, since the base stations require electricity and Internet connectivity. Macro base stations are also often installed at high grounds in order to maximize the coverage. In addition of the investments mentioned, installing the base station itself requires a visit of a telecommunications engineer which, in case of a rural site, may also be a significant cost.

Kuha is a name for a rural base station solution that is designed to solve the rural connectivity related problems that were mentioned in Sections 2.1 and 2.2 and to provide an easy method to extend LTE coverage. Kuha aims to provide a plug-and-play type of solution that allows base stations to be installed by anyone even without any prior telecommunications experience. The requirements for the installation are designed to be as simple as possible, only requiring simple set of tools for mounting the base station itself in addition of power and Internet connectivity. If the location does not provide a wired connection, Kuha can also operate over a satellite backhaul. Flexibility for the power input is also considered and, for example, solar panels can be considered in areas that provide enough electricity via solar power. Kuha base stations operate on operators' frequency which means that the Kuha base stations are an extension to the already established cellular network.

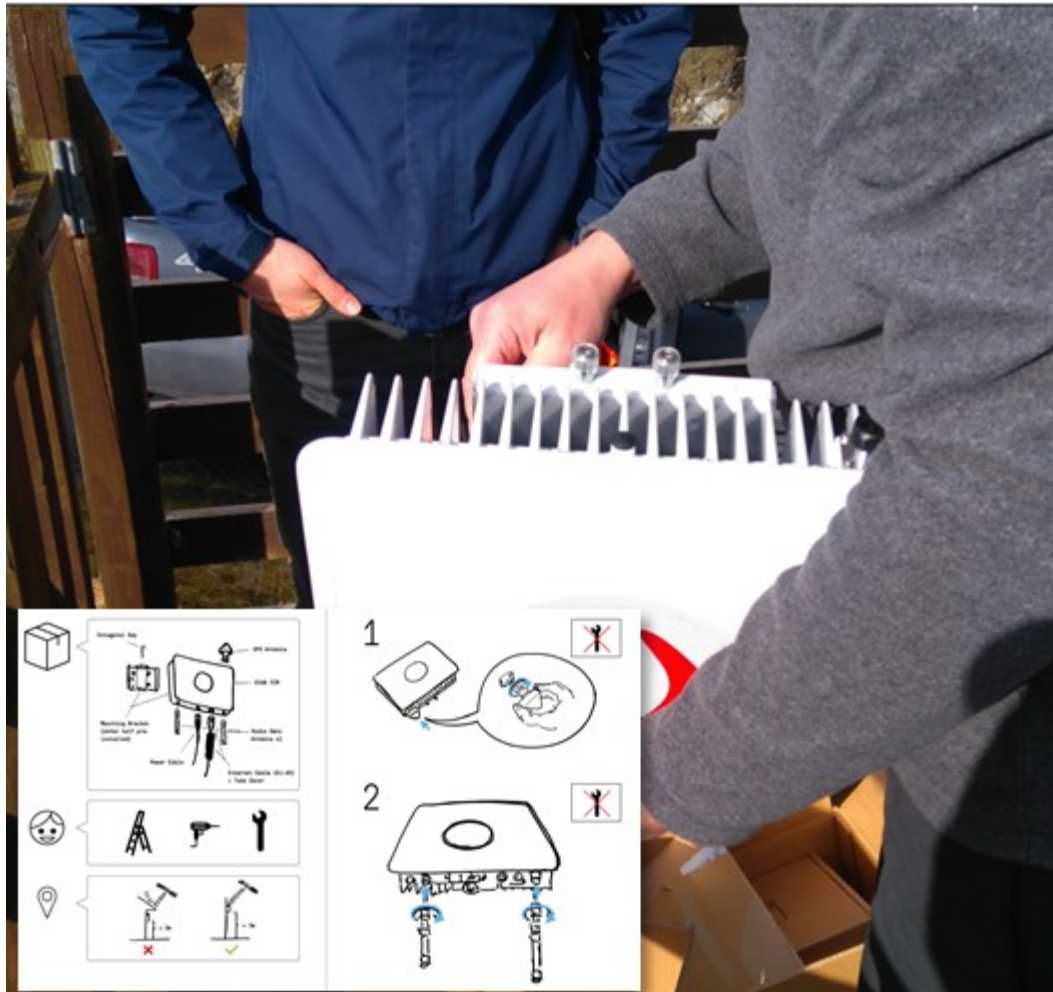


Figure 4. Simple instructions are provided with the Kuha base station

Kuha solution provides simple installation instructions that can be seen in Figure 4. This allows rural communities and hard-to-reach areas to gain LTE connectivity. The community maintains the base station locally in addition to the Kuha team's remote observing. Kuha Cloud OSS (operations support system) is a management system that is developed to maintain the Kuha base stations as well as update their software remotely. Kuha OSS collects certain type of data from the base stations that are installed in the field. This data is used to deduce the status of the Kuha base stations and it contains information such as base station's internal alarms, counters, and heartbeat. This data is processed and illustrated in an administrator view that lists all the devices and provides an interface to do certain type of maintainability actions on malfunctioning devices. In the case a problem that cannot be handled remotely has been noticed, the local maintainer can be instructed to take relevant actions. Kuha type of approach allows service providers to sell LTE coverage in places where it has not been profitable before. Figure 5 below illustrates the connectivity diagram of the Kuha base station solution.

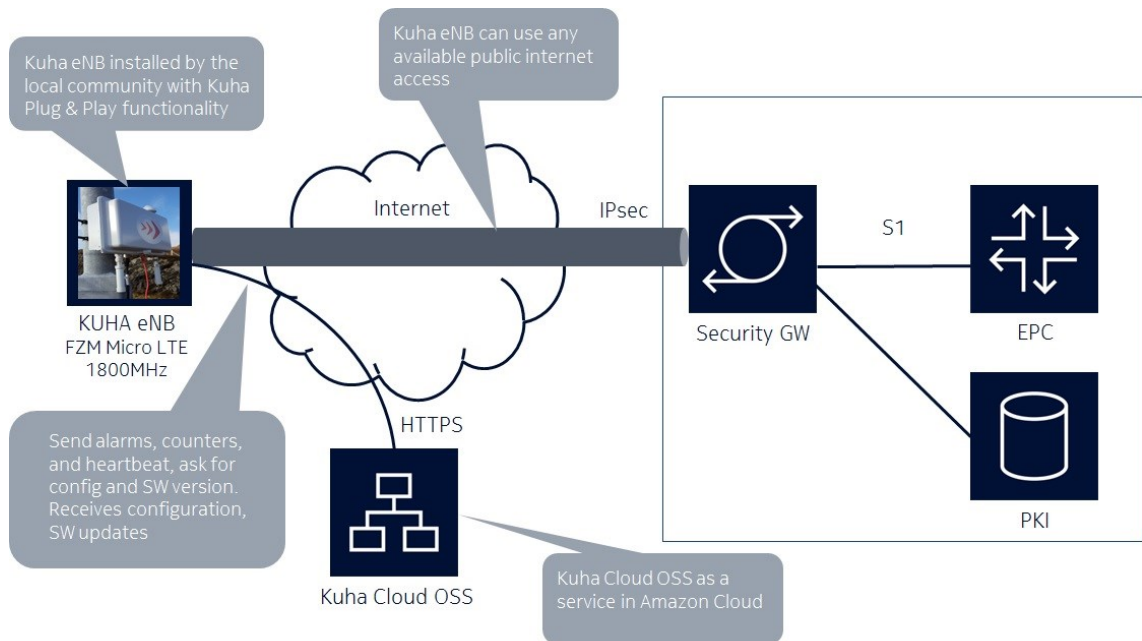


Figure 5. *Kuha solution architecture*

Secure connectivity between the base stations and the operator's core network is achieved by IPsec, which is a security protocol for providing security to IP and upper layer protocols such as UDP and TCP. IPsec is capable of protecting packets between hosts, between network security gateways or between hosts and security gateways. Security gateway is usually a router that supports IPsec or a firewall [8]. In Figure 5 the connection is between a host and a security gateway.

Kuha base station solution utilizes an existing software and hardware. Hardware is discussed in detail in Section 2.4 and Chapter 4. The software handling the LTE protocol and the air interface will be left untouched. However, in order to achieve the plug-and-play type of installation and non-traditional backhaul connectivity, a set of software called transport inside the base station needs to be reimplemented to some extent, while certain parts of it will be kept as is. Transport software is a term used in this thesis to describe a set of software that handles, for example, the IPsec connectivity handling between the base station and the security gateway and the base station's network interface configuration.

Kuha project will already contain two separate versions of the transport, versions 1 and 2. Version 1 utilizes the existing transport software with some modifications made into it in order to create a working device. Version 2 is more ambitious approach, where large parts of the transport software will be rewritten. The goals for the reimplementation are to speed up the reboot time of the base station and to enable more extensive management actions from the Kuha OSS. This thesis concentrates on the transport version 2 development and testing. The old transport implementation also contains design decisions and features that are not needed within the scope of the Kuha. The continuous integration

system and the integration tests described in this thesis were designed to verify that the changes made to the transport software do not break the use cases that the legacy transport software provided. Some of the integration tests also include certain type of requirements that were designed specifically to ensure that the intended improvements regarding for example the reboot times were achieved.

2.4 Flexi Zone Micro

Flexi Zone Micro (FZM) is a base station that provides small cell solution. FZM is optimized for outdoor environment. FZM has multiple hardware variants that provide support for different radio technologies such as frequency-division duplex (FDD) as well as time-division duplex (TDD). FZM supports a bandwidth of up to 20MHz with varying transmit power as well as Multiple-Input and Multiple-Output (MIMO) method to increase the radio link capacity. The term “Kuha base station” in this thesis refers to a FZM although Kuha OSS solution also provides support for any kind of base station hardware.

3. TESTING AND PRACTICES IN SOFTWARE DEVELOPMENT

This chapter takes a look at how the software development methodologies have developed throughout the years. Section 3.1 briefly discusses the software models called waterfall model and how it compares to agile methodology. Section 3.2 will introduce how the software testing is established in waterfall and agile methodologies and what benefits the agile methodology can bring over waterfall model. Section 3.3 introduces a software development practice called continuous integration and how that software development projects can benefit from using it. Final section will introduce Gitlab, a version control system that also provides functionality that can be used to implement continuous integration into software development project.

3.1 Agile methodologies

Historically, software projects used to rely on waterfall development practice in which a dedicated integration phase was executed usually after the development phase. In the integration phase, the software changes made by the teams of developers were merged together with the goal of outputting a working product. Figure 6 illustrates the steps of waterfall model. In this figure, each of the boxes represents a phase of the product and they are executed in order from top to down, hence the name waterfall model. Once a phase is completed, the project moves on to next phase and does not go back. The output of the previous phase is used as an input of the following phase. This means that any modifications made to the already completed phases would make the following phases unstable. [9]

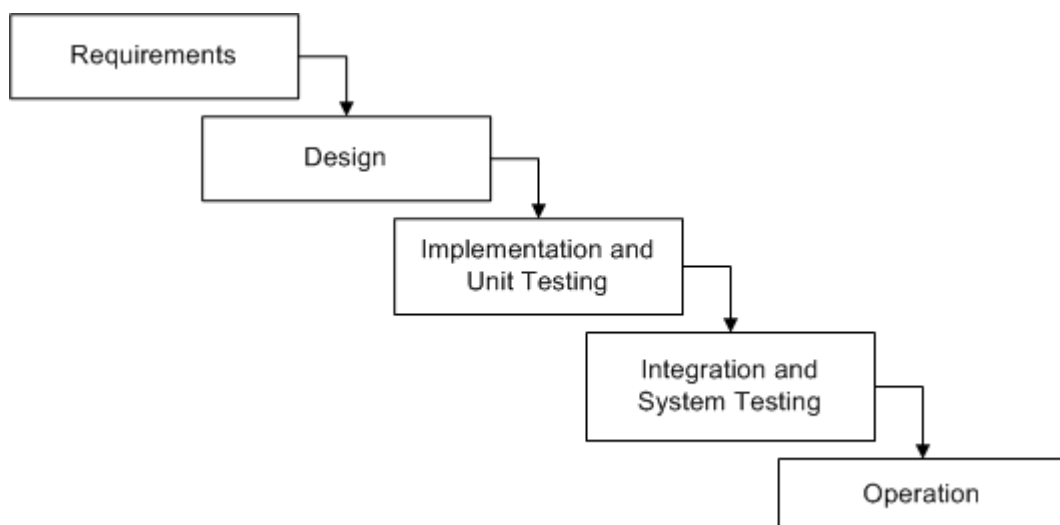


Figure 6. Waterfall model. Copied from [9]

Due to the obvious shortcomings of the waterfall model, a new software development concept was created. This new concept, agile methodology, have been used since the beginning of the 21st century. In agile methodology as well as in the waterfall model, the customer was included into the very early phase of the project. What is done differently in agile compared to the waterfall model, the customer is being kept in the loop throughout the project. In waterfall model, the customer requirements were frozen after the requirements phase, leaving developers to create a product just with that information. During the implementation, the market, or the environment for which the project was created for may have changed significantly. This means that once the product is ready, it may already be obsolete and worthless for the customer. In agile, the implementation is made in iterative fashion, keeping the customer in the process throughout the project. Having constant feedback from the customer allows the project to adapt accordingly to the world changing around it. One iteration in agile is called a sprint and it is usually anything between 1 to 4 weeks. During the sprint, the team selects tasks from the backlog and implements them. After the sprint, customer may review the changes made and influence on the tasks done in the following sprints [10]. Figure 7 illustrates the process behind the term sprint in one of the agile methodologies called Scrum.



Figure 7. Sprint process. Copied from [11]

In Scrum the product backlog is a collection of the functional and non-functional requirements. These requirements undergo a process called grooming in which user stories are created. A user story is a basic unit of work in Scrum and usually the goal is that each user story can be completed within one sprint. This means that a single requirement can be translated into multiple user stories. In Scrum, the mentality regarding software bugs is that a user story should not introduce any new bugs into the product. In case the user story does introduce new bugs, the user story is not considered done until the newly introduced bugs are resolved. User stories also usually hold an estimate, called story point, that is used to display how much a user story takes time to complete. After multiple sprints, the team may then use these estimates when they are planning how much user

stories they take into the sprint backlog in the beginning of each sprint. Certain types of charts called burn-down and burn-up are used to display the progress as well as the work needed to complete the project. Figure 8 illustrates how a burn-down chart could look like. Usually burn-down charts have two lines where one is illustrating how the linear progress would go and the other illustrating how the actual work is done. [12]

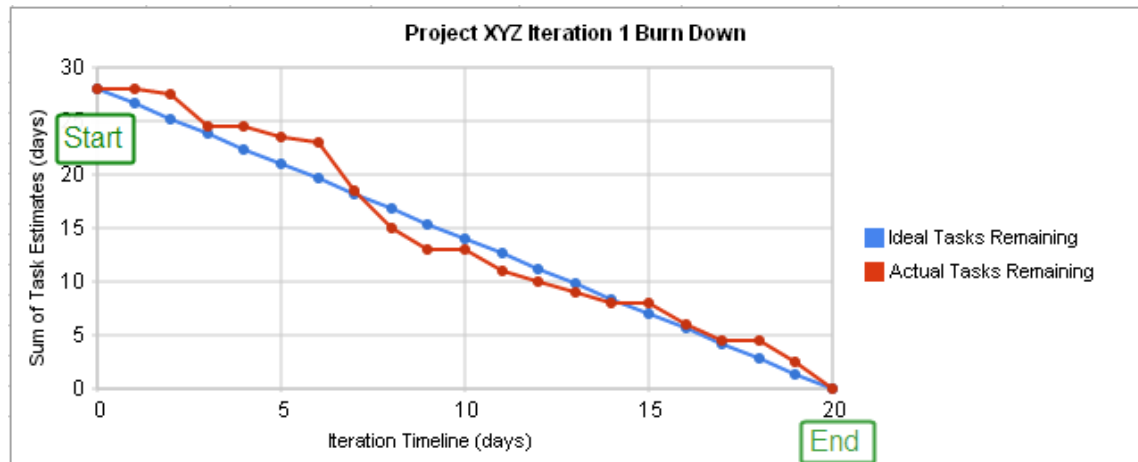


Figure 8. Burn-down chart

In the beginning of the sprint, a sprint planning meeting is held in which the team agrees on what user stories they will include into the sprint backlog. Usually the decision is made using the input given by the customer on what is important to have next. During the sprint, a short meeting, called daily, will be held in which the team members update each other on the progress of their user stories. The goal is to keep the daily meetings very short time-wise, usually only 10 to 20 minutes long. One very common way to keep the Scrum daily meeting short is to keep it as a standing meeting. During the daily meeting, team members may address any problems or risks they have noticed during the sprint regarding their own user stories or the project as a whole. After each sprint in Scrum, a sprint review and sprint retrospective meetings will be held. Sprint review meeting is used to introduce the newly made increments to the stakeholders who can then accept or reject the changes. Sprint retrospective is a team's internal meeting held to inspect the areas that need improvement but also to identify the practices that brought value to the sprint process. Value increasing practices can be identified by studying if they improved a developer's efficiency or not. [13]

3.2 Software testing

As it was introduced in the previous section, in waterfall model the integration and system testing were an independent step and was started after the implementation of the code was already done. If waterfall model caused problems from in terms of budget and schedule being exceeded, it also caused problems in terms of testing if the development team had to take one or multiple steps back. Going back steps in waterfall model also meant that

testing had to be redone. From testing point of view also the quality, completeness and stability of the requirements had to be very high since any defects found in the integration or system testing would possibly cause delays in the delivery schedule. Delivering the new software into testers in the very late part of the project would also mean that testers were on the critical path of delivery of the software. If there were any delays during requirement, design or implementation, and the delivery date is non-flexible, that would cause the time dedicated for testing to be very short [14].

The absolute goal for software testing would be to test every possible permutation that the software might have. However, in real world this is not possible since even the simplest programs can have millions of possible input and output combinations. Even if the testing is not able to test all the possible combinations, testing should still bring value into the software development. This means that the testing should start with the assumption that the software contains errors and testing is used to find these problems. Testing can be roughly divided into two testing strategies, black-box testing and white-box testing [15]. Figure 9 illustrates the differences between these testing strategies.

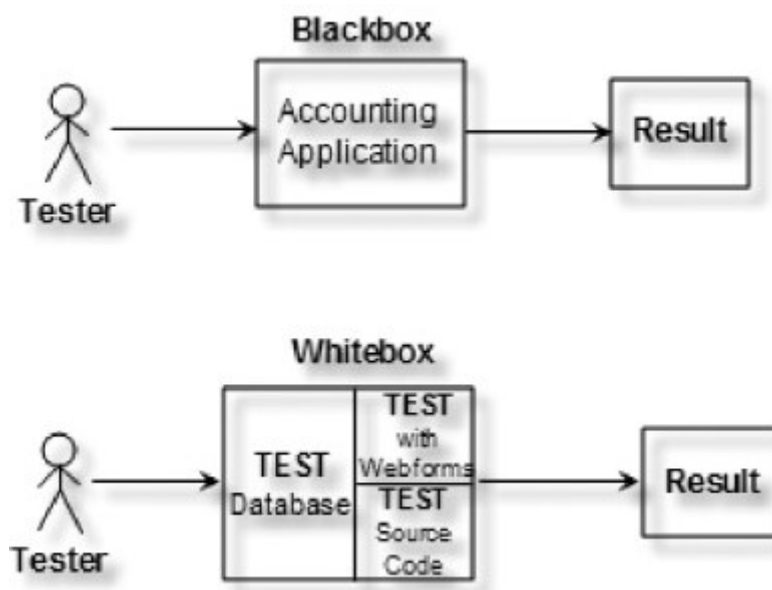


Figure 9. Black-box and white-box testing. Copied from [16]

Black-box testing is a term used for testing that sees the program as a complete program, hence the name. In black-box testing, the test cases are not concerned about the internal workings of the software, instead only the scenarios where the program does not behave as expected. In black-box testing the amount of possible input-output combinations is almost always very high and testing may be very complex when testing programs that have non-volatile memory. Depending on the level in which the test case testing the system is, it might be challenging to find the code block in which the required fix might be in [15]. Integration testing can be considered as one type of black-box testing. In integration testing, the complete system is built by adding modules to-be-tested into an existing

machine containing rest of the software. Tests are then executed against this freshly integrated module. Integration testing should take place in an environment that resembles the final production environment. After the integration tests are executed, acceptance tests can be done to verify that the system fulfils the original requirements [17].

White-box testing is the opposite from black-box testing. In white-box testing, the code to be tested can be freely inspected and the test cases can be written in code unit level. This means that one of the goals could be to reach high test coverage. This kind of requirement is also rather challenging to fulfil since the number of possible paths that the program might have will be very high. One type of white-box testing is unit testing and as the name implies, it tests the code as units such as input and output combinations of functions. Executing every possible permutation of a single function may bring full coverage in the testing report, but in some cases, it might hide the logical faults that the function might have. [15]

In software development, a software risk analysis can be done to ease up the testing process. This analysis defines what to test, priority and the depth of testing. This can help identify areas that are critical and should be tested thoroughly. Software risk analysis should be done at a very early phase of the project, once the high-level requirements are clear. When the test implementation then begins, the developers and testers can refer to this analysis when deciding what type of test cases should be implemented and in which order. Test implementation in this context refers to a process in which test data is acquired, test procedures are developed, and test environment is established. Testing in the software development contains multiple different levels in which the testing takes place. Moving on from unit tests towards integration or system tests, also the testing level moves towards from the code unit to a production-like environment. Production-like environment is important during testing since it may reveal issues within the specification or the code that was not noticed during the requirement gathering or implementation. One good approach for creating the environment that resembles the production is to use the data gathered from the current customers. This is of course not feasible in the scenario where there are thousands of customers with everyone having unique configuration or environment. In this type of scenario, the test environment could be created by gathering samples from the customer base and creating a series of profiles. These profiles would usually contain data that could be used to group the customers. The knowledge gained from this grouping could then be utilized when creating the test environments as this would limit the required amount of variations [14].

3.3 Continuous integration

Waterfall approach introduced in the Section 3.1 introduced conflicts within the software as well as delivery delays and unplanned costs. Continuous integration was developed to overcome the issues presented by the waterfall model [18]. Continuous integration is a software engineering practice in which every commit that a developer makes is verified

by running an automated build and tests. Commit in this context represents a new commit in the version control system. This kind of approach allows the developer to receive almost instant feedback about the quality about the changes that were made in that particular commit. Continuous integration therefore helps developers to discover bugs and errors that might have been created during the creation of the new software or modifying an existing one. Fixing the errors discovered by continuous integration avoids the scenario where errors and bugs build up. Also fixing the errors and bugs in a software that is newly created or modified is easier since every commit gets tested and the amount of the changes that were made are usually manageable. Figure 10 illustrates the continuous integration workflow.

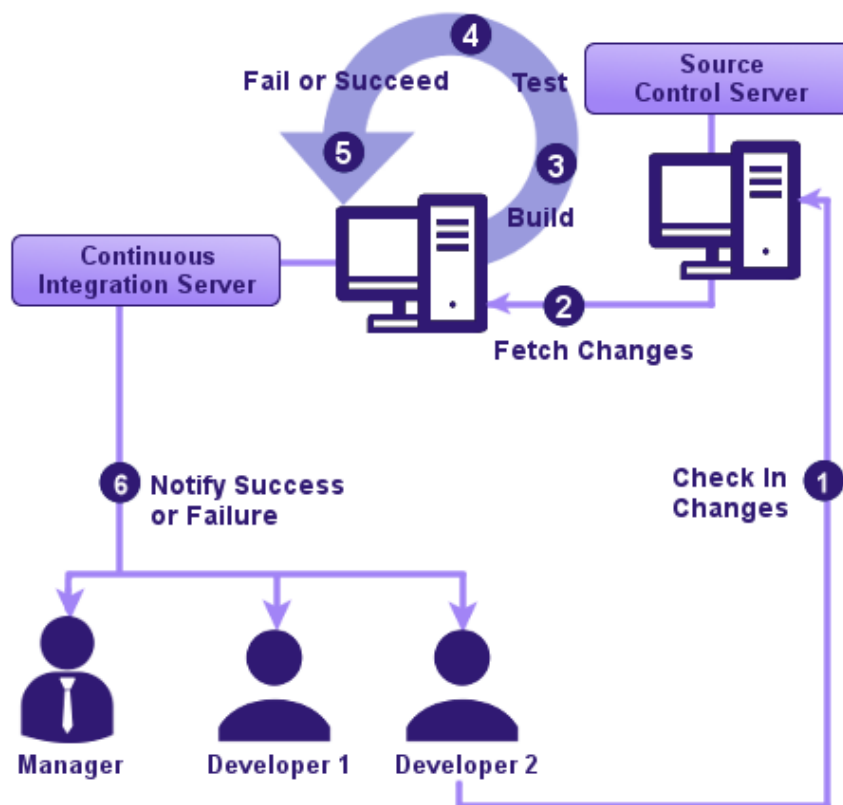


Figure 10. Continuous integration workflow. Copied from [19]

A big part of continuous integration is the version control system that is a shared repository for the software. Nowadays, arguably Git is the de facto standard for the version control system, but also Subversion or CVS can be used [20]. The other big part of continuous integration are the tools that are used to achieve continuous integration environment. Nowadays there exists large variety of continuous integration tools in both open source as well as commercial software. Jenkins and Gitlab CI are some of the tools that can be used in continuous integration [21]. Gitlab is used in the practical part of this thesis and it will be discussed more in depth in Section 3.4 and Chapter 4.

While continuous integration is about risk reduction and helping to fix integration and regression issues faster, it also provides better visibility about the state of the project for both technical and non-technical members. Many continuous integration tools provide some sort of graphical interface that displays information about the automated the automated building and testing and how successful their execution were. An example of such user interface can be seen in Figure 11.

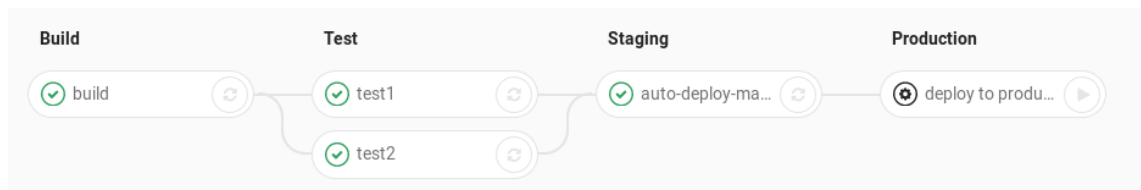


Figure 11. *Graphical interface illustrating continuous integration output. Copied from [22]*

Continuous integration can also open and facilitate communication channels between team members and enforce collaborative problem solving. The automated process provided by continuous integration can also be extended to automatic deployment where the changes that were accepted by the continuous integration process are also deployed forward in the pipeline. This next step in the pipeline can be, for example, a staging environment where developers can observe how the software works before it is deployed to end user.

3.4 Gitlab

Gitlab is an open source Git repository management system written in Ruby and Go programming languages. It provides a web interface that contains tools needed for project work such as documentation through Wiki tools, issue tracking and code review in terms of merge requests. Gitlab also provides user permission management where the developers may have different roles on different repositories and these permissions define for example those who can add new developers to the project [23]. Some of the more prominent features of Gitlab will be discussed in the following chapters.

3.4.1 Pipelines

Pipeline is a term used in Gitlab context to describe a group of jobs that get executed during processing of the pipeline itself. Between the pipeline and a job, Gitlab also has a term called stage. Whenever a stage is run, every job that is inside that stage get executed in parallel. A job can be considered as the basic unit of the Gitlab continuous integration

pipeline and it contains one or more commands that will be executed during the execution of the job itself. Often the steps are Shell commands and Gitlab also examines the return values of every command and if any of the commands return an error value then the execution of the job stops.

Gitlab user interface constructs the user interface so that it is easy for the user to tell the order of the stages that are run as well as the jobs that were run within certain stage. The execution order of the jobs within a stage cannot be predetermined unlike the execution order of the stages. Whenever a job fails, a stage fails also unless it is specifically defined that a job is allowed to fail. Figure 11 in Section 3.3 is illustrating a pipeline. In that figure “Build”, “Test”, “Staging” and “Production” are illustrating the stages within the pipeline. From the figure, it can be also seen that “test1” and “test2” were the jobs that were executed as part of the “Test” stage. Gitlab also provides an overall view of the pipelines that have been run.

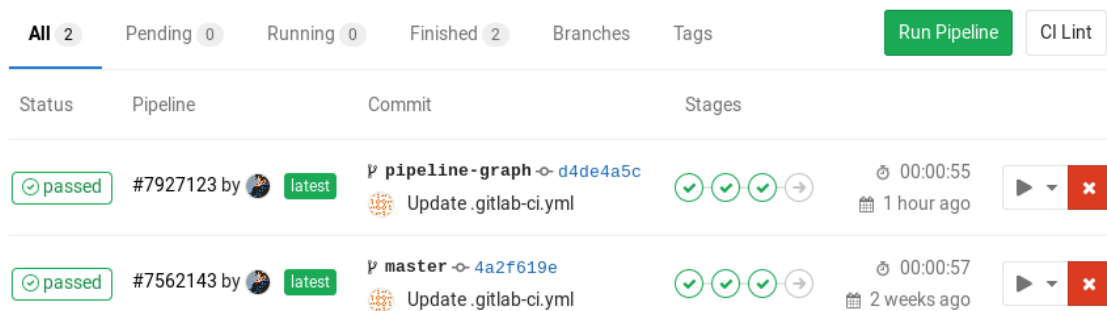


Figure 12. Overall view of the pipelines. Modified from [22]

Figure 12 illustrates an overall view that shows the status of the latest pipelines that were run. The jobs that were run during pipeline may differ depending on certain rules that are defined. For example, a job may contain a rule that defines whether to run the pipeline only on certain branch [22]. In Gitlab, the pipelines are defined by using a YAML-based file that describes the stages and the jobs as well as the steps taken within a job. This configuration file is called `gitlab-ci.yml` and by default Gitlab searches for it from the root of the repository.

```

stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - make build

test_job:
  stage: test
  script:
    - make test

deploy_job:
  stage: deploy
  script:
    - make deploy
  when: manual

```

Program 1. *YAML-based file describing the Gitlab pipeline*

In program 1 above, an example of Gitlab YAML file is introduced. This file defines jobs “build_job”, “test_job” and “deploy_job” as well as stages “build”, “test” and “deploy”. The execution order in this case would follow the order of the stages that were defined in the beginning of the file. As it was mentioned before, jobs within a stage get executed in parallel. This would mean that a new job called, for example, “integration_test” could be added to the “test” stage and it would be executed in parallel with the “test_job”. If either of those jobs were to fail, the test stage would fail.

3.4.2 Merge requests

Merge request is a term used in Gitlab to describe a request to merge new changes into a certain branch. Merge requests are often coupled with Git workflow called feature branching in which the new features are implemented in a new specific feature branch that is branched out of the master branch. The idea behind this kind of approach is to encapsulate the work done by the developer so that it does not disturb the codebase of the master branch, and that the master branch would never contain broken code. Once the feature implementation is completed, a merge requests will be opened. In this new merge request the developer usually describes what the new feature implements, and the code differences will be compared against the master branch. Merge request allow other developers to review the new modifications and comment on them or initiate other kind of discussion related to the implementation. Once the modifications are accepted by other developers, the new feature gets merged into the master branch and the developer can move on to developing new features in a new feature branch [24]. Figure 13 below illustrates one possible workflow that could be utilized when merge requests are used in the software development.

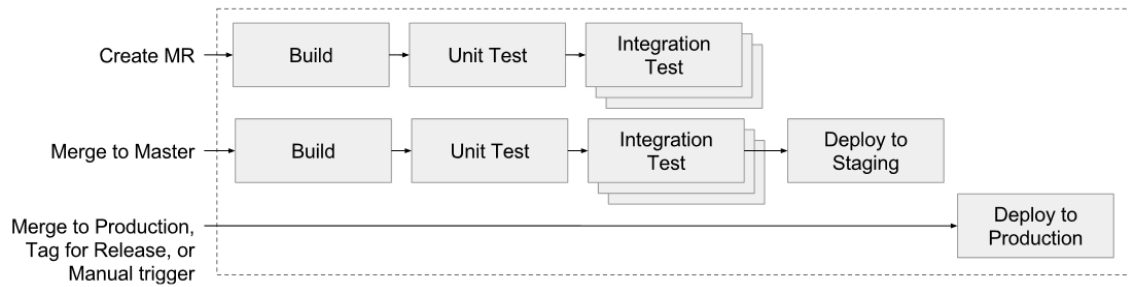


Figure 13. *One possible workflow when utilizing merge requests*

In the case a merge request gets accidentally accepted and merged or after merging problems are found in the master, many of the version control systems allow a possibility to revert the changes that were made. Gitlab provides a possibility to do a revert for the merge requests. Reverting a merge request undoes all the changes that were introduced in the new merge request, so this means that multiple commits will get reverted from the master branch.

Merge requests are used in the continuous integration environment that was developed as a practical part of this thesis. There is not a clear definition on how many approvals does a merge request require before it can be merged to the master. In projects that have many developers, usually at least 2 approvals are required. Build and test pipeline gets executed for all the branches and their results are visible in the merge request itself, and the prerequisites for approval of merge request are an accepted review as well as a successful pipeline.

3.4.3 Issue tracking

Gitlab provides a way to map the relation between the code and the issues via issue tracking. Gitlab's graphical user interface provides a possibility for developers to create new issues and create feature branch for implementing the needed modifications. Issue tracking in Gitlab context provide also a possibility to estimate the amount of work in hours needed for the implementation work. Issues also provide a platform for a further discussion about the possible implementation approach or the validity of the issue. Issues are also linked to the merge requests allowing easy management of between the merge request merging and issue closure.

3.4.4 Gitlab runners

Gitlab Runner is a Go-based binary which handles the communication between the Gitlab server and the integration or the testing machine. Gitlab Runner is run on the machine that is used to execute the jobs defined in `gitlab-ci.yml` configuration file. Once the job is executed, Gitlab Runner also reports the results back into Gitlab server from where the results can be observed using the graphical interface displayed in Figures 11 and 12 on

pages 17 and 18. Gitlab Runner allows jobs to be executed in different kind of execution environments. This configuration is referred to as job executor. Possible executors in Gitlab are:

- Shell
- Docker
- Docker Machine (auto-scaling)
- Parallels
- VirtualBox
- SSH
- Kubernetes.

Different types of executors offer different kind of possibilities for the environment. For example, comparison between Docker and Shell reveals that by default, Docker offers a clean build environment while Shell does not. [25]

4. CONTINUOUS INTEGRATION IN KUHA

This chapter introduces the continuous integration system that was developed as a practical part of this thesis. Section 4.1 introduces the test environment and components in it as well as how do the components interact with each other. Section 4.2 introduces the pipeline that handles software compilation and the integration test execution. In Section 4.3 the integration tests are discussed more in depth from a technical point of view. Section 4.4 will provide detailed view to a set of software that was implemented for utility actions. Last section will take a look into how the future of the system would look like.

4.1 Test environment

The test environment used in transport software testing consists of the base station (Flexi Zone Micro, FZM), user equipment (UE, in the context of this thesis, a Samsung Galaxy S5), a Linux server called Smokebox, a power distribution unit and two ethernet switches that are used for connecting the FZMs and the Smokebox. The continuous integration process itself is initiated by the Gitlab CI and is executed in an OpenStack cloud instance. This instance handles building and test initiation. There is also a supporting cloud instance that handles several utility activities such as providing download links to the compiled software. Figure 14 illustrates the interconnectivity between the physical components of the test environment.

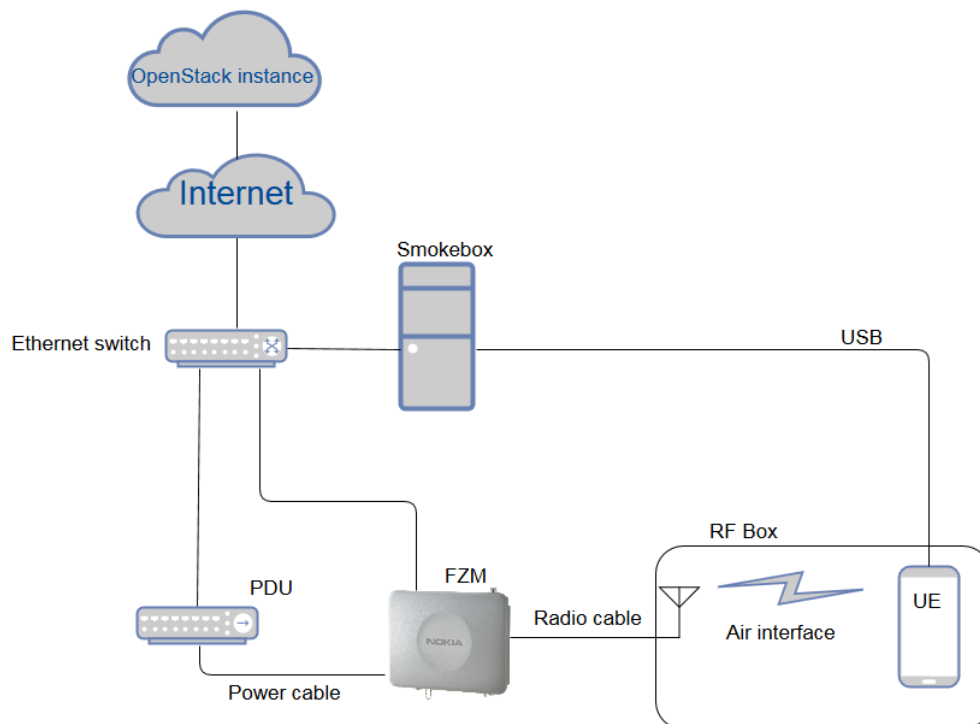


Figure 14. Test environment connectivity diagram

The communication between the UE and the FZM is handled by using the basic LTE air interface. Control and user plane signalling is achieved with a simulated evolved packet core (EPC) that handles the basic core network functionalities such as the UE attach handling and UE data packet routing. The air interface between the FZM and the UE is secured by having a separated radio frequency box in which the UE resides. Inside the radio box there are connectors for USB as well as for the radio antennas. FZM is connected to the radio frequency box via a radio cable. This radio frequency box works as a Faraday cage filtering out the incoming or outgoing signals that may be present in the laboratory environment. This type of setup ensures that the test cases that are testing the UE connectivity, test it against the correct base station. Also, as an extra measure the UE is locked to the LTE frequency band in which the FZM is transmitting in. Figure 15 is a picture taken of the insides of the radio box.



Figure 15. *Contents of the radio frequency box*

All the radio connectors in the FZMs are always attenuated, even if the FZM is used by a radio frequency box. If the FZM is not used by any user equipment, then also a certain type of radio frequency terminator is used after the attenuator to minimize the amount of radio waves radiated. FZM that is used in the test environment are configured so that they use the transmit power of 24 dBm. As it can be seen from the Figure 19 in Subsection 4.1.3 on page 27, after attenuation and the small air interface the received signal strength for the UE residing in radio frequency box is -78dBm.

4.1.1 Base station

Base station is an essential part of the testing environment setup in the continuous integration pipeline. In this testing environment the base station in use is FZM. Compiled

transport software gets packaged and deployed to the FZM and integration tests are then run against this freshly compiled software. Any errors experienced during testing phase will result in the whole continuous integration pipeline to fail.



Figure 16. *Flexi Zone Micro in the laboratory environment*

Figure 16 is a picture taken from the Flexi Zone Micro that is used to run the integration testing. The radio attenuators that were mentioned in Section 4.1 can also be seen connected under the FZM. Due to the fact that the testing environment is indoors, and the building structures filter out most the GPS signals, a GPS signal repeater is set up to the laboratory environment. Despite this fact, the GPS signal is often too weak for the FZMs that are set up in the laboratory. This is why rather than just having GPS antennas in place, the GPS antennas are brought closer to the GPS signal repeater by having a radio cable between the FZM and the GPS antenna.

4.1.2 Power distribution unit

The FZM receives its input power from a power distribution unit (PDU). PDU is a device that allows remote handling of its power sockets. The test environment uses APC AP7920B PDU, that provides 8 independent power sockets and control functionality for them. Since power outages are an expected occurrence in the environment in which the Kuha base station is deployed, a certain type of robustness needs to be expected from both the software and the hardware. PDU can be used in the integration tests to simulate the power outages and that the automated recovery actions are sufficient. PDU used in the test environment provides an SSH and Telnet interfaces that can be used to modify the states of the power sockets.

```
apc>olStatus all
E000: Success
 1: fzm14: On
 2: Outlet 2: On
 3: Outlet 3: On
 4: Outlet 4: On
 5: Outlet 5: On
 6: Outlet 6: On
 7: Outlet 7: On
 8: Outlet 8: On
apc>
```

Figure 17. SSH interface of the PDU showing the statuses of the sockets

APC PDUs allow sockets to have custom names. Figure 17 is a screenshot taken from the SSH interface that the PDU provides with a command “olStatus all” given that is used to print the status of all the sockets. Socket 1 is currently used by the FZM that is used to execute the integration tests. The FZM has an identification of “fzm14” in the lab environment which is why the socket is also named accordingly. This eases up the mapping between the PDU sockets and the FZMs in the test cases as well as in a scenario where the test environment is extended to include multiple FZMs.

Typically, PDUs can also be used to monitor for example the current or power usage of all the devices connected to it. However, this kind of functionality is not a requirement now in the testing environment although the PDU in use provides it. The PDU is connected to the Smokebox via an ethernet cable and the integration tests use SSH when issuing commands to the PDU.

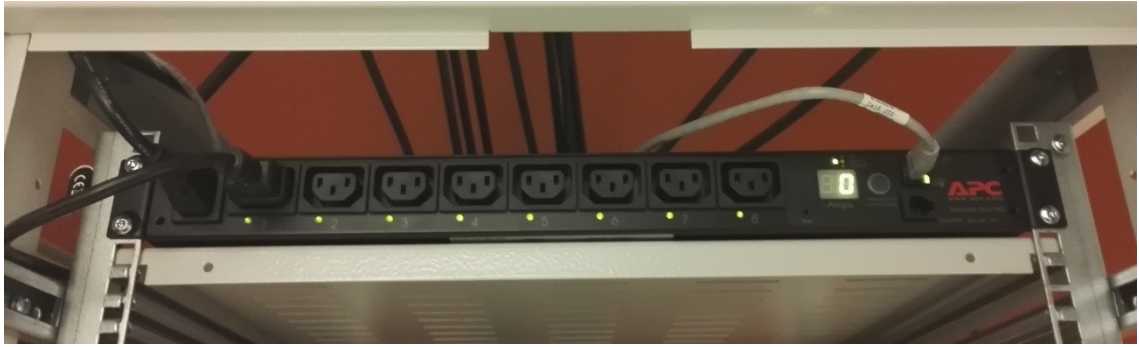


Figure 18. Power distribution unit

Figure 18 is a picture taken of the APC 8 socket power distribution unit. Currently only one socket is used since the testing environment only consists of one FZM. Test library written for the PDU state manipulation consists of functionalities such as rebooting individual sockets as well as checking socket states. During the PDU test library initialization, the FZM identification is given as a parameter and the test library uses this information when rebooting or checking the states of the power sockets.

4.1.3 User equipment

User equipment (UE) used in the testing is Samsung Galaxy S5. In this thesis, the term UE is used to refer to a mobile phone although in LTE the term user equipment refers to any kind of LTE-capable mobile device, such as a mobile phone or a tablet. In the testing environment, the UEs can be connected to the base stations via traditional air interface or directly via a physical radio cable. When the radio cable connection is used, it is extremely important to use the radio attenuator between the connection since the transmit power will be a lot less attenuated compared to air interface. In the case the air interface is used, the radiation from base station is contained so a certain type of radio frequency (RF) box must be used.

Due to the fact the base stations in the lab radiate on different LTE bands, the user equipment is also locked to a certain frequency band to ensure that the tests are run towards the correct base station. Samsung Galaxy S5 provides functionality for frequency locking by using certain type of key codes in phone application to gain access to developer specific options.

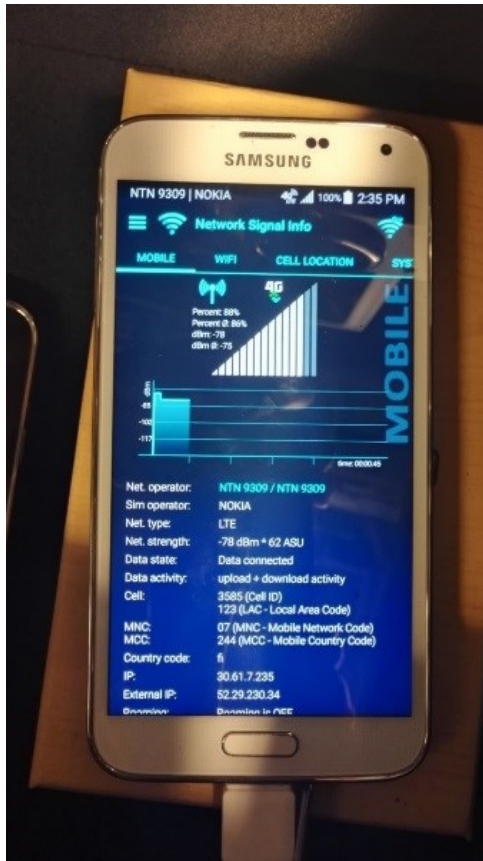


Figure 19. User equipment

The UE can be controlled remotely using a test library. This test library contains functionality for issuing simple HTTP requests to the Internet. This way the integration tests can verify the functionality of the user plane. Also, the test library contains functionality that enables and disables the airplane mode in the UE. Practical experience during the development work suggested that this kind of functionality was necessary since the UE might not immediately do the cell attaching when the cell becomes available. It was seen that enabling and then disabling the airplane mode triggered the attaching procedure in the UE side. For this reason, no control plane specific commands were created since the attaching procedure can be used to verify control plane functionality.

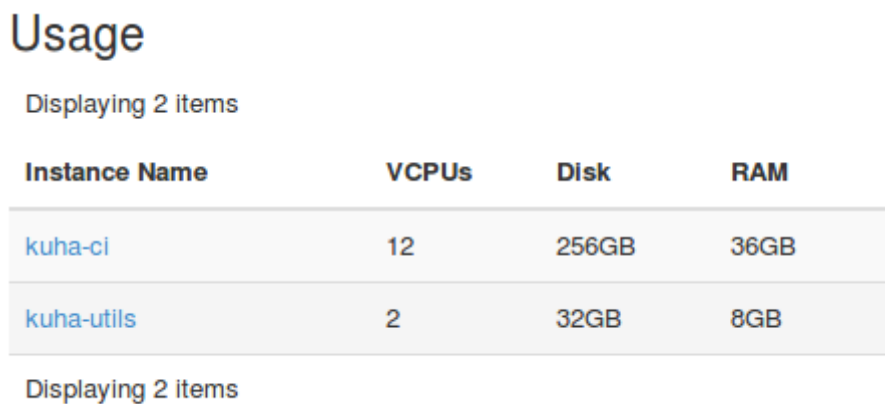
4.1.4 Smokebox

Smokebox is a term used in the test environment and in this thesis for a Linux-based server that is the central piece of the testing environment. Smokebox is used to connect to the FZMs, UEs and to the PDU. Smokebox is a Quanta manufactured server hardware. The FZMs and PDU are connected to it via an Ethernet cable and the UEs via USB. Integration test job in Gitlab CI that is executed in the cloud utilizes an SSH connection to connect to the Smokebox. The test job executes the tests on FZM by using Robot framework. Test libraries written for the Robot Framework can be used to easily manipulate the test environment's state. Robot framework will be discussed more in depth in

Subsection 4.3.1. The Ethernet connectivity between the FZMs and the Smokebox in the test environment is achieved by two Quanta-based switches, T1048-LB9 and T3048-LY9. The Smokebox also runs Android Debug Bridge (ADB) which is used to issue commands to the UEs. ADB is a command-line tool that provides an Unix shell like connection to the UEs that are connected to the host computer.

4.1.5 OpenStack cloud instances

Software compilation is done on an OpenStack cloud instance. The connectivity and build triggering between the Gitlab and the cloud instance is achieved by using a Gitlab runner. Figure 20 is a screenshot representing the OpenStack's graphical user interface that shows the overview of the created cloud instances.



The screenshot shows a web interface titled "Usage" with a sub-header "Displaying 2 items". Below this is a table with four columns: "Instance Name", "VCPUs", "Disk", and "RAM". The table contains two rows of data. Below the table, it says "Displaying 2 items" again.

Instance Name	VCPUs	Disk	RAM
kuha-ci	12	256GB	36GB
kuha-utils	2	32GB	8GB

Figure 20. OpenStack graphical user interface displaying created instances

The testing environment consists of two different cloud instances. The continuous integration processes, such as software compilation and test execution triggering, are executed on the cloud instance called kuha-ci. Transport software compilation is automated by a build automation tool called Make, which is widely used in Unix-like operating systems. Make can be allowed to utilize all the available cores on the host machine for parallel execution. Therefore, the number of allocated virtual processors is kept as high as possible.

Utility scripts are executed on an instance called kuha-utils. Utility scripts handle certain supporting tasks such as uploading new software releases into an internal S3 bucket. This instance also hosts a very simple webpage that lists all the aforementioned software releases and provides download links to them. These utility programs are introduced in Section 4.4.

4.2 Continuous integration pipeline

In the testing environment, the jobs are executed in a Shell executor but the build itself within a Docker container. The reason for this laboured approach lies within the dependencies of the compilation. The legacy build system has been tailored to be used in a specific compilation environment with symbolic links in the repository that point to certain network file shares. These network file shares then provide software libraries against which the compilation is made. The ready-made Docker images that Gitlab or Docker hub provides do not fulfil this kind of very specific requirement, which resulted a decision to create a custom Docker image.

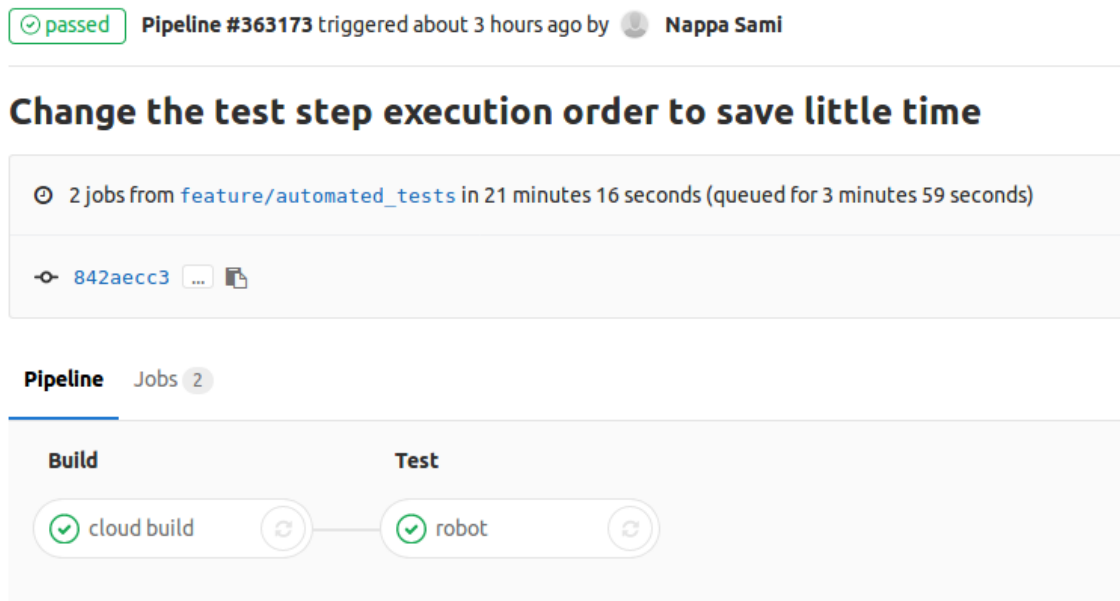


Figure 21. Overall picture of the build pipeline

Figure 21 is a screenshot taken from the Gitlab web user interface illustrating the output of the executed pipeline. The build pipeline consists of two stages, build and test, with both having one job. These stages will be discussed in the following subsections. Continuous integration process follows the precept illustrated in Figure 10 on page 15 where the continuous integration process should be triggered by a new commit in the version control system. Whenever developers commit their changes into the version control system, the continuous integration process defined in `.gitlab-ci.yml` ignites the continuous integration pipeline. This pipeline starts the execution from the software compilation and then moves on to the integration tests in the case the software compilation was successful.

4.2.1 Compilation

Before the software compilation can be started, a Docker image is downloaded from an S3 storage and is imported into the integration machine's Docker service. Docker is a container virtualization technology. Docker is designed to overcome the overhead, that

the traditional virtual machines introduce into the host system. While regular virtual machine is essentially just a copy of the operating system that is running on top of a hypervisor in the host hardware, Docker container is a lightweight computing resource. Docker approach eliminates the need of having separate operating systems, for every container since the containers utilize the host operating system [26]. Figure 22 illustrates the differences between traditional virtual machines and Docker containers.

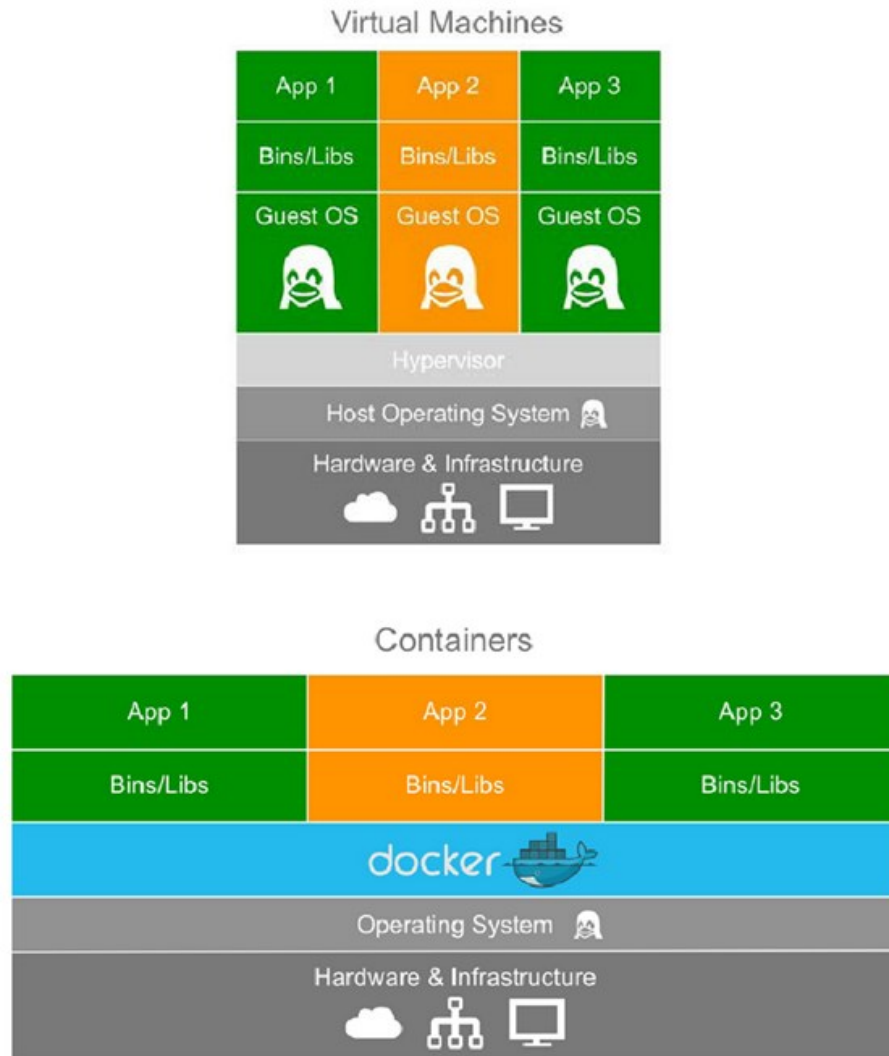


Figure 22. Comparison between traditional virtual machines and Docker containers. Modified from [26]

During compilation and the Docker image start-up, the necessary network file share mounts are made. This kind of approach was done so that the compilation could be executed somewhat environment independently. Full environment independency is not achieved by this approach since the network file share mounts are part of the internal infrastructure and cannot be accessed from an outside network. The full environment independency in compilation could be achieved by including all the required dependencies within the Docker image. However, it was decided that this approach would not be used

due to the complexity of the legacy dependencies and due to the fact that including all the dependencies into one Docker image would vastly grow the size of the image itself.

The compilation initiation is handled by a Perl script that sets certain environment variables for the build. This Perl script then invokes the actual software compilation that uses `make`. The compilation initiating Perl script also handles certain command line arguments, such as “clean” that is used to clean the already compiled object files. Compilation phase also packages the compiled transport software into a tar package that is then usable by the FZM.

4.2.2 Test stage

Testing stage executes integration tests against the FZM with the newly compiled transport software. The main goal for integration tests is to test the usual use case scenarios that the Kuha base station solution provides and to test the common faulty scenarios that it should recover from. From Gitlab CI’s point of view the configuration for the test stage is default in terms of failures and execution. Whenever the software compilation succeeds in the build stage, the compiled binaries, called build artefacts, will be passed forward on to the following stages. These build artefacts are also stored in the Gitlab server for 24 hours after a successful compilation and can be accessed via Gitlab’s web user interface for manual inspection. The testing stage uses the build artefact from the previous stage and deploys them into the FZM that is used for the integration tests and reboots it. After the reboot, the FZM has taken the new software into use and the execution of the integration tests can be started on Smokebox. Gitlab CI system inspects the return values of the build steps and fails the job in case the command that was executed has returned a value that indicates that an error has happened. In case the Robot command, that is executing the integration tests, fails also the whole test stage fails and the developer will be notified via email. Robot framework provides logs after every execution and these logs can be used to inspect the possible faults that happened during the integration test phase. The logs from the integration tests are stored in the Gitlab server for 1 week. The integration tests will be approached in more technical way in Section 4.3.

Since the test stage or the FZM contains no logic beyond the simple fault detection, it is possible for the test stage to deploy a software into the FZM that does compile but does not work. This kind of software could drive the FZM into such state where it is not possible to use the FZM for test stages that the following build pipelines would require. This kind of scenario requires a manual intervention. As an improvement, a software could be created for the test stage that would trigger recovery actions automatically in the FZM side after failures. One simple recovery action would be to store a proven-to-work software in to a safe place within the FZM’s non-volatile memory and take that into use when a failure during the integration tests is detected.

4.2.3 Baseline knife request creation

Baseline is a term used to illustrate a complete, compiled base station software. This software includes several subcomponents that handle all the functionality of a base station. Transport software is one subcomponent of a baseline. Baseline contains a security measure that can be used to verify the integrity of its contents, such as the software and the configuration. There are multiple ways to ensure integrity of a software in computer systems. One of the ways is to use a secure hash function to calculate the digest of a software whose integrity is to be ensured. For example, a hash function called MD5 produces a 120-bit digest from the given input. The idea is that if a malicious user has somehow injected the software, the calculated message digest will also differ from the original one [27].

This same method is used to ensure the integrity of a baseline and its contents. The digest of the software that is about to be run will be calculated during the FZM boot. If the calculated digest does not match with the information that the FZM already has, it refuses to execute the software as a safety measure. Therefore, a baseline knife must be created to fulfil this requirement. Baseline knife is a term used to describe an official modification of an existing baseline. Baseline knife creation process calculates the digest of the modified software within the baseline, outputting a new baseline version where the message digest will be calculated correctly. This newly outputted baseline can then reliably and securely be used in FZM.

As a final stage in the Gitlab CI, a new request for the baseline knife will be created. This request requires information about the version from which the baseline is created as well as the modified software. The modified software will be contained in a zip file called `knife.zip`. The term `knife.zip` is also used in this thesis to describe the modified software that will be used in the knife build. The baseline knife does not get immediately created due to the fact that the baseline knife creation executes a full baseline compilation that takes several hours to complete. Therefore, it was decided that the baseline knife request creation stage would not wait for the baseline knife compilation to complete. Instead, the responsibility to do the necessary actions once the baseline knife creation is completed was moved to the supporting software that is introduced in Section 4.4.

The baseline knife request creation is not yet included in the continuous integration process. This is because the new transport software is not considered mature enough for the continuous packaging needs that the baseline knife request creation provides. However, moving towards the continuous baseline knife request creation can be done with very little effort since the scripts required for packaging the software and the knife request creation already exists.

4.3 Integration tests

Since the transport software undergoes a massive rework, new test scenarios for verifying the functionality must be created. For this reason, Robot framework introduced in the next subsection will be utilized. As for demonstration purposes, Subsection 4.3.2 introduces two of the created test scenarios.

4.3.1 Robot framework

Robot framework is an open source, Python-based, keyword-driven acceptance test automation framework. It is designed to be general purpose framework for acceptance test-driven development (ATDD). Robot framework provides easy-to-use tabular test data files that provide certain level of abstraction to the test files and the libraries that execute the functions on system under test (SUT). This extensible nature of Robot framework makes it very easy to combine it with in for example Selenium in website development. [28]. Below is a figure that illustrates the modular nature that the Robot framework is built upon.

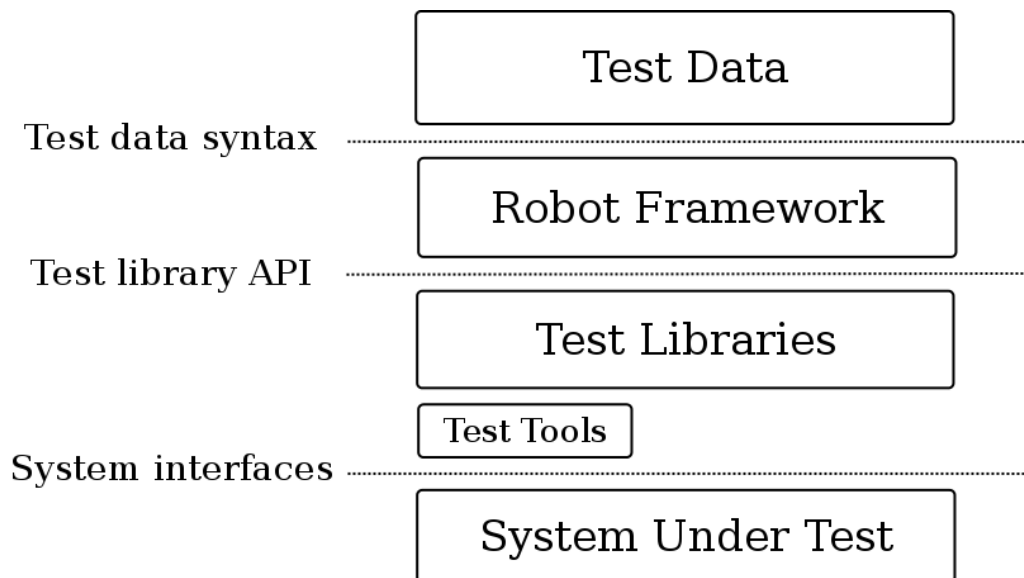


Figure 23. Modular nature of the Robot framework. Copied from [29]

Test libraries mentioned in Figure 23 provide the interface between the test data and the system under test. Test data syntax is designed to be easy to understand containing only the top-level keywords that define the test structure. Custom test libraries are created for the system that is tested. These libraries link test data keywords with the associated commands in the system and, they can be used to examine and validate the output of the given commands or the state of the system. [30]


```

*** Settings ***
Documentation    A test suite with a single test for valid login.
...
...            This test has a workflow that is created using keywords in
...            the imported resource file.
Resource        resource.txt

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]      Close Browser

```

Program 2. *An example test data file. Copied from [29]*

Program 2 above illustrates a simple Robot framework test data file. This file may contain a voluntary documentation field that can be used to describe the current test suite. A test suite is a term used to describe a set of tests that have some sort of connection between each other. Such test suites could be regression or smoke test. Robot framework allows easy execution of a certain test suite if the tester is not interested in running all the possible test cases. One test case consists of one or multiple keywords that define the steps that are done in the said test case. A keyword is a term used in Robot framework to illustrate a step of execution. This kind of approach allows test cases to be constructed freely. By using the example shown above in Program 2, a test case for testing the invalid login credentials could be done by using most of the keywords already used and made for the test case called “Valid Login”. This would mean that only the keyword “Welcome Page Should Be Open” needs to be changed to a keyword that verifies that the login was invalid. Keyword-driven testing simplifies test case writing and allows developers to create generic keywords that could be utilized in many test cases, thus increasing the efficiency of the time used to write test cases.

Robot framework also has a support for variables. Variables are defined with a dollar sign, followed by the variable name inside curly brackets. This means that with syntax `${REBOOT_TIME}` a variable called REBOOT_TIME could be defined. If a keyword has a return value, then the return value could be assigned into this variable and examined later.

In the continuous integration system, the test libraries mentioned in Figure 23 were written for FZM, switch, PDU and the UE. These test libraries allow the developer to manipulate the state of a single device or a combination of the different devices connected to the testing environment. These combinations can be used to manipulate the test environment into a state that imitates real world requirements and use cases as closely as possible. By writing comprehensive test libraries, the developers will be able to create new test cases with little effort.

4.3.2 Test scenarios

One of the goals for the reimplemented transport software is to achieve faster reboot times as well as overall plug-and-play type of operability. The integration test scenarios introduced in this subsection are formatted using the high level of abstraction that the test data file format in Robot framework also utilizes. This subsection introduces two of the existing test cases. The following programs will be containing variables that are named with uppercase letters. These variables are defined in a separate resource file, but they are not shown in this subsection. The test case flow can be understood without knowing the values of these variables.

```

Power cycle FZM in normal condition
  ${outlet_status}    PDU.is outlet on
  Should Be True     ${outlet_status}
  PDU.outlet off on
  Log    Sleep for ${POWER_CYCLE_REBOOT_TIME}
  Sleep  ${POWER_CYCLE_REBOOT_TIME}
  FZM.reconnect SSH to FZM
  Make sure SSH connection is open
  Log    Wait for ${BACKHAUL_REBOOT_TIME}
  Sleep  ${BACKHAUL_REBOOT_TIME}
  FZM.reconnect SSH to FZM
  Verify backhaul connection
  Verify backhaul connection with UE

```

Program 3. Test case for power outage

Test case written for power outage recovery testing is illustrated in Program 3. This test case verifies that the PDU outlet is on before cutting off the power from the FZM. The test keyword implemented for PDU will issue a reboot command to the power outlet. This delayed reboot command cuts the power from the power outlet, then waits for a pre-defined time before returning the power to the outlet. The pre-defined wait time is configured within the PDU and in this case, it is 5 seconds. After issuing the power reset command, the test execution will wait for the FZM to reboot itself. Finally, once the SSH connection can be re-established, the test case will verify the functionality of the FZM by testing the backhaul connectivity from within the FZM itself and from the UE's perspective. Practical experience has shown that testing the connectivity from both the FZM as well as from the UE can be very beneficial when narrowing down the faulty scenarios. This is because during the transport development, also the logic that handles the packet routing within the FZM had to be reimplemented. Due to the fact that the UE data is routed in the user plane, it is also routed differently within the FZM and transport component is responsible for handling this.

```

Test autoconnection
  Verify that configuration exists
  FZM.decommission
  Log    Wait for the FZM to reboot and commission itself
  Repeat Keyword    2 times    Sleep    ${REBOOT_TIME}
  Wait Until Keyword Succeeds    3 times    ${REBOOT_TIME}    FZM.reconnect
SSH to FZM
  Verify that configuration exists
  Sleep    ${BACKHAUL_REBOOT_TIME}
  Verify backhaul connection
  Verify backhaul connection with UE

```

Program 4. Test case for autoconnection

Program 4 illustrates a test case that tests the autoconnection functionality. Autoconnection is a term used to illustrate a procedure in which the FZM fetches the required certificates as well as gets its configuration from the cloud OSS and commissions itself. Commissioning, in this context, indicates a procedure in which the FZM does a full base station reconfiguration. This reconfiguration also includes parameters that are not important from the transport point of view, such as the radio parameters. First this test case verifies that the FZM does have a configuration in it. After that it will decommission the FZM, which in turn will trigger the autoconnection procedure. Decommissioning procedure will delete the existing device configuration and the certificates from the file system of the FZM. Once the decommissioning is executed, the test case waits for the FZM to commission itself and to fetch the required certificates. The FZM will execute a system reboot after the decommissioning and after the successful commissioning, which is why the sleep keyword is executed twice in the test case. Commissioning is triggered by a program that is running within the FZM. This program has certain conditions on when the commissioning should occur, and missing configuration is one of those conditions. Once the commissioning is done, the test case will verify that the new configuration is stored within the FZM and that the connectivity from both the FZM and the UE is working. Ideally, autoconnection should be done very rarely, only when the FZM is first taken into use or when a partition fallback occurs. Since the autoconnection is such a thorough functionality, it must work. Malfunction in autoconnection will prevent the deployment of the FZM, thus the reason why this test case was developed.

Functional Test Log Generated
20180323 16:07:39 GMT+02:00
19 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:10:35	█
All Tests	1	1	0	00:10:35	█

Statistics by Tag

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Functional	1	1	0	00:11:03	█

Test Execution Log

SUITE Functional

Full Name: Functional
 Documentation: Test suite for functional tests
 Source: /home/sanappa/robot/tests/functional.robot
 Start / End / Elapsed: 20180323 15:56:36.537 / 20180323 16:07:39.724 / 00:11:03.187
 Status: 1 critical test, 1 passed, 0 failed
 1 test total, 1 passed, 0 failed

TEARDOWN resources.Generic suite teardown

TEST Test autoconnection

Full Name: Functional.Test autoconnection
 Start / End / Elapsed: 20180323 15:56:48.958 / 20180323 16:07:23.641 / 00:10:34.683
 Status: PASS (critical)

- SETUP** resources.Reset reboot counter
- KEYWORD** resources.Verify that configuration exists
- KEYWORD** FZM.Decommission
- KEYWORD** BuiltIn.Log Wait for the FZM to reboot and commission itself
- KEYWORD** BuiltIn.Repeat Keyword 2 times, Sleep, \${REBOOT_TIME}
- KEYWORD** BuiltIn.Wait Until Keyword Succeeds 3 times, \${REBOOT_TIME}, FZM.reconnect SSH to FZM
- KEYWORD** resources.Verify that configuration exists
- KEYWORD** BuiltIn.Sleep \${BACKHAUL_REBOOT_TIME}
- KEYWORD** resources.Verify backhaul connection
- KEYWORD** resources.Verify backhaul connection with UE

Figure 24. Log created by Robot framework

Robot framework creates log outputs whenever test cases are executed. This log can be used to inspect what failures happened during the test case execution. Figure 24 is a log output created by executing only the test case called “Test autoconnection” that was introduced in Program 4. As it can be seen, all the keywords that were executed were successful, which also means that the test case execution was successful.

4.3.3 Software architecture in integration tests

Some of the variables mentioned in the test case examples above, such as the “REBOOT_TIME”, are defined in a common resource file that contains the definition for common variables. This resource file also provides some basic test keywords and is responsible for importing and initializing the test libraries that are used to issue commands to FZM, UE, network switches and the PDU. The resource file can be seen as a one extra layer of abstraction between the test data and system under test. This allows an approach where the test data files can be constructed in high level of abstraction, therefore increasing the readability of the test data files. Figure 25 is an illustration of the dependencies

within the test environment. The figure is read from top to down, higher elements are dependent on the lower elements that their arrows are pointed to. The figure is created by using the Figure 23 shown in Subsection 4.3.1 as the basis.

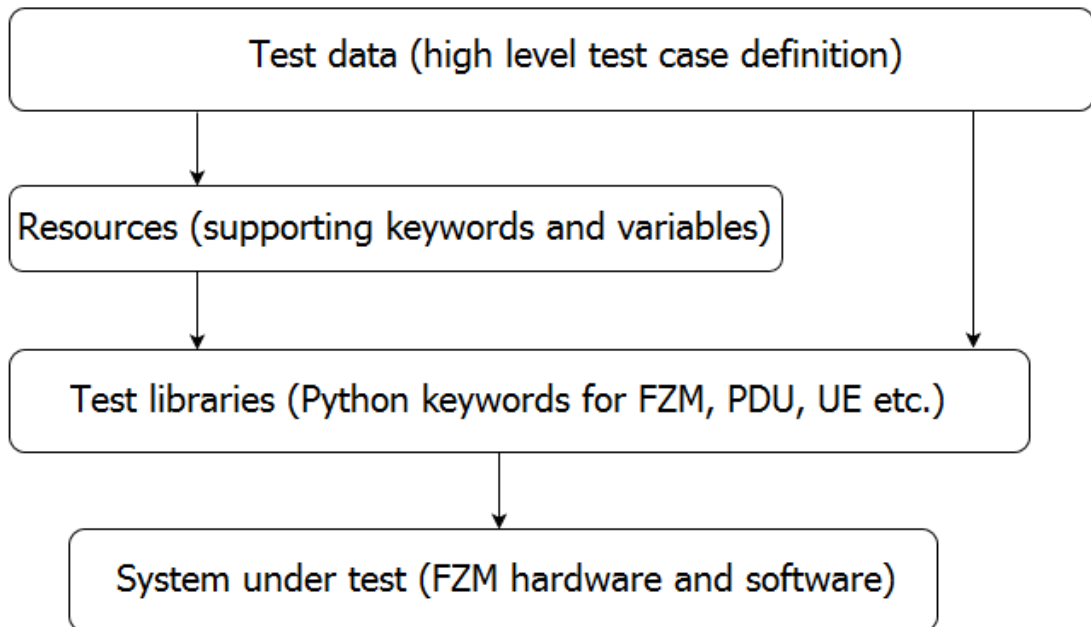


Figure 25. *Software architecture in testing environment*

Test libraries are implemented in Python. They provide an interface to the system under test for the resource and test data files. Usually test libraries only contain very simple functionality, for example, a possibility to test UE data connectivity that is introduced below in program 5. The result from this keyword can then be used to deduce if the user plane works in the test case or not. Robot framework is able to utilize the functions implemented in Python classes in the test data files.

```

Verify backhaul connection with UE
    ${UE_param}    Check if UE param is given
    Run Keyword Unless    ${UE_param}    Return From Keyword
    Log    Reconnect UE to cell in case connection was lost earlier
    UE.reconnect to cell
    ${UE_ping}    UE.ping    www.google.com
    Should Be True    ${UE_ping}
    ${UE_connectivity_state}    UE.is data and voice connected
    Should Be True    ${UE_connectivity_state}
  
```

Program 5. *Keyword for testing UE connectivity*

Above, in program 5, the resource keyword for testing the UE data connectivity is introduced. This keyword utilizes keywords provided by the Robot framework's built-in library as well custom test libraries. This keyword first verifies whether the UE variable containing the UE identification is given in the Robot execution command. In the case

the UE variable is not defined, the keyword exits. This keyword also uses a custom-made test library for issuing commands to the UE.

```
def reconnect_to_cell(self):
    self.set_airplane_mode(True)
    self.set_airplane_mode(False)
    time.sleep(5)

def set_airplane_mode(self, onOff):
    self._execute_shell("adb -s " + self.ue + " shell 'settings put global
airplane_mode_on %i'" % onOff)
    self._execute_shell("adb -s " + self.ue + " shell 'am broadcast -a an-
droid.intent.action.AIRPLANE_MODE'")
```

Program 6. *Methods from UE test library*

Program 6 illustrates methods the UE class provides in the Python level. The methods introduced above utilize SSH to issue commands to the UE by using the ADB. The aforementioned UE identification is given to the UE class as a parameter in the constructor. This ensures that the ADB commands are issued to the correct UE.

The resource file contains also a definition for a Robot framework specific test suite teardown and test case setup procedure. In Robot framework, a teardown is a procedure that will be executed after everything else is executed. Teardown can be either at test case level or test suite level. Suite level teardown will be executed after every test case within the suite is executed or if the test suite execution is stopped because of test case failure. Similarly, test case teardown is executed after the test case regardless of whether the test case was successful or failure. Test suite and test case setups are executed before the test suite and test case respectively. Teardowns can be used to for example executed clean up procedures within the test environment after the testing has ended [29].

In this thesis, a test suite teardown and test case setups are utilized. The test suite teardown is used to set the environment back into such a state where the test case execution of the following pipelines is not compromised. The teardown procedure will enable the network switch to which the FZM's backhaul is connected to as well as enable the power output from the PDU outlet. The status of the executed teardown called "Generic suite teardown" can be seen in Figure 24. Test case setup utilizes test library call to reset an internal FZM reboot counter. This is done because FZM contains certain very low-level logic that is used to make assumptions about the health of the hardware and the software. One indicator for unhealthy hardware is that if the FZM faces software or hardware failures and has to reboot the system often. When the reboot counter reaches certain value, FZM will automatically execute recovery actions which are unwanted during the execution of the integration tests. Reboot counter value will also get incremented even in scenarios where the reboot could be considered safe, such as the case when the test case itself issues the reboot command. The keyword that handles the reboot counter resetting is called "Reset reboot counter" and its execution can also be seen in the log output in Figure 24.

4.4 Supporting software for the continuous integration system

Due to the fact that the continuous integration process includes also systems that cannot be controlled, a set of supporting software was created. These programs allow the created continuous integration system to operate seamlessly with the existing system and to minimize the manual work that would otherwise be required after the continuous integration pipeline. Both introduced programs are running on the kuha-utils OpenStack cloud instance that was introduced in Subsection 4.1.5. Neither of these programs have any performance requirements which is why they are executed within same cloud instance with minimum hardware requirements. These two programs are also not under constant development.

This section contains two subsections that introduces the supporting software in detail. Subsection 4.4.1 introduces WFT sniffer (Workflow tool sniffer) that automatically downloads compiled software from a third-party source and uploads into Kuha managed network storage. Subsection 4.4.2 introduces program that hosts a website that contains download links to the compiled software.

4.4.1 WFT sniffer

WFT sniffer is a Shell script that handles the management actions that are required for the successfully compiled baseline knives. The requirement for this kind of software arose from the fact that the build system that creates the knife baselines does not store the outputs longer than 2 weeks. It is not feasible for the developer to monitor the status of the newly created baseline knife request since the baseline knife creation takes several hours to complete after the continuous integration pipeline has already finished. This was the motivation behind the development of the WFT sniffer. In brief, WFT sniffer checks the execution status of the baseline knife that was created at the end of the continuous integration pipeline execution and stores the baseline into S3 storage if the status indicates that the knife creation was successful. Information about failures during execution of the WFT sniffer itself will be notified via email, which enables a rapid response. WFT sniffer also used to include housekeeping feature that would delete the zip file used while creating the knife request, but this feature was removed since the knife.zip provides useful debugging information. Figure 26 illustrates the flow of execution of the WFT sniffer.

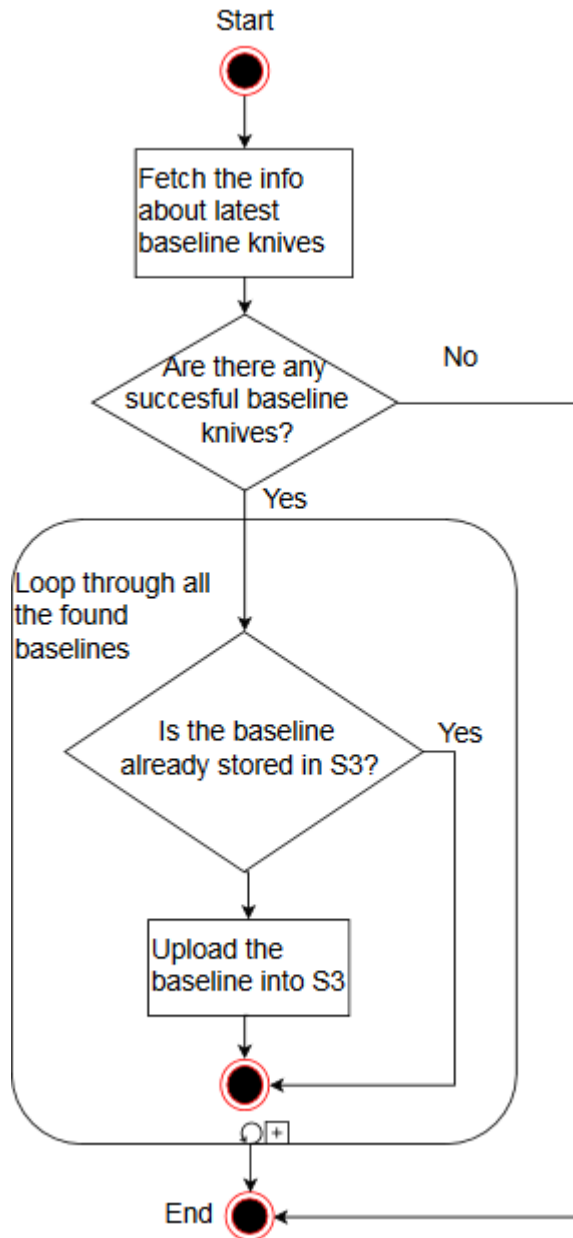


Figure 26. *WFT sniffer activity diagram*

The internal S3 storage used in storing the baseline knives does not provide infinite storage space which means that at some point the S3 storage will be full. Some logic could be added to the WFT sniffer so that it could do housekeeping within the S3 and check the used storage amount and delete the oldest baselines from it. This kind of functionality was not deemed necessary at this part of the project since new baseline knives are not created every day and it is feasible to monitor the S3 storage status manually and remove the old baseline knives by hand. Also, this rather simple sounding housekeeping functionality requirement would very rapidly become rather complex implementation since one baseline knife is always related to one baseline version. The project is supporting multiple baseline versions which in turn means that the logic behind the housekeeping would have to take this fact into account. Goal for this requirement is that it should not be possible to end up into a situation where there would not be any baseline knives stored

for certain baseline version. Ideally, the actual housekeeping functionality would have to store at least one or multiple knives from each of the baseline version.

Status	Pipeline	Commit	Stages
passed	#435467 by latest	master -> 5bef8b24 Remove systemctl and cloud deployment ...	00:00:02 7 minutes ago
passed	#435348 by latest	master -> 5bef8b24 Remove systemctl and cloud deployment ...	00:00:04 about an hour ago
passed	#435226 by latest	master -> 5bef8b24 Remove systemctl and cloud deployment ...	00:00:02 about 2 hours ago

Figure 27. Screenshot from the scheduled WFT sniffer executions

WFT sniffer is executed periodically once in an hour using the Gitlab CI schedules that provides a Cron like execution possibility. The advantage of using this kind of approach is that it provides certain sort of transparency to the system since the execution output can be seen from the Gitlab UI. By default, the schedule execution will use the latest commit in the master branch of the WFT sniffer repository.

4.4.2 Baseline listing

Baseline listing is a Python-based simple web page that lists all the baselines that are stored in the S3 storage. The baselines are uploaded into S3 storage with access control privileges public within the intranet. Certain actions are made towards establishing a production line that would enable fast uploading of the knife baseline into multiple base stations. The script that is being developed for this requirement uses the data visible in this website to download the baseline knife, creating an installer from it and uploading it into the base station. However, the aforementioned script is not a part of this thesis.

Kuha FZM baselines		
latest		
2017-11-14 11:27		247137_release_image.tar
2017-11-13 14:59		246791_release_image.tar
2017-11-13 16:00		246833_release_image.tar
2017-11-03 09:08		244169_release_image.tar
2017-10-24 19:53		241677_release_image.tar
2017-10-24 20:25		241679_release_image.tar
2017-10-09 03:07		236803_release_image.tar
2017-10-04 15:00		236801_release_image.tar
2017-10-03 13:40		236487_release_image.tar
2017-09-29 09:54		235847_release_image.tar
2017-09-29 14:29		235971_release_image.tar
2017-09-27 07:20		234895_release_image.tar
2017-09-25 13:00		234193_release_image.tar
2017-08-25 14:00		226939_release_image.tar
2017-08-23 09:07		225589_release_image.tar
2017-08-21 10:01		225067_release_image.tar
2017-08-21 10:02		224875_release_image.tar
2017-08-21 10:02		224943_release_image.tar
2017-08-21 11:21		224849_release_image.tar
2017-08-18 13:22		225227_release_image.tar

Figure 28. Website listing all the knife baselines

Figure 28 is a screenshot taken from the website listing all the successful baseline knives. Part of the screenshot is censored due to the business-critical data contained in it.

#4	P master → 57ad607b Don't deploy web server to cloud since ci runn...	deploy to cloud (#577438) by	5 months ago	Rollback
#3	P master → 221aa73a Remove obsolete ampersand from python com...	deploy to cloud (#551315) by	6 months ago	Rollback
#2	P master → f8028116 Start the webserver in the cloud	deploy to cloud (#551227) by	6 months ago	Rollback

Figure 29. *Baseline listing software deployments*

Whenever a new commit is made into the repository that holds the baseline listing source code, the Gitlab CI will automatically do the deployment of the new source code into the kuha-utils cloud instance. The script that handles the deployment will kill the running web server process from the cloud instance and starts a new process. The deployment itself utilizes the environments functionality that the Gitlab provides. As seen in Figure 29, the environment's functionality allows an easy way to rollback baseline listing deployments if such action needed to be taken. This would allow to deploy working version back into the cloud instance while solving the issues with the version that was rolled back.

4.5 Further development of the system

The continuous integration system will be under further development even beyond the practical part of this thesis. From the development direction point of view, the following logical steps from the continuous integration would be to move towards continuous delivery and continuous deployment. Due to the fact that the new software releases require more extensive testing than only the automated tests created within the context of this thesis, the continuous deployment would not be a wise extension as of yet. Extending from the software introduced in this thesis, the move towards continuous delivery and deployment could be done by having the pipeline extended from the WFT sniffer. This move would render the baseline listing software introduced in Subsection 4.4.2 somewhat obsolete. As a result of the practical work done during this thesis, the continuous integration pipeline ends into the WFT sniffer. This is because WFT sniffer is the last component in the pipeline, since it monitors the status of the knife builds and uploads the finished knife builds into S3. From an architectural point of view, the Kuha OSS already provides functionality for deploying new baselines into the devices that are connected to it. With this in mind, instead of using WFT sniffer to just store the finished knife builds into S3, it could be used to also inform the Kuha OSS about the new baseline being ready for deployment. This allows very easy way to extend the continuous integration into continuous deployment.

Kuha OSS software development is making a use of continuous delivery where the development team utilizes a staging environment that is separate from the production environment but otherwise identical. Whenever a new feature is added to the codebase, it is deployed to the staging environment. This approach is used to verify and test the new features in addition of unit and integration tests.

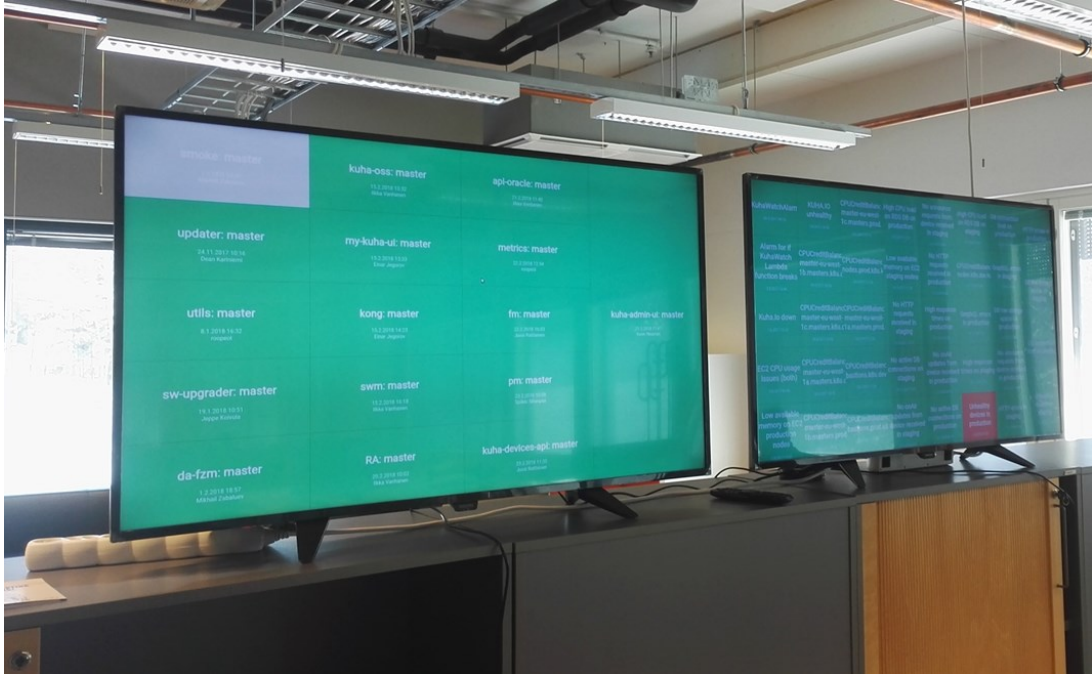


Figure 30. Radiators

Developers use radiators shown in Figure 30 to monitor the state of the staging and production environments for possible faults. Both the staging and production environment contain some logic that monitors the state of the environment. The values received from the system are called metrics and these metrics are used to detect any faulty states within the environment. Whenever a metric breaches the threshold, an alarm will be raised. Every alarm of the system is represented by a box in the radiator view with the addition of direct alarms pushed into the team's internal communication software. Whenever an alarm occurs, it causes the corresponding radiator box to go red, thus giving the visual feedback to the developers. Radiators also provide good and fast overall picture of the environment's state. Detecting the faults already in staging environment helps to maintain a stable production environment. There is no specific time period defined for how long the new feature needs to be in staging before the release to production can be made. The decision on releasing is done on case by case which means that whenever the new feature is deemed working, it is released to production environment.

From the integration testing point of view, development of additional test cases is an obvious improvement. Furthermore, since Gitlab provides support for scheduling the pipeline execution, test cases could be developed utilizing that feature. One possibility would be to create a Gitlab schedule that would execute a certain type of test case that tests

robustness of the complete system. This test case could be scheduled to run during the weekend when there is no other software development done. Normally the test environment and the devices in it are idling during the weekend. The test case itself would for example test the full data transmission, from UE to the Internet. In case the test case fails or there are other errors found during the testing, the test script would collect the relevant logs from the environment and store them. This would then allow developers to investigate the possible error causes during the upcoming week. This type of test case would be beneficial considering the environment in which the base station will eventually be in. The test environment does not currently test the VoLTE features, which would also be beneficial to test. Of course, this would require either dedicated hardware if, for example, the voice quality would be the passing criteria for the test case. Another possibility would be to create some sort of custom software that would verify the connectivity between the UEs performing the VoLTE call.

5. CONCLUSIONS

This thesis took a deep dive into the mobile network technologies and the issues that rural areas are bringing to the mobile network operators. This thesis also introduced Kuha, which is a solution that is designed to overcome the issues related to rural connectivity. The theoretical part also introduced the software development practice called continuous integration and how it, with the help of modern methodologies, solves the issues that traditional software development methodologies brought.

The continuous integration system that was created during this thesis has been in use and under development ever since the project started. This system will be under further development even after this thesis. Robot framework that was used with the integration tests allows an easy way to extend the existing set of test cases. Test libraries written for Robot framework provide already an extensive control over the testing environment in which the integration tests will be executed.

Further development chapters introduced the following steps that the system could take. At the end of the writing process of this thesis, my personal view is that the move towards continuous delivery seems to be quite near as the discussions within the team have had such trends.

The goal of creating a continuous integration system, that was introduced in introduction chapter, was fulfilled. Integration tests are run for every new feature in order to insure quality. There have been occurrences in which the integration tests have revealed an underlying issue that a new feature has introduced. Therefore, it can be concluded that the integration tests have brought value to the software development team.

REFERENCES

1. **Sauter, Martin.** *3G, 4G and Beyond : Bringing Networks, Devices and the Web Together.* New York : John Wiley & Sons, Incorporated, 2013.
2. **Ericsson.** *Ericsson Mobility Report November 2017.* 2017.
3. *Heterogenous Networks (HetNets) Using Small Cells.* **Mistry, Priyal.** International Journal of Scientific & Technology Research, 2015.
4. *Base Station Placement Algorithm for Large-Scale LTE Heterogeneous Networks.* **Lee S, Lee S, Kim K, Kim YH.** PLoS ONE, 2015.
5. **Hassan, Q. F., Khan, A. U. R., Madani, S. A.** *Internet of things : challenges, advances, and applications.* Milton : CRC Press, 2017.
6. **BCS.** *The Internet of Things: Living in a Connected World.* BCS, 2017.
7. **Rashmi Sharan Sinha, Yiqiao Wei, Seung-Hoon Hwang.** A survey on LPWA technology: LoRa and NB-IoT. *ICT Express.* 2017.
8. **Naganand Doraswamy, Dan Harkins.** *IPSec: The New Security Standard for the Internet, Intranets, and Virtual Private Networks.* Prentice Hall Professional, 2003.
9. **Chowdhury, Shahriar Iqbal.** Royce initial Waterfall model an investigation. *Galib's virtual identity.* [Online] October 12, 2009. [Cited: December 2, 2017.] <https://imgalib.wordpress.com/2009/10/12/royce-initial-waterfall-model-an-investigation/>.
10. *Approaches for development of Software Projects: Agile methodology.* **Kotaiah, Bonthu and Khalil, Md Asif.** Udaipur : International Journal of Advanced Research in Computer Science, 2017.
11. **Kunz, Leigh and Associates (KL&A).** Agile Software Development. [Online] [Cited: January 21, 2018.] <http://kunzleigh.com/about/our-approach/>.
12. **McKenna, Dave.** *The Art of Scrum: How Scrum Masters Bind Dev Teams and Unleash Agility.* Apress, 2016.
13. **Canty, Denise.** *Agile for Project Managers.* Philadelphia, PA : Auerbach Publications, 2015.
14. **Jaskiel, Rick D. Craig and Stefan P.** *Systematic Software Testing.* Artech House, 2002.

15. **Myers, Glenford J.** *The Art of Software Testing, second edition*. John Wiley & Sons, 2004.
16. **S. Koirala, S. Sheikh.** *Software Testing: Interview Questions*. Infinity Science Press, 2008.
17. **Lewis, William E.** *Software testing and continuous quality improvement, third edition*. s.l. : Auerbach Publications, 2009.
18. **Smart, John Ferguson.** *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media, Inc., 2011.
19. **ThinkSys.** The Fundamentals of Continuous Integration Testing. *ThinkSys QA Blog*. [Online] February 24, 2017. [Cited: November 28, 2017.] <http://blogs.thinksys.com/fundamentals-continuous-integration-testing/>.
20. **ZeroTurnaround.** Java Tools and Technologies Landscape Report 2016: Trends and Historical data. [Online] July 14, 2016. [Cited: May 15, 2018.] <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016-trends/>.
21. **Soni, M.** *DevOps for Web Development*. Birmingham : Packt Publishing, 2016.
22. **GitLab Inc.** Introduction to pipelines and jobs. *Gitlab Documentation*. [Online] [Cited: January 20, 2018.] <https://docs.gitlab.com/ee/ci/pipelines.html>.
23. **Hethy, J.** *GitLab Repository Management*. Olton : Packt Publishing, 2013.
24. **Atlassian.** Git Feature Branch Workflow. *Atlassian Git Tutorial*. [Online] [Cited: January 20, 2018.] <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>.
25. **Gitlab Inc.** Gitlab Runner. *Gitlab Documentation*. [Online] 2017. [Cited: January 20, 2018.] <https://docs.gitlab.com/runner/>.
26. **McKendrick, Russ.** *Extending Docker*. Birmingham : Packt Publishing, 2016.
27. **Furht, Borko.** *Encyclopedia of Multimedia*. Springer Science & Business Media, 2008.
28. **Laukkanen, Pekka.** *Data-Driven and Keyword-Driven Test Automation Frameworks*. Espoo, 2006.
29. **Pekka Klärck, Janne Härkönen et al.** Robot Framework. [Online] [Cited: November 15, 2017.] <http://robotframework.org/>.

30. **Bisht, S.** *Robot Framework Test Automation*. Olton : Packt Publishing, 2013.