OULUN YLIOPISTO

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAM IN ELECTRICAL AND WIRELESS COMMUNICATIONS ENGINEERING

MASTER’S THESIS

REUSABLE GENERIC SOFTWARE ROBOT

Author	Jori-Pekka Rautava
Supervisor	Mika Ylianttila
Second examiner	Erkki Harjula
Technical advisor	Tero Mäntymaa

November 2019

Rautava J-P. (2019) Reusable generic software robot. University of Oulu, Degree Programme in Electrical Engineering. Master's Thesis, 44 p.

ABSTRACT

The main purpose of this thesis was to create a generic reusable software robot which can be deployed into any IaaS type of a cloud service. In this thesis the first thing to be researched was how to implement a virtualised environment into a cloud service. The possibilities for virtualising the environment were a container and a virtual machine. The two possible implementations were researched since the resulting implementation must be compatible with a cloud service. Firstly, it was found that a container-based implementation would be the best option because it is lightweight to move around and secondly, a start-up time of a new instance in a cloud service is fast.

Possible cloud providers were scanned after researching possible implementation methods. Two possible cloud providers, AWS and Azure, were studied more closely since they offer an infrastructure as a service and once they are commonly used. AWS was chosen to be the platform to be used because of a higher maturity level and also because of the possibility to add or remove container capabilities.

Finally, it was discussed how a generic reusable software robot was implemented. Notable circumstances of suitable tasks for a software robot were considered.

Key words: Sikuli, AWS, Azure, virtual machine, Docker, virtualisation.

Rautava J-P. (2019) Kertakäyttöinen geneerinen ohjelmistorobotti. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 44 s.

TIIVISTELMÄ

Tässä työssä tutkittiin, kuinka geneerinen kertakäyttöinen ohjelmistorobotti voidaan toteuttaa pilvipalvelussa. Ensin tarkasteltiin erilaisia virtualisointimenetelmiä, joilla ohjelmistorobotti voitaisiin toteuttaa. Tutkitut menetelmät olivat virtuaalikone ja kontti. Näitä kahta toteutustapaa vertailtiin huomioiden valmiin toteutuksen sopivuus pilvipalveluun. Kontti todettiin sopivimmaksi toteutustavaksi, koska se vie vähän tilaa ja uuden instanssin käynnistäminen on nopeaa.

Pilvipalvelutarjoajia tutkittiin, kun sopiva toteutusmenetelmä ohjelmistorobotille oli löydetty. Tutkimuksessa keskityttiin AWS:ään ja Azureen, jotka ovat tällä hetkellä suurimpia markkinoilla toimivia *infrastructure as a service* -tyyppisten pilvipalveuiden tarjoajia. AWS valittiin toteutusalustaksi, koska se on teknisesti edistyneempi kuin Azure ja AWS:ssä on mahdollista lisätä ja poistaa kontin oikeuksia.

Lopuksi esiteltiin, kuinka geneerinen kertakäyttöinen ohjelmistorobotti toteutettiin ja mitä täytyy ottaa huomioon, kun päätetään sopivasta käyttökohteesta ohjelmistorobotille.

Avainsanat: Sikuli, AWS, Azure, virtuaalikone, Docker, virtualisointi.

TABLE OF CONTENTS

FOREWORD

The topic was invented when having conversation with my colleagues Tero Mäntymaa and Tomi Leppälahti. One thing led to another and soon I had a topic which could be introduced to Mika Ylianttila who was the supervisor of this thesis.

I would like to thank everyone who participated in the process of writing this thesis. I would like to thank Tero for all the moments when we have discussed of the state of the thesis and where to head next. Next, I would like to thank Vincit for being so flexible, making it easy to combine working and writing thesis.

I would like to thank Titto for giving priceless help with the grammar of this thesis. Jari Hannu, thank you for mentoring me through the university courses and for clarifying the path through course combinations. Mika Ylianttila, thank you for being supervisor for this thesis and giving feedback when feedback was needed.

For my parents, thank you for raising me and encouraging me to pursue my dreams and goals which led me to this point.

To all my Teekkari friends, thank you. You have made this journey an exciting one. You have been the reason why I have had so great time during my studies. All the experiences and memories will remain, as well as the friendship we have.

Finally, but definitely not the least, thanks to my fiancé, Hanna, who has supported me through the tough moments when I felt like giving up with final courses. With your support I managed to push through all the courses and finish my degree.

At Turku 29.11.2019

Jori-Pekka Rautava

LIST OF ABBREVIATIONS AND SYMBOLS

ABI	Application binary interface
API	Application programming interface
AWS	Amazon web services
CoW	Copy-on-write
CPU	Central processing unit
CSV	Comma separated values
DCT	Docker content trust
GUI	Graphical user interface
HID	Human interface device
HTTPS	Hypertext transfer protocol secure
IaaS	Infrastructure as a service
IDE	Integrated development environment
ISA	Instruction set architecture
JVM	Java virtual machine
KVM	Kernel-based virtual machine
LXC	Linux containers
R/W	Read and write
RAM	Random access memory
RPA	Robotic process automation
OS	Operating system
PaaS	Platform as a service
UID	User ID
VM	Virtual machine
X11	X window system
MB	Megabyte
GB	Gigabyte
Bash	A command processor which runs in a shell
x86	Processor architecture by Intel
x86-64	64-bit processor architecture version of an x86

1 INTRODUCTION

This thesis will focus on possible means of creating a reusable generic software robot. In the theory part of this thesis different ways to implement a reusable generic software robot will be compared. The major interest will be in comparing a container and a virtual machine -based implementations. In a practical part of this thesis a generic reusable software robot will be developed based on the research made in the theory part. The development of a generic reusable software robot aims to make robotic process automation (RPA) solutions more robust and easier to implement in a cost-efficient way when compared to traditional implementations. In the later implementation either an actual machine or a virtual machine is occupied all the time by a robot even the actual tasks are processed only part of the time.

In the theory part possible technologies to implement the software robot in a cloud service like Amazon web services (AWS) and Microsoft Azure will be researched. The best way to implement the software robot measured with indicators of cost and performance will be researched when deciding the technology which will be used in the practical part.

Running multiple robots simultaneously is a performance factor influencing the implementation methods of a software robot. Having multiple robots working simultaneously needs to be researched from an orchestration point of view. A virtualisation is researched in order to provide a functional desktop for a robot working on top of the same hardware with other robots. The robot needs to have a desktop inside a container so it could work. The possible ways to implement the desktop inside a container are discussed.

Possible ways to implement the software robot into a cloud service are a container and a virtual machine. These implementation methods are compared in sense of costs, efficiency and performance. The scaling of the resources in the cloud service will be researched because the scaling of the resources will have an effect in the costs.

After the theory part a detailed explanation on how the reusable generic software robot was implemented is provided. At first, creating a container which has all the necessary parts to run the Sikuli scripts is discussed. Next, implementing a container image into a cloud service and how to manage a lifecycle of a container is researched. Finally, a whole infrastructure needed for managing multiple robots from single control panel is studied.

2 TECHNICAL ANALYSIS

A generic reusable software robot must be easy to implement into different environments. This limitation makes it imminent to study different methods to virtualise the environment where the software robot will work. There are basically two main categories of virtualisations to be considered: a container-based and a hypervisor-based virtualisation. Roughly it can be said that the hypervisor-based virtualisation means the usage of a virtual machine and the container-based virtualisation is about different types of containers. These are the only types which are considered because the software robot acts just like a human from the computer point of view. A screen and input devices for the software robot must be provided just like if a human would be using the computer.

In this part, a virtual machine and container technologies and their differences are in focus. At first, both technologies are introduced and the differences between the technologies are considered. Secondly, the cloud environments are examined, and the necessary factors for the most efficient implementation method for a generic reusable software robot are researched. The most efficient implementation method is considered from multiple angles including the costs, a size of the robot and the performance factors of the robot. The performance factors include, for example, information about how fast the robot is and how fast the robot can be started.

In a virtualisation the platform on which the virtualisations are run, is called a host operating system and the operating system in a virtualised environment is called a guest operating system. One of the biggest benefits of the virtualisation is that a software can be run on a different hardware, but the software does not know that the hardware is different. The virtualisation makes it possible to implement many different types of software on the same hardware. Through the virtualisation, a better utilisation rate for a bare metal servers are accomplished and a hardware related configuration is not needed meaning that an automation can be utilised with creating new virtualised instances. [1]

A virtualisation solves various problems known in a software engineering and in communication networks. Many networks, like bare metal servers, are idle most of the time, but with the virtualisation the utilisation rate of the resources can be higher. The resource utilisation rate is an important aspect with containers and virtual machines as well: The same hardware can host multiple containers and virtual machines simultaneously which means that there are a better utilisation rate with the hardware resources when multiple virtualised environments can use them simultaneously. [1]

The virtualisation enables a better interoperability between different platforms. In today's world, a growing number of devices are connected to the networks which makes moving a software like a data an advantage. With virtualisation the software does not have to worry about an underlying hardware. Different processor architectures need a different kind of instruction sets for a software to work properly. For a virtualised system the hardware is virtualised, and it looks always the same for the virtualised system regardless of the actual underlying hardware. Thus, virtualising a system gives a greater level of an abstraction since the virtualised system is not restricted to a certain processor architecture. The virtualisation enables creating a software that runs properly anywhere, because the virtualisation takes care of translating an instruction set to a form which the underlying hardware can understand. [2]

The virtualisation began in the late 1960's and in the early 1970's when companies used a hardware which was sitting most of the time idle. IBM started researching ways to have a better utilisation rate through a resource sharing in the time domain. When IBM released a new model using time-shared resources the costs of the computing resources dropped drastically making it possible for more people and companies to use the computing resources. The benefits of the virtualisation have not changed since and today's data centres use virtualisation to have a better utilisation rate for a hardware meaning less an actual hardware for reducing costs of the computing resources of a single end user. [3]

2.1 A virtual machine

A virtual machine (VM) is a concept which is used in many different meanings. The different meanings of a virtual machine might be confusing because a virtual machine can mean on one hand a complete machine where programs can be used just like with an actual computer or on the other hand a virtual machine can mean an environment where some specific task can be executed. As the latter one the concept of a virtual machine was firstly introduced to a general public of developers in 1995 through the Java virtual machine (JVM) when Sun Microsystems released the first Java programming language. The JVM is used to translate a Java code into a bytecode which then can be executed platform independently [4]. Since virtual machines have spread across digital platforms and today virtual machines are used almost on every software related industry:

1. Almost all modern programming languages use virtual machines to execute a code, for example JavaScript Engine in a browser or an Android Java Virtual Machine in an Android operating system. [4]
2. In a software development to produce stable and reproduceable environments.
3. Security features, for example sandboxing, make a virtual machine a perfect way to analyse and test malicious items.

The virtual machines can be roughly divided into four different categories according how specific the details of the implementation are. These categories are not strict and different types might overlap in an actual virtual machine design. The four different types of virtual machines from the most detailed to the most abstract are:

1. A full instruction set architecture (ISA) virtual machine
2. An application binary interface (ABI) virtual machine
3. A virtual ISA virtual machine
4. A language virtual machine

An ISA virtual machine is the most detailed version of a virtual machine. It expresses a full virtualisation of an actual computer. All the logics from an actual computer are implemented and the user cannot tell the difference between an ISA virtual machine and an actual computer. There are multiple providers for this kind of virtual machines for example VirtualBox and QEMU. The ISA virtual machines are what usually are meant when a term virtual machine is used. [4]

An ABI is an interface between binary programs which defines how a data or computational routines can be accessed using a machine code which is a low level and a platform dependent code. An ABI virtual machine has a higher level of an abstraction when compared to an ISA virtual machine because an ABI virtual machine does not virtualise a complete computer but only the parts needed to provide an application binary interface for applications. An ABI virtual machine can be used to emulate an interface for applications to run on a system parallel with applications using a native ABI. An ABI virtual machine is more of an emulation than an actual virtualisation because it emulates a processor architecture which can be used to run applications and then transforms the instruction set to be usable on an actual underlying processor. [4]

A virtual ISA virtual machine only virtualises higher level of an ISA semantics and only the parts needed by a runtime engine to run an application. A virtual ISA is used so certain applications made for a virtual ISA can be executed. For example, the JVM which compiles a Java code into a bytecode which then can be run in the different environments. A virtual ISA is a lightweight version of an ISA virtual machine since not all the semantics are implemented except only the necessary parts to get some specific task done. [4]

A language virtual machine is just a runtime engine which translates a guest language into a form which can be understood by an underlying hardware. A language virtual machine is usually for the higher level coding languages, hence they also take care of a memory management and other similar functionalities abstracted by the language. [4]

Hence when virtual machines are mentioned it means an ISA virtual machine. A virtual machine is a completely isolated system that works on top of a host operating system. Operating system in a virtual machine is called a guest operating system and the platform on which it runs is called a host operating system. A guest operating system is connected via a hypervisor into a host operating system. A hypervisor of a virtual machine is an extra software layer between virtualised system and a real machine. The layer prevents hardware resources related constraints and enables us to run any kind of a system inside a virtual machine. [2]

2.1.1 A system virtual machine

A system virtual machine is what usually is meant when we speak of a virtual machine. The first two types of virtual machines, an ISA virtual machine and an ABI virtual machine, can be classified as a system virtual machine and the last two types, a virtual ISA virtual machine and a language virtual machine, can be classified as a process virtual machine. A system virtual machine has a complete system inside the virtual machine, and it mimics an actual physical machine. The Figure 1 presents two system virtual machines on top of a host operating system.

System virtual machines were invented in the 1960s and in the 1970s when computer hardware was expensive and there was a need for multiple users to use the same hardware simultaneously for different tasks. Early virtual machines enabled isolating users from each other and running programs and different operating systems simultaneously without interrupting other users. When the price of a computer hardware reduced, the applications for the virtual machines changed. Currently one of the most important applications for the system virtual machines is an isolation. The isolation is needed in big data centres where multiple completely different systems run on a single server. The isolation gives users a great advantage in the security because, if a virtual machine gets compromised, it does not affect to other virtual machines and systems running on the same hardware. [2]

2.1.2 A process virtual machine

A process virtual machine is a platform virtualised only for a certain process. A process virtual machine is created when a process starts, and it is terminated when the process terminates. A process virtual machine is often used in systems where there are multiple users simultaneously. Each user gets their own virtual machine and they get the feeling of having a machine just for themselves. A multiuser environment is the easiest way to implement an access for multiple people to a certain program which is usually used rarely and takes a great amount of a disk space. [2]

A virtual ISA is a process virtual machine because its only purpose is to do one task. A virtual ISA provides an instruction set and an execution model for a virtual machine. Because of a virtual ISA providing an instruction set for a virtual machine it can be seen as a virtual language making it a compilation target. [4]

Virtual language can be thought to be a compilation target for the actual language which is a programming language used by the developers to write programs. Virtual languages are usually not human readable, like a bytecode, which a computer understands but it is hard or impossible for humans to read. There are also exceptions like the JavaScript which can be thought to be a virtual language for the other programming languages. If a language can be compiled into the JavaScript it can be run in any environment providing the JavaScript engine making the Java-Script a virtual language to the actual language being compiled. [4]

In a browser a virtual machine takes care of running the JavaScript enabling responsive content on a web page. Every modern browser supports the JavaScript making it de facto of a web development and meaning that every language that can be translated into a JavaScript code can be executed in a browser. It would not be possible to run multiple different languages in a browser without a virtual machine translating the code into a JavaScript code. The virtual machine translating the code is a language virtual machine and it is an integral part of a browser meaning the virtual machine works smoothly with the browser. Smooth work is needed; page loads must be as fast as possible since people expect a real time experience when using the Internet.

2.1.3 A hypervisor

A hypervisor takes care of the interaction between a virtual machine and the underlying hardware by managing all the resources. When a virtual machine tries to interact with the hardware, for example when writing into a disk, the hypervisor checks the operation and then performs the operation for the virtual machine. The virtual machine cannot see the hypervisor working between the actual hardware and the virtual machine, making the virtual machine process actions as it would perform operations directly without the intermediary layer. [2]

Hypervisors can be divided into two classes according to their usage:

1. A native or a bare metal hypervisor
2. A hosted hypervisor

A native or a bare metal hypervisor run on the hardware of the host system and directly controls the hardware. The guest operating system is then run on the second layer on top of the hypervisor. Whereas hosted hypervisors provide a new software layer on top of a host operating system the third software layer is added in a form of a guest operating system. Both hypervisor classes are pictured in Figure 1 but the difference is that a native hypervisor is parallel with a host operating system when a hosted hypervisor sits on top of the host operating system. [5]

A hosted hypervisor provides more hardware compatibility, but it cannot access the hardware directly but uses an underlying host operating system and its drivers for the hardware access. The additional layer of the software adds to a resource overhead making the performance of a hosted hypervisor virtual machine poorer than with a native hypervisor. A native hypervisor controls directly the hardware below the hypervisor giving it a better performance, scalability and stability compared to a hosted hypervisor. A better performance is achieved by directly controlling the hardware reducing the time taken to perform I/O operations. A better scalability and stability are achieved because the only limiting factor is the underlying hardware. The only way to crash the virtual machine having a native hypervisor is because of the hypervisor or the virtual machine related errors. A flaw is caused by abovementioned limitations with the hardware compatibility. [4]

As hypervisors can be implemented into the different layers and interaction with the layers below happens through different methods, there are many different hypervisor providers today, Among the most popular ones are VMware, VirtualBox, KVM and Hyper-V.

1. The Hyper-V is the Microsoft's own virtualisation environment which can create virtual machines on the x86-64 Windows systems [6].
2. Kernel-based virtual machine (KVM) is a Linux equivalent for the Hyper-V and it can be run on an x86 hardware on Linux. [7]
3. The hypervisor of the VMware can be run both on the Windows and Linux environments even though it is based on a Linux kernel [8].
4. The VirtualBox is a hosted hypervisor and it can be used on operating systems such as Linux, Windows, Mac and Solaris. [5, 9]

A virtual machine does not know that it is virtualised since all the hardware is emulated for it by the hypervisor. A virtual machine does not run only a complete copy of an operating system but also a copy of all the hardware the virtual machine needs. Because of these copies existing, the virtual machine might quickly reserve notable amount of random-access memory (RAM) and central processing unit (CPU) resources from a host operating system. It is impossible to configure virtual machines to reserve computational resources dynamically. A virtual machine must have resources allocated for its usage prior launching. Thus, it means that even not all the resources are used by the virtual machine, they cannot be utilised by anyone else on the same hardware, either.

An isolation is one of the biggest advantages in using virtual machines. The isolation is provided by the hypervisor [10]. The structure of a virtual machine on top of a host operating system is shown in Figure 1. The hypervisor makes it possible to run multiple virtual machines on the same host because the hypervisor provides own kernel for each virtual machine.

In Figure 1 a virtual machine is divided into pieces. A virtual machine consists of multiple layers: an operating system (OS) on the first layer, binaries and libraries on top of the first layer and applications on the last layer. An operating system is connected to the hypervisor, which provides the kernel for the virtual machine. On top of the operating system there are binaries and libraries needed for the applications to be run. Also binaries and libraries needed by the operating system are located here. On top of all these, there are applications. For the user this kind of setup looks just like using a physical computer. [2]

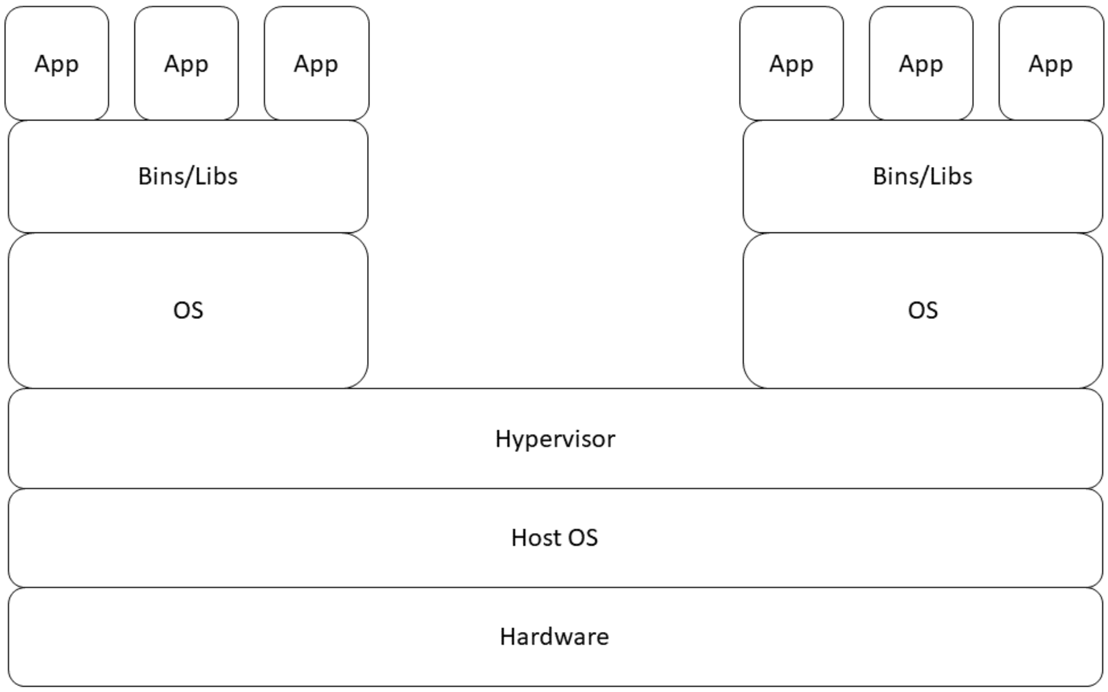

Figure 1. Basic setup of two virtual machines on top of a host operating system. Complete set containing an OS, binaries and libraries and applications are parts of a single virtual machine.

2.2 A container

A container is an operating-system-level virtualisation. Containers access a kernel of the host operating system. By accessing the kernel, a container can be more lightweight than a virtual machine. Accessing the kernel also enables a much quicker start-up compared to a virtual machine, since a container is more like a program whereas a virtual machine needs multiple processes to be spawned in order to start.

A container technology takes advantage of a Linux chroot process isolation. The chroot is a Linux command for running programs in a mode where they see a specified folder as their root folder [11]. A regular program sees all the resources in the operating system but the chroot lets the program to see only the resources under the root folder which the chroot has created. Programs in the containers can see only the resources inside the containers and cannot access any of the resources of a host operating system. The usage of the chroot means telling for the operating system that a specific folder is its root folder. From the root folder it is not possible to go backwards and this way we provide the isolation for a certain process.

The Figure 2 presents a structure of several containers on top of a host system. A container engine means for example Docker daemon on Unix system or Docker engine which runs on top of the Hyper-V on the Windows. A container engine is a software layer which makes operating-system-level virtualisation for the container. Multiple containers can share binaries and libraries making the resulting container instances smaller than the virtual machines and compact to deploy. [8]

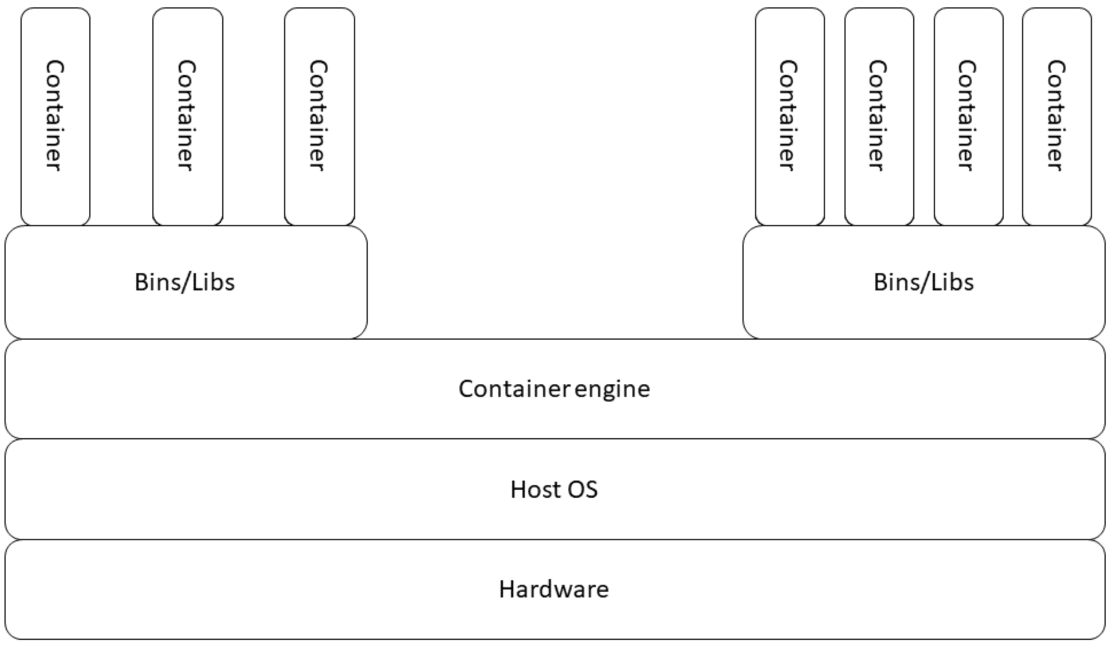

Figure 2. A container system where multiple containers are run on top of the container engine. Containers can share binaries and libraries they need to make a single container more lightweight.

2.2.1 Container providers

Linux containers (LXC) is a Linux tool for managing containers. LXC allows different isolated Linux systems to be run in a single Linux kernel [12]. LXC uses the Linux kernel features for containing users in an isolated environment called chroot jail. Chroot and cgroups are a base for isolating environments in the Linux. LXC introduces two types of containers, privileged and unprivileged [13]. From the privileged containers it is possible to escape because the container user ID (UID) 0 is mapped to the host's UID 0. A mandatory access control, for example apparmor or selinux, are only prevention of escaping from the privileged container [13]. Privileged containers are not safe to use as a root user because of the possibility to escape from the container. If a container gets infected and an attacker can escape from the container then the whole host operating system is compromised. [8, 13]

Unprivileged containers are safer because of choices in design. The container UID 0 is mapped to some unprivileged user on the host. This way if a container gets compromised and an attacker could escape from a container, the only extra rights the attacker would have are those the unprivileged user has. A security model does not enforce usage of security capabilities needed for a privileged user, but they can be used to add an extra layer of security. The additional layer of security is more of a kernel security issue than a usual security issue like escaping from a container. The kernel security issues are not due to LXC issues, but they are due to kernel bugs and should not be considered even when container security is discussed. Privileged and unprivileged containers can be considered mutually secure but in case of breach the unprivileged container is a safer choice from the host operating system point of view. [13]

The Rkt is a Linux based container runtime environment. The project Rkt is often referenced as Rocket. The Rocket has a core unit called *pod* which can contain one or more applications. The Rocket does not use a central daemon for executing a pod instead each pod is executed in a self-contained, isolated environment. The Rocket implements the app container but can also execute other container images for example Docker images. [14]

The Rocket has a different approach in process handling compared to the Docker. The Docker has a daemon running but the Rocket starts a new process when a container is started. Because of this all new processes are subprocesses to the initial process whereas the Docker runs its processes separately. Compared to the Docker, the Rocket is more robust since it does not use daemons. Failure in daemon might lead to the broken containers because in a Docker case all containers would fail if the Docker daemon fails. [15]

The Docker container was first developed for a Linux environment but in 2014 the Windows announced support for the Docker engine on a Windows Server and in 2016 Microsoft announced support for the Windows 10 with the Hyper-V [16, 17]. The Docker support for the Windows made it possible to deploy Docker containers into the Windows Azure cloud service which has helped the Docker to become the de facto container technology. [17]

The Docker container consists of a thin read and write (R/W) layer and below that layer exists the image which contains an operating system and applications. The thin R/W layer is for all the modifications made into running container. When a container is deleted only the thin R/W layer is deleted but the image stays unmodified and from that image new containers can be deployed. The R/W layer grows every time the container is used and something new is installed this is because of copy-on-write (CoW) functionality. [18]

Copy-on-write is used in the virtualised environments since it saves memory resources. With CoW there is no need to copy everything into a new memory address space because old resources can be shared with the processes using the data. Only when a process wants to write something new to the existing data a new copy is made. When the new process uses existing memory space there is no need to use so much memory hence it is not necessary to copy all existing data into new memory block during every process call. [19]

2.2.2 Differences between container providers

All above container providers have similar properties, but there are some differences as shown in Table 1. The most notable difference between them is that only the Docker is supported by the AWS and the Azure. Thus, for now we will not have any other option but to use a Docker container to deploy into a cloud service. The Docker is also the only container provider which supports other guest operating systems than Linux. [15, 18, 20]

Other differences between the container providers are quite small and discussing these minor differences in detail would require its own research hence we will not study those differences more closely in this thesis. A Docker container is based on an LXC container system meaning that the Docker containers have same workflows and methods inside the Docker engine which runs a container when compared to LXC container. The Rocket is developed with different approach compared to LXC and Docker. The Rocket is based on pods which are designed to run a single application where Docker and LXC can run an entire system container more easily because of design choices. [12, 14, 18]

The configuration of a container with the container providers takes place through a configuration file when an image is built. A configuration file has all the information a container should contain when it is ready and what it should look like in a deployment environment. [12, 15, 18]

An isolation is essential reason to use a container and the technologies used to implement isolation has effect on the security features of a container. All container providers use Unix kernel-based methods such as chroot and namespaces to provide isolation. The isolation methods are similar because the development history and point of origin of all the container providers are in the Unix world. [12, 14, 18]

A default configuration for a container consists of a minimum number of rights to keep the container as secure as possible. All container providers offer the possibility to give the root privileges for the container. The root privileges enable the container to perform any task given, but also enable breaking the isolation and interacting with an underlying system. Adding only certain capabilities needed by the container is more secure than giving all the possible privileges; this is also called a principle of the least privilege [21].

Table 1. Differences between container providers

	LXC	Rocket	Docker
Configuration	.conf file	.json files	Dockerfile
Launching process	Creates a new process	Always a new process which is a subprocess to initial process	A new process under docker daemon
Isolation	Kernel-based, uses e.g. chroot and namespaces	Kernel-based, uses e.g. chroot and namespaces	Kernel-based, uses e.g. chroot and namespaces
Capabilities	Privileged and unprivileged containers. The isolation of privileged containers can be broken more easily than an unprivileged container	Capabilities can be added/removed. Can be given privileges which break isolation	Capabilities can be added/removed. Can be given privileges which break isolation
Container type	System container	Application container	Application container
Supported guest OS	Linux	Linux	Linux, Windows, MacOS
Support in cloud services	—	—	AWS, Azure

Adding only certain capabilities for the container is supported by the Rocket and Docker. LXC provides either only an unprivileged container which have none of the extra privileges or a privileged container having all the privileges. LXC is meant to be used as a system container which explains the lesser modification options. A system container would be quite useless without privileges, because the operating system inside a container could not function properly on all tasks. [15, 18, 20]

2.2.3 An application container

An application container holds only one application and it only serves for one purpose. An example of an application container could be a Node.js server running inside a container. The Node.js server would be the only application running inside the container and the container would only have certain ports open from which the server and networks outside the container can communicate. Application containers are useful for example in a full-stack development: a backend and a frontend can run in their respective containers and a database is in some permanent location where data can be read and written by the backend.

Application containers can be orchestrated in cloud services. Every container can contain only one application, and multiple containers communicate with each other over a network. Bigger applications can consist of multiple application containers where each application container has only single task to do. A data will be transferred between the containers via an internal network or over the Internet using open application programming interface (API) endpoints.

Application containers are designed for an agile development where one part of a system can be deployed when the development is ready. An application container also makes deploying systems more like writing a code. Everything is modular and parts of the system can be developed without having to alter the whole system. Testing of the system is also easier because a small part of a complete system can be tested at a time and verified that it is working as expected. Through a modular structure a system becomes more robust and debugging problems will be easier because a problem area can be narrowed down. After narrowing down the problem the fix is easier to deploy.

A modern browser can support a container technology too. Mozilla Firefox enables user to use a browser container which keeps all the specified content inside a container [22]. The browser container is an application container because inside a container part of an application runs isolated from the rest of a browser. The containers have their own resources allocated making the browser container also a security hardening feature.

2.2.4 A system container

A system container contains a full operating system image. A system container cannot run different operating system compared to the host system, for example a Linux guest cannot be run inside a container on a Windows host. Operating system agnostic nature of a system container reduces possible deploying scenarios. A system container is useful in a situation where a virtual machine-like functionality is needed but a more lightweight solution is preferred. A system container is more lightweight than a virtual machine because a system container does not contain virtualised hardware resources and a kernel. [23, 24]

A system container does not have as many use cases as an application container, but they have their place. A system container is an ideal solution when a complete system needs to be deployed but there is a limited amount of space available. Since a system container is more lightweight than a virtual machine, hence more system containers than virtual machines can be fitted into same space. A start-up time of a container is less than with a virtual machine so if fast response times are needed the system container is a good choice. For example, an organisation having multiple users sharing virtualised environments would benefit of a quicker start-up time of the system container compared to a virtual machine.

A system container can run only a guest operating system which is the same as a host operating system. Hence the benefit of a smaller deploying size is lost if different operating systems are needed making a virtual machine a better solution.

2.3 Comparison between a virtual machine and a container

A virtual machine is a complete copy of a functional operating system where container is designed to do smaller, individual tasks. A container is designed to be expendable and reusable which means there is no sense to save data inside a container. A virtual machine can save its state after shutting down therefore it works just like a physical machine and saving data inside a virtual machine is justified. In Table 2 previous topics are gathered into a single table for easier comparison between a virtual machine and a container. Many properties are quite similar in both technologies but from the table we can easily find which platform would be better for the robot to function.

Figure 3 shows a decision tree of which implementation environment should be used for a software robot. First, it is necessary to identify what programs the robot needs to use. It is important to remember that, for example, Excel files will be opened and read programmatically so there is no need for the robot to use a graphic user interface (GUI).

If an application is completely browser based the operating system does not have any effect on choosing the technology hence it is possible to use both a container and a virtual machine for implementation. If the robot needs to use a desktop application, then it is necessary to identify on which operating systems the application can run. If the application can be run only on the Windows, a virtual machine is the only possible way to implement the robot since on a Windows container it is not possible to have a desktop for the robot. If the application can be run with Linux, then both a virtual machine and a container are possible methods to implement the robot.

There are multiple things to be considered when choosing a proper implementation method for the robot and it is necessary to understand the workflow of the robot completely. Many things can be done using APIs or programmatically without using GUI making the robot more effective compared to human. The most important thing is to understand what is wanted to be achieved, not how it has been done previously because a robot might do it completely differently in order to be efficient.

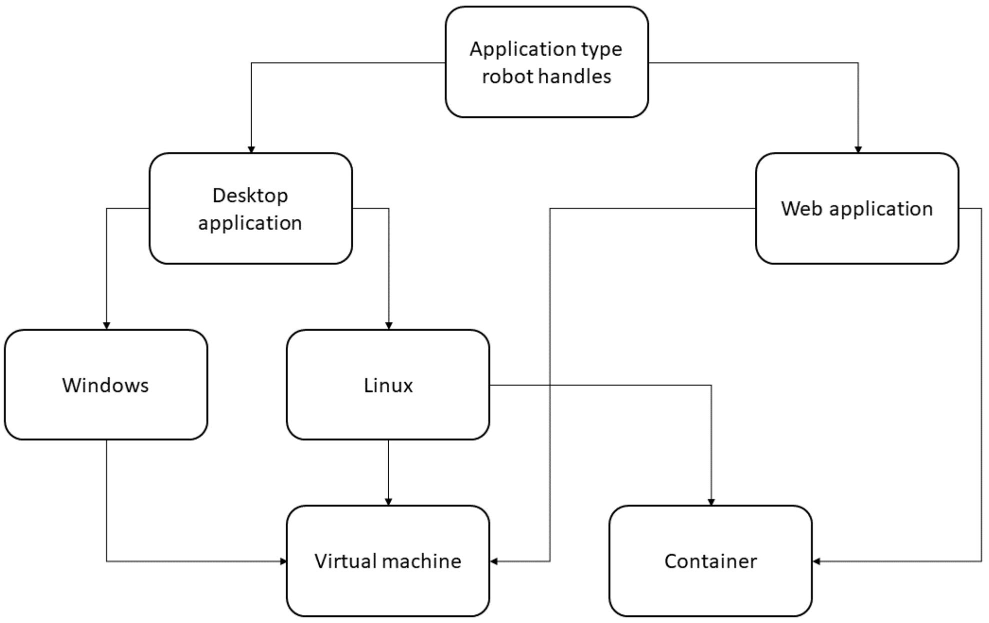

Figure 3. A decision tree of possible technologies for implementing a software robot.

2.3.1 Size

A container image requires far less space compared to a virtual machine, since a virtual machine runs a complete operating system inside whereas a container runs only libraries and binaries which it needs to be functional. A generic reusable software robot would have only the Sikuli framework inside a container along the necessary libraries and binaries making it a lot smaller when compared to a virtual machine containing a robot.

In usage container allocate only the amount of memory needed for the main process which makes it smaller than a virtual machine. As for a generic software robot, an unlimited[1] number of containers could run on top of single stack containing a container engine, binaries and libraries as seen in Figure 2. This means that it is necessary to allocate resources only once for everything else except the containers holding an application along application specific binaries and libraries. A single robot container uses around 85 *MB*, ergo they can easily be deployed with minimal costs.

A container is more lightweight compared to a virtual machine since the container shares a kernel with the host operating system whereas a virtual machine contains both a kernel and hardware virtualisation inside an image. An Ubuntu 18.04 Docker container is 116 *MB* in size, when a virtual machine with the same operating system is much bigger. An installation media of an Ubuntu 18.04 is 1.9 *GB* in size and after installation the operating system will take 8.6 *GB* [25]. Here the amount of space needed is quite different, the container being 15.5 times smaller in size than the similar virtual machine. The size of the virtual machine is accumulated with the size taken by a virtualised hardware, a kernel and other necessities making the resulting image as big as over 10 *GB*. Even the resulted container image is around 3 *GB* in size the space

[1] Number of resources would limit number of robots but in a cloud environment number of resources are presumed to be infinite.

taken by a single container is only abovementioned 85 *MB* meaning that for the size of a single virtual machine over hundred containers can be deployed.

2.3.2 Deployment

Both containers and virtual machines dispose probably the biggest problem in the software development: the code that works on the machine of the developer, is not functioning properly on the production environment. With a virtualised environment developer can tweak every single detail to function perfectly and be sure that during the deployment into the production environment there will not be environment related issues.

With containers the portable deployments are made easy since a functional image is deployed directly to the production environment. In a container all necessary libraries are moved within the image and in production we can be sure, that the system works exactly in the same way as when it was developed [26]. A virtual machine deployment is guaranteed to work always since a virtual machine contains a full copy of the hardware, operating system and all other files in a working computer. A virtual machine image is significantly bigger as the image contains all abovementioned items resulting the deployment to take a longer time than with the container. Virtual machines have their place in the software development but with a reusable software robot time taken with deploying a new instance is critical and the fastest possible solution is the best.

The development of new features is also easier with a virtualised environment since any developer can have instantly the correct environment setup when they run a container or a virtual machine. This reduces time needed to solve problems which are not related to the program but rather to the development environment. The time taken to create a new environment depends of the initial state. If an operating system image is at hand, a new virtual machine can be created in a few minutes. If the operating system image needs to be downloaded, the setup time increases significantly. With a container-based solution a creation time of a new container image is fixed. A new image can be created in around 20 minutes. Time is quite a long since there are so many different libraries and binaries which need to be downloaded and installed.

2.3.3 Isolation

Already in year 2015 Docker containers had achieved the same level of the isolation as virtual machines [26]. A container isolation is one important part in choosing a container. A container isolation means that no data will be leaked between the host operating system and the container even they share the kernel. Because of the isolation a compromised container shall not have any effect to the host operating system and the compromised container can be shut down and be replaced easily with a clean one.

A container and a virtual machine are both well isolated, but the virtual machine has a hypervisor which controls the traffic between the host and the guest operating system making it easier to break a container isolation when compared to breaking a virtual machine isolation. The container isolation can be broken by a user with only choosing a set of unsafe configurations. For example, an X window system (X11) sharing the host operating system's X11 socket with the container breaks the isolation between the container and the host operating system allowing to escape from the container [27]. If the container is properly configured the isolation is by default intact and an attacker needs to find another way to break the isolation.

The Docker engine produces a container isolation. When Docker engine is started new namespaces and control groups are provided for the container. Use of namespaces provides an isolation that is, it isolates the container from seeing or affecting the other processes in a host system or in the other containers. Namespaces are provided by a kernel which partitions kernel resources to logically isolated blocks that cannot interfere with each other. [28]

The container isolation also provides own network stacks for each container. A container having an own network stack provides an isolation between the containers. With an own network stack, a container cannot connect to another container in a same network. The ultimate isolation like this can be an unwanted feature when developing applications consisting of multiple containers. When multiple containers need to interact with each other, the host system can be configured to enable interaction between the container interfaces. A network architecture of multiple containers is exactly like multiple actual computers connected to common Ethernet switch. All containers can see each other on the network but the data is transferred only between the containers needing it.

2.3.4 Security

Escaping from a virtual machine or a container means breaking the isolation of a virtualised environment. Breaking out from a virtual machine means that an attacker can interact with a hypervisor. In addition, because of the structure of a hypervisor virtualisation all the other virtual machines on the same hypervisor are compromised too. Breaking out of a container means similar things. When an attacker breaks out of a container, the attacker can interact with an underlying container engine and break into other containers running on the same engine. Through a hypervisor and a container engine the attacker could be able to interact also with the underlying hardware making it possible to wreak havoc for the hardware creating possibly great costs.

If a container gets compromised, it will not affect the other containers because of the isolation. On a basic setup, it is not possible to move from one container to another container not to mention to a container engine or a host operating system beneath the container engine. Some services like an X window server might break the isolation of the container if a window is shared between the container and the host [27]. Breaking the isolation is since the isolation is already broken by user when sharing resources between the host and the container. If no resources are shared, breaking the isolation is a much more difficult task and needs some other vulnerability to be exploited which is not anymore related to the security of the container but to the security of the vulnerable software or an underlying kernel.

A kernel is an important part of the security. The kernel is a software layer between all applications and a hardware. The kernel is started among first things after the bootloader has finished. The kernel has some security features implemented directly into. The kernel manages, for example, the users and their credentials, access to the filesystem and drivers including the network stack. The kernel security features are used in a container implementations for example, using the chroot means using a Linux kernel feature for creating an isolated environment. [11, 29]

In 2015 at Docker version 1.8 security was added remarkably. In Docker 1.8 Docker Content Trust (DCT) was added. DCT uses singing keys to sign images. With this signature the image can be verified to be the one it claims to be. DCT framework enables blocking of keys from known compromised sources. With DCT a developer can be sure that along a container image no malware is downloaded if the image is from a legitimate publisher. [30]

Daemons are a program running background process in a Unix-like system and a user cannot interact with the daemon. In a Windows environment program with a daemon functionality is called a service. Daemons run directly under the root process initialised on start-up. [31]

Docker uses a daemon to start process. User can interact with a Docker daemon when starting the daemon. A Docker daemon is a process which manages all the containers running on a platform. A Docker daemon listens for a Docker engine to perform requests by the containers through the Docker engine. A Docker daemon can be accessed through network, but it is not recommended since it makes all containers running on the daemon vulnerable. The Docker launches every new container as a new process under the Docker daemon. This gives containers more stability as if a container crashes it does not impact on other containers running on the daemon. The downside is that if the daemon dies unexpectedly, all the container processes are adopted by the initial process, but the Docker engine cannot interact correctly with the initial process causing all containers to crash. The Docker daemon opened to a network is accessed by default with unencrypted and unauthenticated connections meaning that anyone can access containers through a network and breach the containers. Opening the daemon to a network is advised to be considered thoroughly and if the daemon is opened it should be secured using a secure proxy or a HTTPS (hypertext transfer protocol secure) encrypted socket. [32]

An interaction between containers forces opening ports from a container and furthermore opening the ports exposes more attack surface. The interaction could be secured a little by creating a container cluster where containers are in their own virtual network. The virtual network can have necessary ports open to the Internet through a gateway but the total amount of open ports to the Internet can be reduced when compared to all the containers being directly connected to the Internet. It can be configured hence the container can only accept incoming connections from the virtual network making only the gateway of the virtual network exposed to the Internet from the cluster. Only by breaching the gateway it would be possible for an attacker to even try to breach the containers making a container implementation more secure and less attractive target for the attackers. In Figure 4 a setup, where four containers are inside a virtual network and the virtual network has a gateway to the Internet, is represented. Inside the virtual network all the containers can interact with the gateway and with each other within limitations of the exposed ports on a single container. The gateway handles all requests to the Internet refusing all connections which are not allowed by the gateway configuration.

One form of a virtualisation is sandboxing. Sandboxing is used in a malware testing since from a sandbox the malware cannot contaminate the whole system and the activity of the malware can be researched. Windows introduced sandbox as part of the Windows 10 Pro and Enterprise editions for the first time [33]. Sandboxing is made by creating copy of an actual system and running it in a virtual machine. Since a virtual machine is a complete copy of an actual system the behaviour of the malware or developed code can be tested safely without breaking the actual machine if something goes wrong. Sandboxing is used by many antivirus software for detecting a malicious code by observing what a malicious code tries to execute and what the code puts out. [34]

In a Mozilla Firefox browser, containers can be used for added privacy [22]. A browser container keeps data isolated from other parts of a browser. Even the initial reason for a browser container is adding privacy, the browser container also adds security. Since if a website is run inside a container it is much harder for an attacker to get access to other information on a computer and a browser of a user. If an attacker wants to interact with the rest of the browser, the attacker should first break out from a browser container.

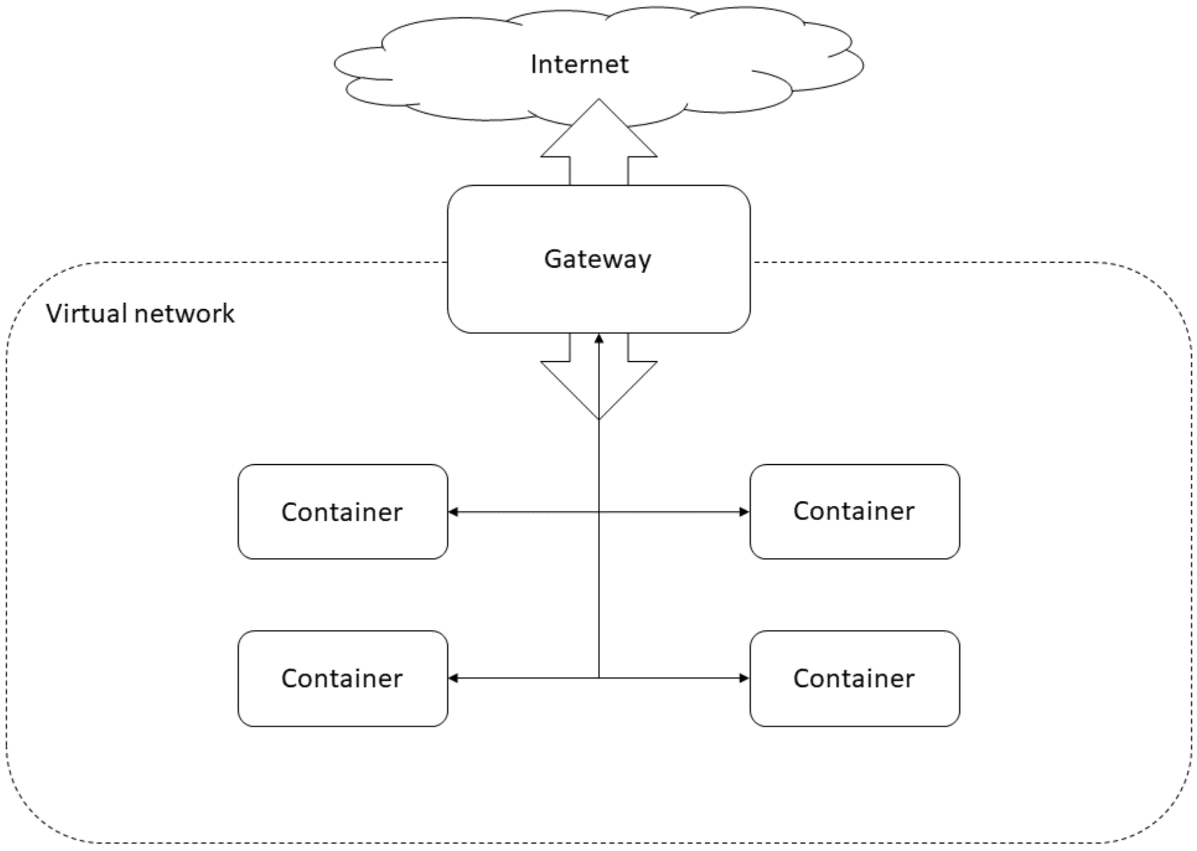

Figure 4. A virtual network containing multiple containers and the gateway to the Internet.

2.3.5 Overview

By now it is clear that a virtual machine and a container are very different. Where virtual machine hosts complete applications or systems, a container is supposed to host only services used by an application making a container smaller package which can be easily updated [35]. A virtual machine is a good solution when it is needed to run multiple applications and complex cases with the robot. A container suits better into more specific tasks where only maximum of few different applications is needed by the robot to perform tasks. If a container contains multiple applications, the container starts to grow close to the size of a virtual machine and most of the benefits of using a container are lost hence it makes no sense to use a container in the multi-application situations.

On the Table 2 a comparison is represented between the Docker container and virtual machine features which have effect on the decision of choosing a proper platform for the robot to work as efficiently as possible. The isolation is one part of security but in the case of a robot it is not as important concern as for example with a web application.

Table 2. Comparison between a Docker container and a virtual machine.

	Container	**Virtual machine**
Save data for permanent usage	No	Yes
Structure	Contains an application	Contains multiple applications
Isolation	Based on a container engine	Based on a hypervisor
Security	1. Used kernel security features 2. Docker daemon 3. Container configura-tion 4. Interaction with other containers	1. Sandboxing 2. Hypervisor 3. Configuration like in a physical computer
Deploy time of a new instance[2]	Tens of seconds	Few minutes
Reasonable number of applica-tions	One	No limit
1. Size of an image 2. Size of a running in-stance	1. From tens of mega-bytes to few gigabytes 2. Tens of megabytes	1. From gigabytes to tens of gigabytes 2. From gigabytes to tens of gigabytes
Cost in AWS[3] [36]	0,0195 $/min	0,086 $/min

A container, where a robot works, is deleted after the robot has run its task and thus there is not much time for the attacker to take an advantage of a possibly breakable isolation. A container security consists of multiple factors as seen in Table 2 where all the factors add to security but can be also an independent attack vector.

Containers have more individual security artefacts since with a virtual machine a hypervisor takes care of multiple things. A kernel security features are taken care of by a hypervisor. When a new virtual machine is started a daemon is not launched, rather a hypervisor takes care of managing a virtual machine.

The smaller size of a container means a smaller price since the virtual machine and container images need to be saved into a cloud environment. The smaller size means also a faster deploy time since there is less data which needs to be transmitted. On the Table 2 the cost is different between a container and a virtual machine for the same amount of uptime since the container needs less resources to run. For example, the virtual machine with 4 *GB* of RAM would probably crash when the robot tries to use multiple tabs in a browser since modern browsers use quite a lot of memory, for example a Chrome browser is known of its ability to eat out all the memory from the system [37]. The memory consumption is not a problem if the system can

[2] Time measured in AWS with Nginx Docker image compared to virtual machine containing Nginx server.
[3] Container: t3a.medium with 2 CPUs and 4 *GB* of RAM. VM: t3a.xlarge with 2 CPUs and 8 *GB* of RAM.

have access to the RAM resources whenever needed but if the RAM is constantly used completely the system is slowed down causing problems for the robot to perform.

As a result, the software robot will be implemented into a container. A container is more lightweight meaning less costs. It is also made easy to deploy to the modern cloud services. Also deploying a new instance is faster with a container when compared to a virtual machine. Only thing which needs more configuring with a container when compared to a virtual machine is saving the data. Since a container is destroyed after finishing its tasks the data needs to be saved somewhere else than in the container. Mounting a network drive to a container filesystem is a great way to implement a persistent data storage into the container. Mounting a network drive to a container filesystem is not possible without added capabilities for the container. To mount a network drive to a container filesystem two capabilities are needed: *SYS_ADMIN* and *DAC_READ_SEARCH.*

2.4 Multiple robots on a same hardware

One of the most important questions when making a reusable software robot is if it is possible to run multiple robots simultaneously on the same hardware. If it is not possible to run multiple robots simultaneously there is only limited amount of use cases for such software robot. A container and a virtual machine enable having multiple robots on top of same hardware, and they will not be affected by each other, since they have their own isolated systems. Because of the isolation, a container and a virtual machine does not know of any other systems running on top of the same hardware which makes them perfect for deploying the robot into a cloud service.

On the same hardware every robot has their own virtualised environment. In a virtual machine situation implementation of a robot onto the hardware is straightforward. The virtual machine is a complete copy of an actual computer, so all robot related software is just implemented into a virtual machine, and the robot can be instantly deployed.

With a container-based implementation the situation is somewhat different. The container has a minimal number of an excessive software when created meaning there is nothing else but the absolute minimum to run a basic set of commands. To implement a robot into the container it is necessary to create a screen for the robot, add all libraries needed by the Sikuli framework and add the robot related software. Some of the robot related software is version dependant meaning that it is not possible to always download the latest version of a software, rather the exact version needs to be used.

The main concern in a case where multiple robots are run on top of the same hardware is the resources needed by a robot. The robot needs some CPU power but mostly RAM to perform its tasks efficiently. If the robot uses all its RAM, it becomes unstable. With too little RAM the loading of a screen content takes too much time and the workflow of the robot is interrupted causing the robot to crash. By giving enough resources for the container running the robot it is possible to be sure that the robot will not crash because of the resources. Eliminating crashing because of a loss of resources makes the robot implementation more robust and enables user to trust the ability of the robot to finish tasks without a constant supervision. Using a cloud environment basically removes the hardware resource related problems since in a cloud environment more resources can be added easily.

Different robot tasks need different amounts of resources and hence it is payed for what is used in a cloud environment it is smart to only use minimum amount of resources needed to perform a task. To have a robot running without problems it is necessary to be able to scale resources when needed.

2.5 Scaling of the resources

Docker applications in containers give developers the possibility to scale both horizontally and vertically. Both AWS and Azure support scaling of the resources. AWS and Azure also support autoscaling which enables services to scale instances to match the load [38, 39]. Scaling is more important in web applications since the load that an instance must handle might vary very much throughout day. Scaling in a use case of a generic software robots is mostly manual horizontal scaling since whenever a new task is needed to be done, it is possible to start a new container instance, essentially a new robot, which then can handle the new task. This way it is possible have as many robots as needed to work for tasks simultaneously.

A horizontal scaling means adding new instances to handle the load. In a robot case it would mean adding new robots to handle same task. The horizontal scaling is a good way to scale up or down the performance when only the number of parallel robots is needed to scale.

A vertical scaling means scaling the features of a deployed instance for example adding more memory or CPUs. The vertical scaling is relevant for a software robot since the minimum number of resources is wanted to be used in a cloud service as using resources adds costs. But when opening a browser inside a container and a robot starts to perform its task, more memory might be needed and then it is wanted to vertically scale memory for one specific robot which needs more memory to perform.

By using a container, it is possible to ease the need of resources when deploying services. On the traditional way of deploying a service, the whole monolithic codebase is deployed every time a new feature has been implemented. This takes resources to be accomplished especially if the service is wanted to have zero downtime since two instances, the new and the old, would be running simultaneously until the deployment is done. With containers each container can be deployed independently meaning that resources needed during the deployment is smaller. Scaling the resources according the needs reduces the costs since in a cloud environment it is payed only for what is used. By using automated scaling provided by cloud services it is possible to make scaling of the resources adaptive. [8]

AWS provides an autoscaling feature which is configured according to the use case. Through the configuration it is possible to decide what the maximum and minimum level of resources used are when scaling from a default setup and the circumstances where autoscaling is triggered. With autoscaling it does not matter if some service is under a heavy usage on some specific time of the day, as when the usage rate starts rising the autoscaling adds resources when configured limits exceed and after the peak is gone the resources are reduced to default values. Azure has similar autoscaling options with AWS and from a user point of view there are no difference how the services function. [38, 39]

2.6 Cloud providers

A cloud is an environment where containers can be deployed. Docker makes it possible to deploy containers with different operating systems. AWS and Azure have both provided support for Docker from 2014 [40-42]. Deploying a container instance starts with having an image of a container which is then moved into a cloud environment. From the cloud environment the image is used when a new container instance is started. When deciding which cloud provider to use it is necessary to focus on what features are necessary for a generic software robot implementation. On the Table 3 a necessary set of features is collected for comparison. It can be seen from the Table 3 that both AWS and Azure have similar set of features. Choosing between Azure and AWS should be done every time a new specific robot case emerges since some differences occur between the AWS and Azure which might affect to the chosen technology. With a generic software robot, the focus is on developing a base image for a robot environment which can then be extended because of deploying Docker image to a cloud service is easy.

Both Azure and AWS provide the possibility to run container instances in a cloud. However, during the development some problems emerged. One of the greatest problems was that Azure does not support yet the possibility to mount network drive into a container, since not all Docker capabilities are supported by Azure [43]. AWS has provided this support for all containers since September 2017 [44]. Adding or dropping capabilities for containers is an important security feature because by default containers are run on an unprivileged mode where they cannot perform most system and network related operations. If a container is run on a privileged mode it can perform operations as a root user from the point of view of the host. Compromising a privileged container could compromise an entire platform where the container is running and other containers running on the platform could be compromised too. Adding capabilities makes it possible to give extra permissions for the container to perform a single task. For example, mounting a network drive inside a container enabling the container to use data from external sources. Usage of a network drive makes it also easier to horizontally scale containers since they have mutual information source which is updated in real time.

AWS is more mature compared to Azure and it can be seen in situations where Azure has not yet implemented some functionality into their services, and this might produce problems for a developer. The maturity level of Azure will rise but it is clearly seen that the time AWS and Azure has been on the market is different and the more mature platform is received from a provider which has been longer in the market. AWS launched first service in 2004 where Azure had their first service launched four years later in 2008 [45, 46]. The 4 years gap explains most of the differences between the maturity of the cloud providers. Azure has a better support for Microsoft's products but with a container solution it is not possible to utilise this support because it is not possible to run Windows inside a container with a functional screen.

Table 3 shows that between AWS and Azure there are not many differences when an RPA solution is considered. For now, the biggest difference between the cloud providers is that AWS supports adding capabilities to containers and Azure does not. This fact makes AWS a more attractive choice to implement the software robot.

Table 3. The support for capabilities needed by an RPA solution in AWS and Azure.

	AWS	**Azure**
Launching container instance	Yes	Yes
Launching container clusters	Yes	Yes
Support for container capabilities	Yes	No
Resource scaling 1. Horizontal 2. Vertical	 1. Yes 2. Yes	 1. Yes 2. Yes

Adding capabilities means that it is possible to mount a network drive inside a container enabling larger spectre of use cases which can be realised with a generic software robot. According the comparison AWS is chosen to be the platform where the generic software robot will be implemented since AWS provides for now a wider range of possible setups with a robot.

There are other cloud providers than AWS and Azure, but these two are the biggest on the market and in general use at the software industry. Smaller providers are often focused on some specific cloud service. With AWS and Azure there are also Google, IBM and Oracle of bigger operators who provide cloud services as an *infrastructure as a service* (IaaS). Smaller cloud service providers who provide a *platform as a service* (PaaS) are for example Heroku, Salesforce App Cloud and Engine Yard.

For the implementation of a software robot IaaS is needed since it is necessary to be able to configure the underlying platform. PaaS is only a platform where applications can be deployed but the user cannot configure the platform according to their own needs rather the default pre-defined settings has to be used.

3 REUSABLE GENERIC SOFTWARE ROBOT

Next, a better method is studied to implement a reusable generic software robot, choices being a virtual machine or a container. During the study, a container proved to be a better solution because that is faster to start in a cloud and more lightweight than a virtual machine which means it is cheaper in use. A cloud platform was chosen to be the AWS because it has a better maturity level and a better support for the features needed.

The use case with a generic software robot is somewhat different from a usual container use case since a stable environment for a robot to handle different kind of tasks is wanted. It is necessary to create an environment where a software robot can perform different tasks without making changes to the environment. A base image where the robot can perform most of its usual tasks is created and the base image is easy to modify if some specific applications are needed for the robot.

Docker was chosen to be the technology to implement the container into a cloud service since all the major cloud providers support Docker containers. Docker has an active development and it is constantly reviewed and updated when new security flaws emerge making it a good choice to use as container provider.

3.1 Sikuli

Sikuli, an open source framework is used for a software robot development. Sikuli is written with Java and it uses a Jython library which is a Python interpreter for a Java platform. Sikuli enables coding with Python but since a Jython library is used, a Java code can be added into a Python code as well. Sikuli supports other languages like JavaScript and Ruby too. The core of Sikuli consists of two parts: a *java.awt.Robot* library which controls the mouse and keyboard events and a *C++* engine based on an *OpenCV* which manages the image recognition capabilities. [47]

As for Sikuli, easy workflows can be made without coding. The integrated development environment (IDE) of Sikuli has ready shortcut buttons to execute certain tasks using only a mouse. Clicking a shortcut button translates into a Python code shown in a Sikuli IDE. Using pictures for creating workflows is an unstable practice and should be avoided whenever possible. For managing a Mozilla Firefox browser, a *Selenium* library is used, the library enables the interaction directly with a browser by using a WebDriver API. Using Sikuli and Selenium it is possible to open a browser and click on any element on the page. By coding confirmations of where the robot is during its run, it is possible to create complex workflows and manage errors making a robot more robust.

When Sikuli 1.1.4. is used, it is possible to deploy a fully functional *.jar*-packets into a container and no additional installation for the framework is needed. Deploying of the *.jar*-packets makes it possible to create a new container image from a Dockerfile. The installation of older Sikuli versions could have been automated but it would have required more configuring when compared to Sikuli 1.1.4. Less stages in creating a container means less possible points of failure. Sikuli 1.1.4. introduced a completely refined App class which takes care of starting and closing programs. A refined App class adds robustness which is important when a robot is working for a long time with a single task. [47]

3.2 The implementation of the robot

Docker container can be built with only few commands following a README.md in the appendix 2. A Dockerfile in the appendix 1 builds a complete image where the robot can be run when necessary packets are in the same folder with the Dockerfile. During an image build all necessary libraries are either downloaded or copied from a folder and after the build a user only needs to start a VNCServer and start using Sikuli. The VNCServer is needed for development so it can be seen what the robot will see when executed. The VNCServer script in a container starts both the VNCServer and the X window system. The VNCViewer, which is used to connect to VNCServer, enables to check a status of a robot while it is working without interfering the run of the robot which is useful feature during development and test runs.

The installation process is fully automated, so user needs only to start the container instance and start a bash script to initialise and start a worker. A worker is a Java based software which communicates with a control panel. A worker tells for the control panel when it is ready to get a new task and during task execution it reports to the control panel according to the rules coded into a Sikuli script.

The X window server which is part of X window system (X11) should take all human interface device (HID) events such as keyboard and mouse events, and process them sending HID events to only X clients, in other words application windows, needing to receive the events. From the Figure 5 it can be seen the place of the X window server between HID events and applications. The X window server also takes care of sending data to a screen so user can see what is happening. A robot does not need a physical screen to function, the robot needs only a functional X window server, so the robot has a virtual screen where data is drawn. The X11 was first developed to be used over the Internet, so support for the Internet protocols is needed to run the X11. [48, 49]

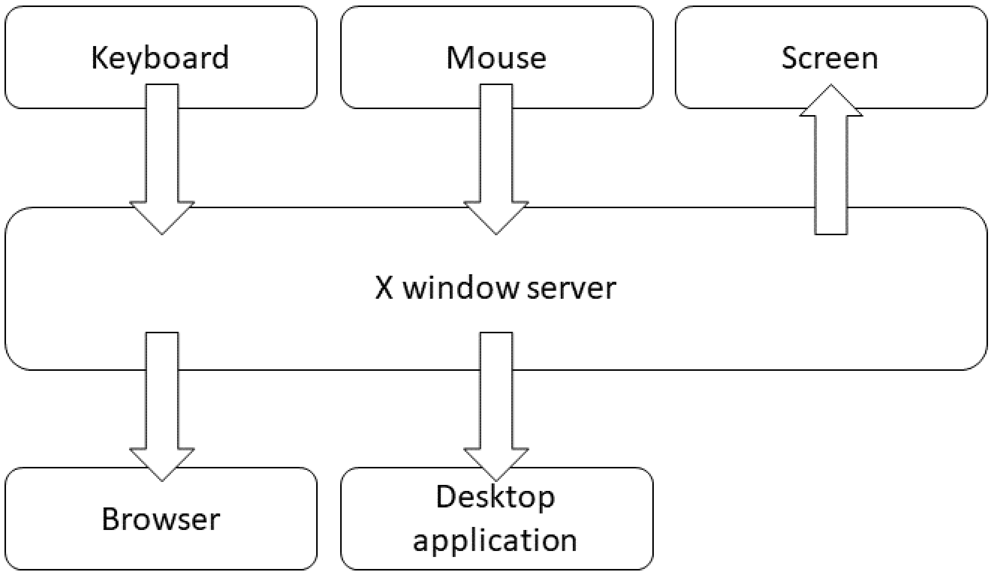

Figure 5. Data flow from HID events to applications through the X window server.

3.3 The workflow of the robot

Figure 6 depicts a general view of components used to make a robot functional. A control panel is the controlling unit which manages all robots and controls everything. The control panel manages upcoming tasks and gives the task for a worker to be done when a worker asks for a task. When a worker is idle it polls the control panel once a minute asking if it has a task. If no task is set to be started within next minute the control panel responds with a wait task for the worker. If a task is set to be started within next minute the control panel sends the task for a robot. The robot then checks if it already has a code for executing the task and if it does not have the code in question, it downloads a zip-file containing all the code files from the AWS.

After all the code files are extracted, the robot starts its task by finding a *main.sikuli* file and running the code. The robot then reports to the control panel during its work as it is configured in the code. Only the data chosen to be sent to the control panel is sent there otherwise only a start of the run and a final status of the run is logged into the control panel. In the running environment of the robot a log file is produced containing all events made by the robot. Every click and typing made by the robot are logged containing coordinates for clicks and the written text for the keyboard events. The log files are saved in the container into same folder in the container filesystem where the worker is started. After the task has finished the log files disappear if they are not transferred somewhere. It is easy to send the log files for example as an email attachment consequently possible error situations can be studied afterwards.

Known possible error situations are coded into the robot script to avoid unnecessary crashes. Sometimes, however the robot might encounter situations it cannot recover from, for example, network loss, slow loading of a network page or a pop-up window of a necessary update. A network loss and a slow page loading can be prevented but after some point it makes no sense to use a development time in such borderline cases. The regular error handling accompanied with waiting a few seconds solves most of the problem situations related to a slow page loading or occasional network losses.

In Figure 6 the RPA stack is shown, and it contains a Sikuli framework and all included libraries. In the RPA stack there will be also a Selenium library written in Python specially developed for the Sikuli framework by Vincit. The RPA stack contains also all the other Python libraries needed for the robot to execute different tasks for example including reading Excel or CSV (comma separated values) files. Few *.jar*-files are also included. The executable code can be considered as the final part of the RPA stack even if it is downloaded from the cloud into a container.

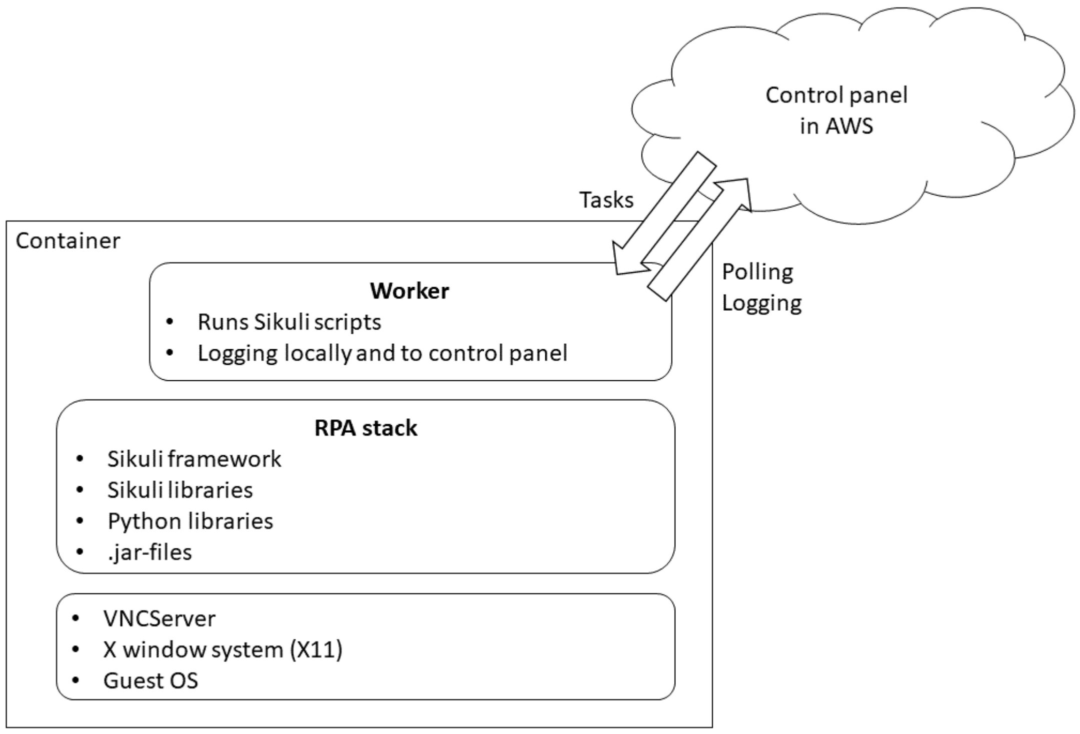

Figure 6. A general view of a high-level workflow of a robot.

3.4 Multiple robots working simultaneously

When the robot implementation is horizontally scaled it means that a new robot is created to work on the same task. A robot control panel should be used for controlling multiple robots simultaneously. The control panel orchestrates multiple robots working on a same task. Task could be, for example, processing data from the table one line at a time, every line is presented as a case in Figure 7. At first, one robot could be working on a task but then it is realised that more robots are needed to work on the task consequently the task could be finished faster. At first a robot would have worked one case at a time and another task would have been given for the robot when it has finished. When two more robots are created, the control panel gives each of them a case to handle and when one of the robots finishes its case the robot gets a new case from the control panel. The control panel manages that all the cases will be finished and none of them is processed more than once.

An orchestration is needed only when the robots use a data source which is not updated when cases are processed. On the robot cases where the data source is an Internet page the data is usually updated in real time because a list of cases changes when a case is processed. In this sense a software robot is just like human being, many people need their own lists of cases in order not to get mixed up or someone else must give then the cases, to be processed.

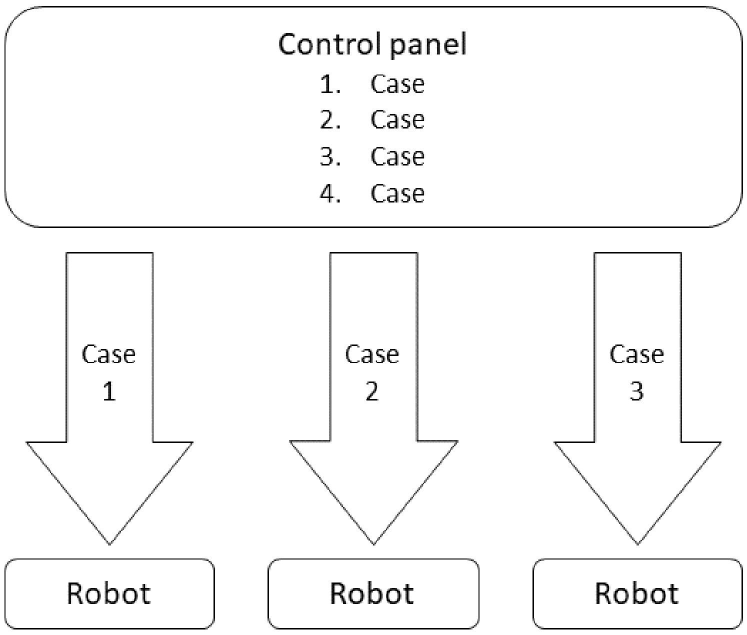

Figure 7. The control panel sharing work to multiple robots.

3.5 Costs

The costs of a software robot consist solely of the costs of using a cloud service. When running a robot in a cloud it is necessary to reflect on a type of a task the robot is working with. Vertically scaling resources for the robot does not give any faster performance once there are enough resources for the robot to function flawlessly. A horizontal scaling can be used but it does not necessarily mean better performance. If a target system is slow or only one session with same credentials is possible it makes no sense to create multiple robots to work with the same task. A single task takes always a same amount of time for a robot to perform therefore with horizontal scaling it is possible to reduce the time needed to perform task successfully, but it does not have impact on the costs. A total cost can be calculated as follows:

$$N * T * P = C \tag{1}$$

Here *N* is the number of robots, *T* is the time taken to perform a task, *P* is the price of a resources per a time unit and *C* is the cost. Now when the robot can perform a single task in constant duration meaning *N* robots need only $\frac{1}{N} * T$ units of time, expanding the equation (1) into a following form:

$$T * P = C \tag{2}$$

The equation (2) shows that costs of a software robot are quite stable and only affected by pricing models of the cloud provider and the time taken to perform tasks which in turn helps to calculate monthly costs.

3.6 Suitable robot use cases

Not everything should be done using a software robot. There are tasks that are suited very well for a software robot like repetitive tasks where decisions are made according the input data. However too many branches in a decision tree makes a software robot more prone to errors and increases the development time since every branch needs to be explicitly defined. Too many branches slow down a software robot since it must be checked always which view a robot is having before acting. If a robot is in a wrong place it can give up and return to start and try the next case or it can crash whichever is more suitable action. Slowing down a robot because of a complicated decision tree increases costs in the production since the robot has to be kept working for a longer period.

A robot makes the most cost efficient and fastest work when it has a repetitive, specific task which does not have a too complicated decision tree. Some consideration is needed when having two robots splitting the task. The output of a first robot could be used as an input for a second robot in order to decrease costs by decreasing the running time of a robot. The case of splitting the task has a risk of the first robot crashing leaving the second robot without input causing it to crash too. Some flag should be used if the task is split between two robots consequently the second robot knows when it can start working.

4 DISCUSSION

The first problem with a generic software robot is a stable environment. The robot needs a predictable environment where the robot can repeat its workflow time and again reliably. The usage of a virtual machine was the first idea but shortly the possibility to use a container emerged. The container seemed a better fit hence its lightweight design which allows fast deployments. After researching both a container and a virtual machine-based solution, it was decided to use a container for the implementation because of the abovementioned reasons.

There is literature of the differences between a virtual machine and a container. From that point of view this thesis did not provide more new information since there was not such deep technical research on the differences between a container and a virtual machine. The research of differences between a container and a virtual machine were mainly directed by the objective of creating a generic reusable software robot.

There is not much literature on a software robot, and it seems that the RPA solutions are quite a new phenomenon. The best suitability for the RPA solutions is in places where repetitive tasks can be automated. The RPA can be used to make an integration between systems but when measured with robustness and reliability the usage of interfaces and pure software defined solutions would be better. The RPA has always a possibility of crashing due to something unexpected on a screen. An ordinary software project is easier to make robust against unexpected situations than the RPA.

A virtualisation is a way to interconnect systems which would otherwise be unable to communicate with each other. A virtualisation can also give more tools to work on different kinds of platforms in the future. In this sense the virtualisation and the RPA solutions have same goals, but the level of abstraction is different. Through a virtualisation it is possible to have systems to work better together and provide coherent interfaces for different applications.

A virtualisation was the only rational solution to provide the environment for a software robot. The robot could have been implemented directly to a cloud service since the cloud providers offer their own robot implementation tools but then it would not have been possible easily to port the implementation to any other cloud service. The general-purpose nature of a platform created own challenges of how to create a solution which can be deployed reliably enough wherever needed. The main problem was to find how the robot can interact with the systems the robot should use. The minor problem was to produce stable environment where everything starts automatically when a new container instance is started.

The main problem was caused by the container isolation. Since a container is an isolated environment, all network connections are refused by default. The isolation should not be broken, and minimum number of ports should be opened to reduce a possible attack surface on a container. A robot can interact easily with systems accessible via the Internet, but other systems require more configuration to be accessible immediately after starting a container instance. The interaction with external systems was enabled by adding capabilities for the container.

The minor problem was result from the requisite of a fully automated start-up process. A container instance should be created without any human involvement since then it is possible to completely automate scaling of resources. The automation of a start-up process was enabled by using multiple bash scripts which have only one task to be performed. The major problem here was how to transfer information of robot ID from a control panel to a container. This was solved by using environmental variables inside a container.

There is no similar research of a general reusable software robot. Containerisation of a robotic process automation has been researched in a specific implementation. The generic nature of this software robot implementation opens new possibilities since there is no need to always configure the environment rather, it is only needed to develop the robot task and then assign it to any robot available. This thesis provides a new insight into possible implementations of robotic process automation and how to take advantage of the untiring nature of a robot.

During the development of a reusable generic software robot a few topics for further development raised. Three major topics were identified for further development.

Firstly, the next steps in the development of a generic reusable software robot would be the development of controlling multiple robots working with a same task. It should be refined how the information and work status is transferred between robots. The performance and speed of multiple parallel working robots could be improved with proper architecture. The old architecture is created on the assumption that only one robot is started with a predefined task to be done. The new architecture is implemented on top of this assumption meaning that the current architecture is not ideal to support usage of multiple robots.

Secondly, the size of a robot container image should be reduced if possible, by tweaking what libraries and binaries is needed and what can be deleted since a robot does not use them. By reducing the image size, it is possible to get even faster start-up times for a robot when a new instance is created in a cloud. A smaller size of a container would also make creating new images from Dockerfile a faster process. Now a new image can be created in around 20 minutes since there is so many different libraries and binaries which needs to be downloaded and installed.

Finally, the control panel should be refined to better support multiple robots working with a same task. This is related to previous topic as well since the information between robots should be transferred via the control panel and the control panel should have all the information of all robots all the time. The control panel could be developed further to enable sending information or files to the container when the robot is created consequently the mounting of a network drive to a container would not be necessary.

On some use cases a virtual machine-based solution would work better if a cloud environment is fixed and does not support needed capabilities. A virtual machine-based solution would also work in a use case where a robot would work incessantly. If a robot is designed to work incessantly it makes no sense to shut down the instance if a robot crashes due to it taking more time to create a new instance than it takes time to restart a crashed robot.

A generic reusable software robot could be used to reduce development time and to ease development of robots. Automation of repetitive tasks can be made easier with a generic reusable software robot. The amount of data is rising in every organisation all over the globe and the need to move and organize the collected data to old systems by hand is rising. The need of manual human work can be reduced in these repetitive tasks with generic reusable software robots. The productivity of an organisation can be increased with software robots since humans can use their time and knowledge to perform tasks where their abilities can be utilised. Repetitive simple tasks are a waste of human abilities in organisations and the productivity of a single human could be significantly increased when a software robot handles those repetitive tasks.

5 SUMMARY

In this thesis, two things were studied. Firstly, the possible methods to implement a general reusable software robot. Secondly, the cloud providers were scanned to find out which cloud environment would be the most suitable for a generic reusable software robot. Finally, based on the abovementioned studies a reusable software robot was developed.

The main purpose of this thesis was to create a generic reusable software robot which can be deployed into any IaaS type of a cloud service. By creating a reusable software robot, it is possible to minimise the time needed to configure the environment so that a robot can work successfully. By minimising the time, it is possible to provide shorter development and deployment times because no extra time is needed to setup the environment.

The first thing to be studied was how to implement a virtualised environment into a cloud service. A container and a virtual machine were the possibilities for virtualising the environment. The two possible implementations were examined based on a limitation that resulting implementation needs to be a cloud service compatible. Firstly, it was found that a container-based implementation would be the best option because it is lightweight to move around and secondly, a start-up time of a new instance in a cloud service is fast.

After studying possible implementation methods, cloud providers were scanned. Two possible cloud providers, AWS and Azure, were studied more closely hence they both offer infrastructure as a service and they are commonly used. AWS was chosen to be a platform to be used because of a higher maturity level and because of the possibility to either add or to remove container capabilities. For a software robot in a cloud it is important to be able to add capabilities for a container so that a network drive can be mounted into that container. The network drive can then be used to save the logs of a robot and to transfer data between a short-term filesystem of a robot and fixed filesystem where the network drive is.

In the implementation of the robot it was focused on the Sikuli framework and its specific features. With Sikuli 1.1.4. it is possible to deploy *.jar*-packets into container and no specific installation for the Sikuli framework is needed. By using compiled *.jar*-packets building of a new container image can be simplified. As a result, a container which can be deployed to AWS or Azure was produced.

6 REFERENCES

[1] Liyanage M. (2018) Software Defined Mobile Network (SDMN) Architecture. Issue Communications Networks 2 lecture material. Pages: 25.

[2] Smith J.E. (2005) The architecture of virtual machines. Computer 38 Issue 5. Pages: 32-38.

[3] Brief history of virtualization. (Updated: 2012) (Accessed 10/2019), Oracle. URL: https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html.

[4] Li X. (2017) Advanced Design and Implementation of Virtual Machines. Publisher: CRC Press.

[5] Hypervisor. (Updated: 2012) (Accessed 10/2019), Oracle. URL: https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1011.html.

[6] Howard J. (Updated: 2008) (Accessed 10/2019), Hyper-V RTM announcement. Available today from the Microsoft Download Centre. Anonymous URL: https://blogs.technet.microsoft.com/jhoward/2008/06/26/hyper-v-rtm-announcement-available-today-from-the-microsoft-download-centre/.

[7] Chirammal H.D., Mukhedkar P. & Vettathu A. (2016) Mastering KVM Virtualization. Publisher: Packt Publishing.

[8] Singh S. (2016) Containers & Docker: Emerging Roles & Future of Cloud Technology. Publisher: IEEE. Pages: 804-807.

[9] VirtualBox. (Updated: 2019) (Accessed 10/2019), Oracle. URL: https://www.virtualbox.org/.

[10] Joy A.M. (2015) Performance Comparison between Linux Containers and Virtual Machines. Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in. Publisher: IEEE. Pages: 342-346.

[11] Linux wiki, Chroot. (Updated: 2018) (Accessed 10/2019), Miscellaneous. URL: https://www.linux.fi/wiki/Chroot.

[12] Linux Containers. (Accessed 11/2019), Canonical Ltd. URL: https://linuxcontainers.org/;.

[13] Linux Containers, Security. (Accessed 11/2019), Canonical Ltd. URL: https://linuxcontainers.org/lxc/security/.

[14] Project rkt. (Updated: 2019) (Accessed 10/2019), Red Hat Inc. URL: https://coreos.com/rkt/.

[15] Rkt documentation in GitHub. (Updated: 2019) (Accessed 11/2019), Miscellaneous. URL: https://github.com/rkt/rkt/tree/master/Documentation.

[16] Announcing Windows 10 Insider Preview Build 14361. (Updated: 2016) (Accessed 10/2019), Microsoft. URL: https://blogs.windows.com/windowsexperience/2016/06/08/announcing-windows-10-insider-preview-build-14361/.

[17] Docker and Microsoft: Integrating Docker with Windows Server and Microsoft Azure. (Updated: 2014) (Accessed 10/2019), Microsoft. URL: https://weblogs.asp.net/scottgu/docker-and-microsoft-integrating-docker-with-windows-server-and-microsoft-azure.

[18] Docker documentation. (Updated: 2018) (Accessed 10/2019), Docker Inc. URL: https://docs.docker.com;.

[19] Kasampalis S. (2010) Copy on Write Based File Systems
Performance Analysis and Implementation. Technical University of Denmark, Department of Informatics.

[20] LXC documentation. (Updated: 2019) (Accessed 11/2019), Miscellaneous. URL: https://github.com/lxc/lxc.

[21] Saltzer J.H. & Schroeder M.D. (1975) The protection of information in computer systems. Proceedings of the IEEE 63 Issue 9. Pages: 1278-1308.

[22] Multi-Account Containers. (Updated: 2019) (Accessed 11/2019), Mozilla. URL: https://support.mozilla.org/en-US/kb/containers.

[23] Nagy G. (Updated: 2015) (Accessed 11/2019), Operating System Containers vs. Application Containers. Anonymous URL: https://blog.risingstack.com/operating-system-containers-vs-application-containers/;.

[24] Tozzi C. (Updated: 2016) (Accessed 11/2019), Why Use a System Container Instead of a Docker App Container? Anonymous URL: https://containerjournal.com/features/use-system-containers-instead-docker-app-containers/.

[25] Ubuntu 18.04.1 LTS (Bionic Beaver). (Updated: 2018) (Accessed 10/2019), Canonical Ltd. URL: http://releases.ubuntu.com/18.04/.

[26] Watson R. (Updated: 2015) (Accessed 10/2019), Docker Democratizes Virtualization for DevOps-Minded Developers and Administrators. Anonymous URL: https://blogs.gartner.com/richard-watson/docker-democratizes-virtualization-devops-minded-developers-administrators/.

[27] Mouat A. (2016) Docker Security: Using Containers Safely in Production. Publisher: O'Reilly Media, Inc.

[28] Docker security. (Updated: 2019) (Accessed 11/2019), Docker Inc. URL: https://docs.docker.com/engine/security/security/.

[29] Perla E., Massimiliano O. & Speake G. (2011) A Guide to Kernel Exploitation : Attacking the Core. Burlington, Mass. Publisher: Syngress.

[30] Vaughan-Nichols S.J. (Updated: 2015) (Accessed 11/2019), Docker 1.8 adds serious container security. Anonymous URL: https://www.zdnet.com/article/docker-1-8-adds-serious-container-security/;.

[31] Daemon definition. (Updated: 2005) (Accessed 11/2019), The Linux Information Project. URL: http://www.linfo.org/daemon.html.

[32] daemon. (Updated: 2019) (Accessed 11/2019), Docker Inc. URL: https://docs.docker.com/engine/reference/commandline/dockerd/.

[33] Pulapaka H. (Updated: 2018) (Accessed 10/2019), Windows Sandbox. Anonymous URL: https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/Windows-Sandbox/ba-p/301849.

[34] Oktavianto D. & Muhardianto I. (2013) Cuckoo Malware Analysis : Analyze Malware using Cuckoo Sandbox. Publisher: Packt Publishing.

[35] Coleman M. (Updated: 2016) (Accessed 10/2019), Containers are not VMs. Anonymous URL: https://www.docker.com/blog/containers-are-not-vms/.

[36] Simple monthly calculator. (Updated: 2019) (Accessed 10/2019), Amazon Web Services Inc. URL: https://calculator.s3.amazonaws.com/index.html.

[37] Gordon W. and Long E. (Updated: 2019) (Accessed 11/2019), Why Chrome uses so much freaking RAM. Anonymous URL: https://lifehacker.com/why-chrome-uses-so-much-freaking-ram-1702537477.

[38] Amazon EC2 Auto Scaling. (Updated: 2019) (Accessed 11/2019), Amazon Web Services Inc. URL: https://aws.amazon.com/ec2/autoscaling/?sc_channel=ba&sc_campaign=autoscaling-ec2-button&sc_medium=button&sc_country=global&sc_geo=global&sc_outcome=aware.

[39] Overview of autoscale in Microsoft Azure Virtual Machines, Cloud Services, and Web Apps. (Updated: 2018) (Accessed 11/2019), Microsoft. URL: https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-overview.

[40] Introducing Amazon EC2 Container Service. (Updated: 2014) (Accessed 11/2019), Amazon Web Services Inc. URL: https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-amazon-ec2-container-service/.

[41] AWS Elastic Beanstalk adds Docker support. (Updated: 2014) (Accessed 11/2019), Amazon Web Services Inc. URL: https://aws.amazon.com/about-aws/whats-new/2014/04/23/aws-elastic-beanstalk-adds-docker-support/.

[42] Zander J. (Updated: 2014) (Accessed 11/2019), New Windows Server containers and Azure support for Docker. Anonymous URL: https://azure.microsoft.com/es-es/blog/new-windows-server-containers-and-azure-support-for-docker/.

[43] Add/remove Linux capabilities for Docker container. (Updated: 2018) (Accessed 10/2019), Microsoft developer network. URL: https://social.msdn.microsoft.com/Forums/azure/en-US/20b5c3f8-f849-4da2-92d9-374a37e6f446/addremove-linux-capabilities-for-docker-container?forum=WAVirtualMachinesforWindows.

[44] Amazon ECS adds support for adding or dropping Linux capabilities to containers. (Updated: 2017) (Accessed 10/2019), Amazon Web Services Inc. URL: https://aws.amazon.com/about-aws/whats-new/2017/09/amazon-ecs-adds-support-for-adding-or-dropping-linux-capabilities-to-containers/.

[45] About AWS. (Updated: 2019) (Accessed 11/2019), Amazon Web Services Inc. URL: https://aws.amazon.com/about-aws/.

[46] Microsoft Announces Windows Azure and Azure Service Platform. (Updated: 2008) (Accessed 11/2019), Microsoft. URL: https://azure.microsoft.com/en-us/blog/microsoft-announces-windows-azure-and-azure-services-platform/.

[47] Hocke R. (Updated: 2019) (Accessed 10/2019), Sikuli Documentation. Anonymous URL: https://sikulix-2014.readthedocs.io/en/latest/index.html.

[48] Lee S., Suh S., Mo S., Trofimov A. & Jeong B. (2009) A Virtual Window System for CE Devices Based on System Virtualization. 2009 6th IEEE Consumer Communications and Networking Conference. Publisher: IEEE.

[49] The Institute of Electrical and Electronics Engineers Inc. (1994) IEEE Std 1295-1993: IEEE Standard for Information Technology--X Window System--Modular Toolkit Environment (MTE). In: Anonymous Standard. Publisher: IEEE. Pages: 13.

7 APPENDICES

Appendix 1 Dockerfile

```
FROM ubuntu:18.04
ENV LC_ALL en_US.UTF-8
ENV LANG en_US.UTF-8
ENV LANGUAGE en_US.UTF-8
ARG DEBIAN_FRONTEND=noninteractive
COPY keyboard /etc/default/keyboard
RUN mkdir -p /root/.Sikulix/Lib && mkdir -p /root/src/SikuliX && mkdir -p /root/Logs
COPY ./sikulix/ /root/src/SikuliX
COPY ./sikulix/jython-standalone-2.7.1.jar /root/.Sikulix
COPY ./sikulix/jruby-complete-9.2.0.0.jar /root/.Sikulix
COPY ./sikulix/Lib/ /root/.Sikulix/Lib
COPY ./initVNC.sh /root/
COPY ./initPlatform.sh /root/
COPY ./initConf.py /root/
COPY ./startRobot.sh /
COPY ./geckodriver.tar.gz /
COPY ./leptonica-1.74.4.tar.gz /root/src
COPY ./tesseract-3.05.02.tar.gz /root/src
RUN apt update && \
apt install -y locales locales-all && locale-gen en_US.UTF-8 && \
apt install -y build-essential sudo firefox wmctrl xdotool xfce4 \
xfce4-goodies tightvncserver openjdk-8-jdk openjdk-8-jre \
autoconf automake libtool autoconf-archive \
pkg-config libpng-dev libjpeg8-dev libtiff5-dev zlib1g-dev \
libicu-dev libpango1.0-dev libcairo2-dev \
libopencv3.2-java python-dev python-numpy maven x11-common cifs-utils && \
sudo ln -s /usr/lib/jni/libopencv_java320.so /usr/lib/libopencv_java.so && \
tar -zxvf /geckodriver.tar.gz && chmod +x /geckodriver && \
mv /geckodriver /usr/bin/ && rm -rf /geckodriver.tar.gz && \
apt remove tesseract-ocr* libleptonica-dev && \
apt autoclean && apt autoremove --purge && \
cd /root/src && tar -zxvf leptonica-1.74.4.tar.gz && \
cd /root/src/leptonica-1.74.4 && \
./configure && sudo make && sudo make install && \
cd /root/src && tar -zxvf tesseract-3.05.02.tar.gz && cd /root/src/tesseract-3.05.02 && \
./autogen.sh && ./configure --enable-debug && \
LDFLAGS="-L/usr/local/lib" CFLAGS="-I/usr/local/include" make && \
sudo make install && sudo make install-langs && sudo ldconfig && \
rm -rf /root/src/tesseract-3.05.02.tar.gz /root/src/leptonica-1.74.4.tar.gz && \
printf 'qwerty\nqwerty\nn\n' | vncpasswd && sudo vncserver && \
sudo vncserver -kill :1 && mv ~/.vnc/xstartup ~/.vnc/xstartup.bak && \
printf '#!/bin/bash\n\nxrdb $HOME/.Xresources\nstartxfce4 &\n' > ~/.vnc/xstartup && \
chmod +x ~/.vnc/xstartup && \
cp /etc/xdg/xfce4/panel/default.xml /etc/xdg/xfce4/xfconf/xfce-perchannel-xml/xfce4-
panel.xml && \
apt autoremove -y && apt autoclean -y
```

Appendix 2 Readme.md

Go to the same folder where Dockerfile is located
Run following commands
```
docker build -t <image name> .
docker run -p 5901:5901 --name <container name> -dit <image name>
docker exec -ti <container name> /bin/bash
```
Don't forget the dot at the end of the build command!
--name flag is optional but makes it easier to recognize your containers

If tab isn't working correctly, it is because of the Xfce bug and it can be fixed with these instructions https://www.starnet.com/xwin32kb/tab-key-not-working-when-using-xfce-desktop/