DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Management of Operating System Installations in Heterogeneous Testbeds

Andreas Resch

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Management of Operating System Installations in Heterogeneous Testbeds

Management von Betriebssystem-Installationen in heterogenen Testumgebungen

Author:	Andreas Resch
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Raphael Hetzel, M.Sc.
Submission Date:	September 15, 2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, September 15, 2021 Andreas Resch

Acknowledgments

I would like to thank professor Jörg Ott and the Chair of Connected Mobility for offering me the opportunity to write this thesis. Furthermore, I am grateful to my advisor Raphael Hetzel for his constructive feedback and patience. Lastly, I want to thank my friends and family for supporting me during my studies.

Abstract

The need to test applications and their underlying operating systems is essential to good software development. With software designed to run on different architectures with various specifications, testing the same software on as many devices as possible is invaluable. However, access to different hardware is often limited by physical space, time schedules, and existing infrastructure. The system proposed by this thesis aims to allow its users to be able to test their software remotely inside a heterogeneous testbed and collect the computed data for further development of their software or usage of the calculated information for their scientific research.

The scarcity of available hardware is widely compensated with virtualized testing setups. However, virtualization is not always a viable option, e.g, when there is the need to test the behavior of the developed software on actual hardware. Virtualized testing environments are popular due to their easy-to-learn configuration and the possibility of creating uncomplicated deployments with multiple services. This thesis explores the possibilities and requirements to create a similar easy-to-use system for bare-metal devices.

We examine existing deployment solutions for virtualized testing environments and define the required components for a system that handles the installation of complete operating systems on bare-metal devices in heterogeneous testbeds. Additionally, we discuss the possibilities of various approaches to manage different hardware and define requirements and restrictions of the deployable software and the services needed to automate and manage the installations.

Contents

1 Introduction

Installing and configuring operating systems is associated with several manual actions that need to be completed before it is possible to run an operating system on a computer and execute applications to collect data. These steps of downloading, installing, and configuring the operating systems are tedious and time-consuming tasks, especially when testing an application on different hardware.

To be able to test software on bare-metal devices in a fast and reliable way, the usage and the access to hardware must be simplified.

Creating a testbed with multiple bare-metal devices requires the installation and configuration of different operating systems. There are different solutions for automated operating system installations. However, most of them require their users to learn custom programming languages and immerse themselves in the respective framework. To reduce the time spent creating testbeds for individual experiments, a remotely accessible multi-purpose testbed with heterogeneous devices is needed that allows the deployment of operating systems.

With an easy-to-use and fast deployment system, the testing of applications and operating systems on bare-metal devices becomes more accessible and allows developers and researchers to focus on the development of their software instead of the creation and setup of a testbed for their specific test case. This thesis proposes a system that allows its users to declaratively create operating system images with a concise syntax inspired by the syntax of *Dockerfiles* and *docker-compose* to allow fast and reproducible deployments of experiments in a heterogeneous testbed.

```
1 FROM debian
2 RUN apt-get update
3 RUN apt-get install -y nginx
4 ENTRYPOINT nginx -g "daemon off;"
```

Figure 1.1: Dockerfile used to create a container running a webserver.

Docker Docker is a popular software that allows the creation of virtualized containers with a declarative file called Dockerfile, describing the complete building process of the resulting image. The building process for Docker containers is reproducible and allows computational reproducibility in scientific experiments. Docker provides a relatively small but powerful command set, allowing the creation of operational software with a concise syntax describing and configuring the content of the container [Mer14].

Figure 1.1 shows an example of a Dockerfile. The definition of Docker containers relies on the use of base images, allowing its users to adapt existing containers to their needs. Thus, the first line of every Dockerfile specifies the required base image with the command `FROM`. Docker provides multiple provisioners to modify the content of a container. `RUN` is used to execute commands within the container during the building process of the image. The command `ENTRYPOINT` specifies the command that is executed when the container is started. The status of a running container is coupled to the status of the entrypoint command, if it exits, then the container is stopped.

The Docker ecosystem includes a *Docker repository*, a collection of preconfigured and prebuilt Docker containers that allows its users to use the work of other developers and to publish their own created containers. Through these repositories, a vast majority of operating systems and popular software solutions are available as Docker containers, reducing the time needed to set up a heterogeneous containerized testing environment, as the publishers of the Docker container already completed the time-consuming tasks of creating an operational image [Mer14].

```
1 services:
2   nginx:
3     hostname: nginx
4     image: nginx
5     ports:
6       - 0.0.0.0:80:80
```

Figure 1.2: Deployment of an nginx container with docker-compose.

docker-compose docker-compose manages the deployments of multiple Docker images using the *YAML Ain't Markup Language (YAML)*. The concise syntax allows fast and easy-to-read deployments and configurations of containers. Among other things, it is possible to create different internal networks to allow the collaboration between the different deployed services [Inc21]. Figure 1.2 shows a minimal working deployment of a Docker container using docker-compose. Each service specified in the docker-compose.yml file creates an instance of a Docker container. In the example shown, docker-compose binds port 80 of the host to the same port of the container, allowing remote clients to access the service offered by the created container. Due to the ease of creating containers and the management of their deployment, Docker is widely used in the industry.

Computational reproducibility Computational reproducibility is essential in scientific research. It means that computed results of experiments are reproducible by other researchers to verify the validity of the presented data. Docker was recommended by Boettinger in 2015 as a tool to achieve the reproducibility of scientific experiments [Boe15].

The system proposed by this thesis aims to extend the computational reproducibility to include the complete operating system by creating reproducible images of operating systems and installing them in heterogeneous testbeds.

The heterogeneous testbeds managed by the proposed system should feature various types of hardware, from *Single Board Computers* (SBC) like the Raspberry Pi 4 Model B to consumer devices such as an Intel NUC to enterprise servers. The usage of the remote testbed should allow its users access to a broad spectrum of bare-metal devices and the ability to install and test their software on different hardware simultaneously.

In the following chapter, we discuss the differences between virtualized and bare-metal deployments and present related work. In Chapter 3 we outline the design decisions taken in the development process and set the basic requirements for heterogeneous testbeds and their management software. In Chapter 4 we present an implementation of a testbed management software, and in Chapter 5 we evaluate the developed system and discuss its advantages and disadvantages. In Chapter 6 we specify different ways to improve the system and in Chapter 7 a short conclusion is given.

2 Background

To be able to manage operating system installations in heterogeneous testbeds, we first need to identify what concepts are needed to install operating systems on heterogeneous hardware. To find the correct solution for the management of a heterogeneous testbed, we evaluate similar systems and combine the needed components.

2.1 Virtualized vs. Bare-Metal Deployments

For developers and researchers without the possibility to create a bare-metal testbed, virtualization is a good alternative to compute necessary data to advance the development of their software or scientific research. Virtualized testing is mostly achieved using Virtual machines and containerized solutions like Docker. With the latter being a more resource-efficient and easier to use solution. Containers share the host's operating system and are used to execute a single application. To run the same application inside a virtual machine, the execution of a complete operating system is needed [YGR19].

The choice between the usage of virtualized or bare-metal deployments depends on the software that needs to be deployed, on the desired evaluations, and on the availability of additional hardware.

One downside of virtualized software is the dependency on the hardware resources that are not available because they are allocated to with other applications and virtual machines, the deployment of a complete operating system on a single bare-metal device guarantees the exclusive usage of the hardware of the bare-metal device and thus allows to achieve more reliable results.

The declarative nature of Docker allows the use of base images, prebuilt and configured Docker containers, as a starting-off point, whereas in the traditional bare-metal deployment, the base image is the image provided by the maintainers of the operating system, amounting to more work for the developers that attempt to create an operational system on bare-metal devices compared to the usage of the containerized solution.

The provisioning of bare-metal devices is mostly accomplished with third-party software like Puppet[1], Chef[2], and Ansible[3] with their respective syntax and programming languages. These solutions can be used for the deployment of bare-metal devices and virtualized machines. However, they require the operating system to be installed beforehand and concentrate on the installation and configuration of the desired services. Some

[1] https://puppet.com
[2] https://www.chef.io
[3] https://www.ansible.com

solutions target the complete configuration of the operating system, such as HashiCorp Packer[4], which is capable of managing the initial installation of the chosen operating system inside a virtualized environment and also handles the provisioning of the created virtual machine.

2.2 Remote Management Concepts

To be able to remotely manage a bare-metal device without manual interactions, the usage of remote management tools is required. The following concepts are examples of different ways to control the behavior of another bare-metal device. Enterprise hardware often contains special hardware components that allow remote management.

Intelligent Platform Management Interface (IPMI) Intelligent Platform Management Interface is a specification for providing systems management capability in hardware. Systems that support the IPMI specification allow access to multiple manageable features, including power control and the possibility to change the boot order and install new operating systems. The IPMI management is also possible when the hardware is powered off, allowing access to the hardware in case of crashes without physical interaction. The specification is implemented by different vendors such as Dell with the *Dell Remote Access Controller* DRAC [Jor04].

Preboot eXecution Environment (PXE) Preboot eXecution Environment is a first stage network bootstrap agent. PXE is loaded out of firmware on the client host and performs DHCP queries to obtain an IP address [Han07]. It allows the booting device to boot an operating system provided by a server in the network. The usage of PXE allows network administrators to offer different boot options for their clients and thereby control the software being executed on their hardware. PXE is often used in combination with the second-stage bootstrap agent *PXELINUX*. PXE loads the second-stage bootloader specified in the DHCP options given by the local DHCP-server. The loading of the required files is often achieved with the usage of the *Trivial File Transfer Protocol* (TFTP) [Han07]. TFTP is a very simple protocol used to transfer files [Sol92]. PXE can be used to install new systems on devices or to boot an operating system provided by the PXE server on a client [Int99].

PXE uses different extensions of the Dynamic Host Configuration Protocol (DHCP) to signal the availability of a bootable operating system to connecting clients. The boot process over PXE requires the DHCP-server to send `dhcp-options` to the clients, these options direct the client either to dedicated boot servers or simply set the offered boot file. As illustrated in figure 2.2, the DHCP-server handles the initial steps in the boot process. The booting client requests an IP Address and receives the available configuration and the servers that handle the booting clients. After the initial connection and configuration, the

[4]https://www.packer.io

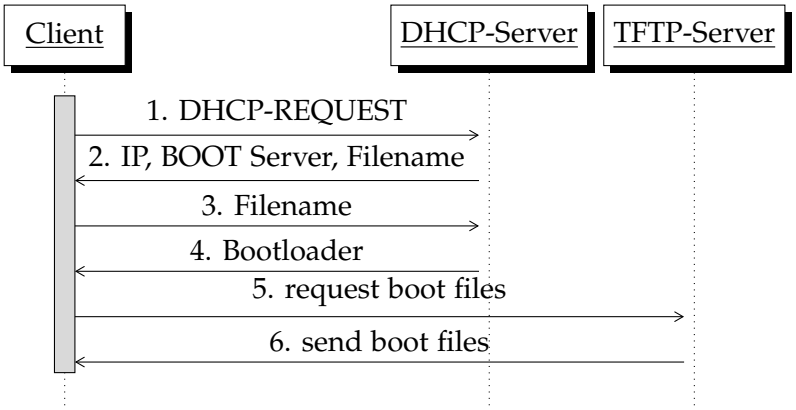

Figure 2.1: Boot process over PXE.

client requests the bootloader from the DHCP-Server, usually the PXELINUX bootloader. PXELINUX then handles the next steps in the boot process, which are mostly requesting the necessary files from the configured TFTP-Server, until the client has received all the required files and can boot the fetched operating system [Han07] [Int99].

PXELINUX A commonly used bootloader for PXE booting is PXELINUX, it is a Syslinux derivative for booting Linux off a network server [Anv08]. PXELINUX uses suffixes to fetch the correct configuration files to differentiate between multiple hardware and guide through the possibly different boot processes. These suffixes are defined by PXELINUX and allow custom boot configurations for multiple clients. After PXELINUX is started on the booting device, the bootloader will attempt to fetch the next boot instructions in the form of a configuration file. This configuration file is provided by the used TFTP-server inside the directory `pxelinux.cfg` and is named after an identifying value of the client, such as the used MAC-Address of the booting device or the assigned IP Address in upper case hexadecimal. A client with the MAC-Address `88:99:AA:BB:CC:DD` will therefore attempt to fetch the file `pxelinux.cfg/01-88-99-aa-bb-cc-dd` before trying to fetch a file with no suffix, as the most customized configuration is preferred [Anv08].

2.3 Related work

Managing a large number of devices is subject to different projects presented in this section. Several projects concentrate on the usage of the concept of *Internet of Things* (IoT), while other focus on more user-directed services.

2.3.1 IoT Clusters

The need for relatively inexpensive clusters used for educational purposes led to the development of clusters in Bolzano [Abr+13] and Southampton [Cox+14]. Both projects

use the Raspberry Pi 1, the first iteration of the affordable Single-board Computers by the Raspberry Pi Foundation. There are other clusters developed at different institutes and corporations, similar to both earlier examples; each one constrained itself to use one particular hardware. Most of the developed SBC clusters use the products of the Raspberry Pi Foundation, clusters with other hardware were constructed but were never scaled beyond ten nodes [Joh+18].

One of the main difficulties in the creation of a cluster with smaller devices is the physical arrangement of the nodes themselves since the layout of the used devices does not always allow for them to be mounted cleanly inside a rack. Most of the previously mentioned projects use custom constructions, created with 3D-Printers, designed by an arts department [Abr+13], or built with LEGO bricks [Cox+14]. The next obstacle is cable management. Each node needs at least a power supply and a network connection. The power supply problem has several solutions: (1) each node has a separate power supply, leading to a maximum amount of cables. (2) clustered power management with multiple-output DC power supplies, reducing the number of cables. (3) Using *Power over Ethernet* (PoE), thus halving the number of needed cables and space requirements, but increasing the overall cost due to the need for managed switches with PoE support [Joh+18].

The main disadvantages of the clusters in Bolzano and Southampton are that both require their nodes to run a specially configured operating system to become a part of the cluster. The installation of new nodes has not been automated and needs to be done by flashing the preconfigured operating system onto a new SD card and subsequently adding the node to the cluster [Abr+13] [Cox+14]. Since at the time of the development of both clusters, the latest available hardware was the Raspberry Pi 1, another option was not possible. The board could not perform a network boot and therefore requires the operating system to be located on the SD card.

2.3.2 FOG

FOG is a Linux-based, free, and open-source computer imaging solution for various versions of Windows (XP, Vista, 7, 8/8.1, 10), Linux, and Mac OS X [Rot16]. It was developed due to the high costs and missing features of commercially available products [Cha16]. FOG is used to create images of hosts and deploy the captured images to other hosts, substantially reducing the time for reimagining a client computer. The project is used in multiple universities and corporations around the world to manage up to 10000 clients [Alb13]. FOG can be deployed on multiple servers at once to reduce the load on single servers and distribute the used services and use dedicated storage nodes.

FOG uses *Preboot eXecution Environments* to start its customized boot menu on the clients, where the user can choose between different boot services. This custom PXE menu also allows the creation of operating system images. The creation of the image is achieved by capturing the content of the local storage device of the booting node. After the cloning of the device, the image can be installed on other devices managed by FOG. The project does not provide a way to build images automatically without user input. It solely concentrates on the management of the installation and the execution of different

tasks inside the network, e.g., powering off idle nodes [Cha16].

2.4 Summary

In the previous paragraphs, we discussed solutions for different problems the system proposed by this thesis seeks to solve. The clusters showcased in Section 2.3.1 give us an idea of what physical aspects need to be considered when creating a heterogeneous testbed. The presented solutions for the power management of the devices inside the clusters can also be utilized in heterogeneous testbeds to accommodate devices with different software or hardware specifications. The project FOG shows a functional approach to handling distributed nodes and shows which services are needed and how they can be utilized to remotely boot different images. The proposed system tries to combine all the inspirations from the Docker ecosystem, FOG, and the different IoT clusters to build a single management system to create, deploy and install images of operating systems.

3 Design

Creating testbeds with heterogeneous bare-metal devices as nodes demands a management system design that can handle different ways of installing operating systems on the used hardware to execute experiments and allow the remote usage of the testbed. In combination with the requirements of a management system, we need to specify the requirements of the testable software, the testbed's internal structure, and the usable hardware.

To create a system that allows the management of operating system installations on heterogeneous devices, it is helpful to utilize working structures and principles of the preexisting solutions discussed in Chapter 1 and Chapter 2. Especially the Docker ecosystem is inspirational as a reference, as it contains all the tools needed to create containers and manage their deployments. A similar system based on the deployment of software on bare-metal devices requires the development of similar tools to allow deployments of complete operating systems on different hardware inside specially created testbeds.

3.1 Deployment Overview

Installing different operating systems on remote devices inside heterogeneous testbeds requires the definition of requirements that need to be satisfied to allow successful deployments.

3.1.1 Operating System Images

Fast and reproducible deployments require the preparation of the deployed software prior to the actual deployment. In the Docker ecosystem, the target environment for the deployed software is created inside a Docker container, forming a so-called image. The usage of preconfigured images allows the distribution of software in predefined, equal environments based on the build instructions of the image. To achieve reproducibility in operating system images, a similar system is needed that bundles the operating system in an image and completes its configuration ahead of the deployment.

To be able to deploy images of operating systems, we need an application that handles the creation and provisioning of the operating system and its services. The syntax of the configuration file for the image should be similarly easy-to-read and as non-verbose as the syntax of the Dockerfile. To create bootable images for heterogeneous hardware, the application used to build the images needs to be able to create and provision operating systems for different types of architectures.

The deployable operating system images must be created on the user's device to add applications and required files to the testing environment. Building the images on the user's device further allows to spot errors during the image creation process, reducing the time needed to create a working, testable, and deployable operating system.

Requirements

The software installed on the nodes inside the testbed must be able to run on the hardware without user input and automatically execute the software or services that must be tested. The operating systems that are deployed in the testbeds can be conventional operating systems and custom systems. To support the deployment of different systems, the management instance of the testbed needs to support the installation of operating systems with remote filesystems and local filesystems.

The operating systems deployed inside the testbeds need to be configured correctly for their intended usage. Due to the wide range of available operating systems and unknown, self-created systems, the nodes can not be generally monitored and accessed without the cooperation of the running software. The requirements needed for a successful deployment inside the remote testbed are necessary to ensure that the testbeds can help with the development of the deployed software.

3.1.2 Deployments

After the creation and provisioning of the operating systems, the resulting images are ready for deployment. A tool is needed that can be easily configured and used to deploy multiple operating systems simultaneously. As described before, docker-compose fills that role in the Docker ecosystem. To install operating systems on bare-metal devices, the counterpart to docker-compose must give instructions to a remote server that handles the actual deployments inside a heterogeneous testbed.

Requirements

To allow the installation of the created operating system images, the configuration of the deployments must contain all the information needed for a successful deployment in a concise configuration language. As the goal is to create a similar component in our system as docker-compose, the syntax should be inspired by the syntax of docker-compose.

3.2 Testbed Architecture

The hardware that executes the operating systems inside the testbed has to adhere to certain requirements to allow remote deployments of operating systems. Since a heterogeneous testbed is naturally composed of different types of hardware, it is necessary to allow various approaches to their management. Some hardware may

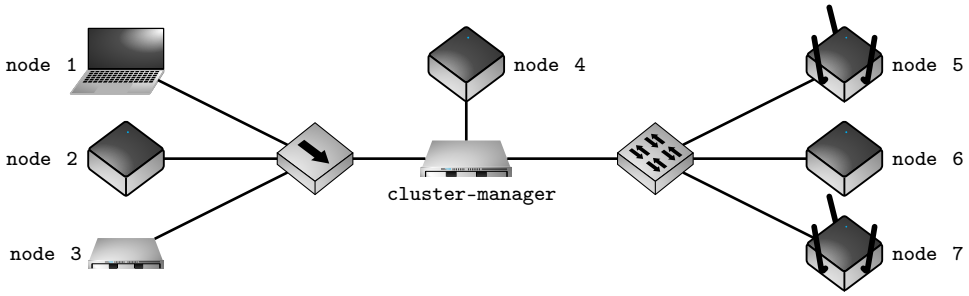

Figure 3.1: Example for a heterogeneous testbed.

contain components that allow remote management using special interfaces, e.g., IPMI or DRAC, while other hardware may need additional support to be manageable from a remote host.

The physical layout of the testbeds depends on the used nodes and their different management possibilities. Figure 3.1 illustrates an example of a heterogeneous testbed. The internal connections in the testbed can be as heterogeneous as the nodes themselves. However, the management server must be able to manage the nodes. The given example uses different possible network connections to manage the nodes. Depending on the capabilities of the used nodes, different connections can be used.

Figure 3.1 also represents the basic structure of a manageable testbed. All the nodes inside the testbed are connected to the managing instance, running the testbed management software. This device is the centerpiece of the testbed, as it governs over all the installations inside the created cluster. To allow such a system to function, several requirements for the internal structure of the network and the hardware of the nodes have to be fulfilled.

3.2.1 Hardware Requirements

The different hardware used as the nodes of the testbed has to be manageable from another host in the system. Manageable means that the nodes must allow some form of control over their boot behavior and their power state.

Each node must attempt to boot from a provided operating system from the network before booting from a local bootable device. This requirement is crucial to the functioning of a remotely managed computer, as it allows another entity inside the network to control the software running on the nodes' hardware.

The second requirement, the control over the power state, is needed to manage the simultaneously powered on devices and force the restart of nodes that need to boot another operating system. Both requirements are essential to the management of testbeds and can be achieved using different technologies and approaches.

Hardware with remote management interfaces such as IPMI or DRAC fulfill both requirements, as these standardized interfaces allow the remote configuration of the boot order and are able to control the power cycle of the node. Bare-metal devices

without dedicated management interfaces require different strategies to be part of the testbeds. The boot configuration of these devices must be modified to adhere to the requirements. Combined with the usage of remote booting solutions such as PXE, the selection between booting from the network or from a local device is possible.

If a bare-metal device does not support booting from an operating system provided by a service inside the network, the deployment of different systems is not possible without manual interactions with the hardware, which are not desired in a remotely used testbed.

The power management of devices without standardized remote management interfaces can be achieved using different approaches. If the hardware supports Power Over Ethernet, then a Power Over Ethernet capable switch can be used to fulfill the requirement. Otherwise, external power solutions such as Smart Devices or individual constructions to manage the power delivery to the nodes may be used. These peripherals used for the power management must be controllable by the management instance of the testbed. Creating different possibilities to handle the power management of the nodes permits the addition of heterogeneous hardware to the testbed that otherwise would not be remotely manageable.

3.2.2 Service Requirements

The testbeds need to offer their users a framework that enables the execution of deployments to gather as much information as needed to evaluate the tested software. Since the deployments of the operating systems are carried out remotely, the users have no direct access to the hardware and rely on the output of their deployed applications to evaluate the results.

To collect the output and gathered data from the nodes, the testbeds need to offer the ability to use a centralized logging system to facilitate a possibility to persist the computed values beyond the lifespan of the current deployment. The logging system needs to be easy-to-use and accessible via a simplistic approach and must not interfere with the running application. The developers should not need to implement major changes in the tested applications to allow the use of the logging system. Assuming that most modern operating systems contain a functional network stack, an HTTP-Endpoint on the management server would allow each node to send their messages over the internal network using an HTTP-Post call to a network-wide logging system.

In cases where the deployed operating system does not include a working network stack, the availability of a serial interface on the target node can be used as an alternative log source. The establishment of a serial connection between the node and the management server can be created directly or remotely by using an additional device that reads the output of the node and forwards it over the internal network to the logging endpoint on the management server. The peripheral devices used to transmit the logging output can be smaller devices such as an ESP32 or a Raspberry Pi. However, due to the heterogeneity of the used hardware in the testbed, the existence of a serial interface can not be generally assumed. In instances where no network stack and no

serial interfaces are available, the node is not accessible during the execution of the deployment. Thus logs need to be written to a predefined location on the local storage of the device during the deployment. With the knowledge of where the collected data is located, the testbed management software may extract the information from the local device and transfer the collected data to the management server after the completion of the deployed service.

To enable the deployment of the created images in a remote testbed, the managing instance of the testbed needs to offer a repository of operating systems to allow its users to upload new operating systems or retrieve prebuilt images as a starting-off point for new images. This repository of operating systems is to be used analogous to the Docker repository, uploading created images to share with other testbed users, allowing other users to benefit from previously created images and accelerate their deployment process. To provide different operating system images, the managing server needs enough storage space to store the various operating system images.

3.2.3 Network Requirements

The requirements for the internal network connecting the different nodes to the management server are as important as the requirements for the hardware. The creation and management of an internal network inside the testbed allow the management software to orchestrate the different deployments of operating systems. The internal network needs to fulfill different requirements to handle different node architectures, peripherals, and network hardware.

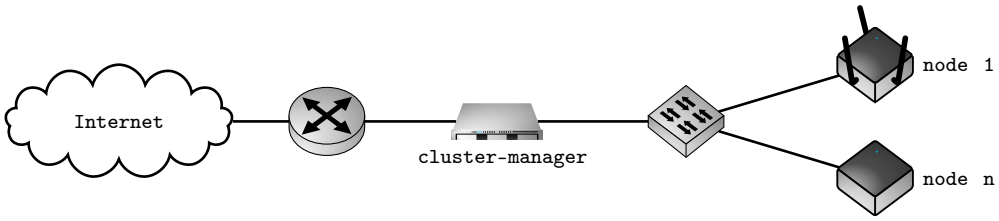

Figure 3.2: Logical network topology of a testbed

In order to provide boot options to the heterogeneous nodes using PXE, the testbed management software needs to control the local, authoritative *Dynamic Host Configuration Protocol* (DHCP) server. Each node needs an IP Address inside the network, managed by the management server. The strategy of leasing the addresses relies on the intended usage of the testbed. If the expected experiments for the nodes rely heavily on the creation of custom networks inside the testbed, the lease should be manageable by each experiment, respectively. Otherwise, the nodes should receive a static IP Address to simplify the network management. The complete control over the address space inside the network allows basic restriction of access to possible network shares, and it eases the association of recorded logging messages received over the network.

Peripherals used in the testbed to manage the power state of nodes or to allow remote

serial connections to the management server require the creation of a separate network. This network is created solely for the management of the peripherals. The type of the network depends on the peripherals in the testbed; most smaller devices that could be used for remote power management, e.g., Smart Plugs, typically only include wireless network interfaces, thus the creation of a wireless network is necessary.

Figure 3.2 illustrates the basic network topology for a testbed. The nodes are connected to the management server, the server is in complete control of the internal network. The nodes themselves should not be part of networks other than the ones managed by the management server. The management server can be accessed over external networks to allow the remote usage of the managed hardware in the testbed.

The created internal networks allow the management server to be constantly connected to the nodes and supporting hardware. To observe the state of the deployments, the nodes must be physically connected at all times to the management server. To be able to successfully create new deployments, the peripherals managing the power cycle of the nodes have to be powered on and controllable at all times.

With the cluster management server being in control of the internal networks, access to external networks can also be managed. The host of the management software should be the only gateway to networks that are not part of the testbed. The management server should be reachable from the outside to allow the remote management of the testbed. Whether or not the traffic from the nodes should be forwarded to external networks depends on the intended use case of the testbed. If the anticipated deployments inside the testbed are completely independent of external data, the cluster management server should not allow connections to the outside. However, if the nodes are used to gather data provided by external services, the management software should forward outgoing connections. A firewall must secure both instances to prevent malicious traffic and unauthorized access to the testbed.

3.3 Testbed Management

To be able to create an easy-to-use management system for the defined heterogeneous testbeds, we need to specify the required components and their tasks in the system.

3.3.1 Management Server

The requirements defined in Section 3.2.2 and 3.2.3 state that the management server has to be able to control all the connected nodes inside the testbed and offer a usable framework that allows the gathering of information during the deployments. To fulfill these requirements, the management server needs to execute several services simultaneously and provide an application to control the deployments on the nodes and the nodes themselves. Nodes that fit the hardware requirements should be easily added to the testbed.

The major part of the testbed management needs to be performed on the management

server itself. Booting an operating system from the network requires the cooperation of several services. To signal the possibility of a network boot, we need to modify the configuration of the local DHCP server and TFTP server. Without the addition of the new deployment to the configuration of these services, the booting node will not attempt to boot from the network. After the device starts to boot, the required filesystem needs to be present on the management server and accessible for the node. All the three mentioned services are essential for all the deployments inside the testbed, and their configuration needs to be automated. The management software must provide an easy-to-use application to ease the configuration of the different services required to boot remote nodes and to deploy images.

3.3.2 Client

An interface is needed to enable the users of the testbeds to interact remotely with the management server. This interface can be a simple REST-API or a more sophisticated client application. The client for the testbed management server should feature a set of commands that allow the deployment of experiments and observe the status of the deployments. The client should connect to the provided API of the management software to allow remote deployments from the user's devices. The existence of a client application eases the deployment of the operating system images, as the client can be used to upload the created images onto the management server. The client application should combine the ability to create images and manage their deployments.

3.3.3 Deployment Components

Analogous to Docker, the deployment of operating systems in heterogeneous testbeds should be possible with the definition of two files, the equivalents to the Dockerfile and the docker-compose.yml. We defined the requirements of the resulting images and the deployments in Section 3.1.1 and 3.1.2. To achieve these results, we must now define the configuration files that allow the deployments.

Imagefile

To allow deployments of operating systems with the ease of the deployments of Docker containers, we need a single file that contains all the build information needed to create a bootable image, the Imagefile. The Imagefile should enable the construction of operating systems that meet the requirements of Section 3.1.1 and feature a readable syntax. The Imagefile acts as an abstraction layer for the configuration of the application needed to build the images, allowing the usage of different builder applications if needed. Contrary to the Docker ecosystem, the

creation and configuration of the operating systems may require the specification of the desired disk layout on the bare-metal device. Thus, we need commands that are able to create different partitions. Additionally, some operating systems require the

completion of an installer before the system can be used. The increased configuration of bare-metal devices makes the creation of operational images more elaborate than the creation of Docker images.

The content of the Imagefile not only creates the desired disk layout and provisions the operating system, but it should also contain information for an eventual deployment inside a heterogeneous testbed. This information is needed for the management server to choose the correct deployment approach and to ensure the successful booting of the image. Thus, the Imagefile must specify if the filesystem of the operating system is designed to be located on the node itself or if the management server can provide it. The content of the Imagefile determines the desired configuration of the resulting operating system image. The operating system itself needs to be created by a builder application that is able to create bootable images. One possible builder application is HashiCorp Packer, an open-source tool for creating identical machine images [Esc20]. Packer includes builders for several different Virtual Machine environments such as VMware, VirtualBox, Amazon EC2, and Vagrant. The creation of bare-metal images requires additional work because Packer is designed to create images for virtualized environments. Packer allows the creation of third-party plugins, one available builder is Packer Builder ARM [Kac21], making it possible to cross-build images for ARM devices. The creation of X86 images relies on the VirtualBox builder. To create bootable bare-metal images, the created Virtual Machine must be exported, and its disk must be converted to a more useful format that allows the extraction of its content or the direct writing on a storage device.

Filesystems Operating systems that require the use of a remote filesystem must be specially configured to be used in the testbed. To allow deployments of the operating system on different nodes, the mount parameters of the filesystem must be adaptable to the internal network and the chosen hardware to guarantee a successful boot process. For example, a node that boots from the network receives an IP Address by the local DHCP server and the location of its bootfiles, the bootfiles contain information about the location of the root filesystem. This location must be adaptable by the management server during the deployment.

Installations of operating systems with their filesystem located on the local storage device of a node do not require the modification of the filesystem during the deployment. These types of deployments may run isolated from other devices inside the testbed. The deployment of operating systems with local filesystems may be necessary when the performance of a developed application or operating system must be measured without interferences from other activities inside the testbed. The gathered data during the experiment must then be collected after the completion of the deployment.

Deploymentfile

To create deployments inside the testbed, the equivalent to the docker-compose.yml, ,the deploymentfile, needs to be defined. The Imagefile contains information for the

deployment approach for the management server, but the accompanying deploymentfile allows the creation of more complex test scenarios with multiple instances of images and different services being deployed simultaneously. The deploymentfile needs to specify the images needed for the experiment and may choose the hardware that should be used for the deployment. The optional selection of the node is important for the testing of applications that require certain hardware. Furthermore, the deployments should be stopped after the required data is collected to allow other users to use the hardware in the testbed. The deploymentfile should feature a possibility to specify when a deployment can be considered complete and release the occupied hardware. The content of the deploymentfile are instructions for the management server to perform the actual deployment inside the heterogeneous testbed.

3.3.4 Storage Devices

The nodes in the heterogeneous testbed will feature different kinds of local storage devices. Smaller devices such as Single Board Computers typically have an SD card as their main storage device, whereas bigger computers have internal Hard disk drives or Solid-state drives. With the deployments of operating systems with a remote filesystem, the internal storage is not used. However, with the possibility of local installations, the state of the local storage device must be observed. Because the longevity of the local storage devices is reduced with the number of local installations. Solid-state drives and SD cards are especially vulnerable to intense usage and should be used as little as possible.

SD cards, when used intensively, have a rather short lifespan compared to other storage solutions. Each read/write cycle reduces the expected lifetime of these small storage devices. The deployment of local operating systems a described in Section 3.1.1 requires the local storage of the nodes to be completely overwritten with every deployment. With potentially multiple local deployments every day in a testbed, the expected lifetime of the SD cards can be immensely reduced and cause early storage failures that lead to corrupt file systems and the eventual inability to create an operational deployment.

The data that is written during the deployments on the local storage devices should therefore be kept on the device to reduce the used read/write cycles on the storage device. However, if a deployment contains sensitive information, the user might insist on the deletion of the complete filesystem on the node. To achieve a compromise between data security and hardware longevity, the storage devices inside the nodes should be used in moderation. The deployment of remote operating systems is possible and should be used for each deployment, as the storage devices inside the management server should be more resilient. If a local deployment is inevitable and the computed data needs to be hidden from other testbed users, a partial deletion of the storage would result in the sensitive data being in an unreadable state and reduce the consumed read/write cycles compared to a complete wiping of the devices.

3.4 Summary

The creation of manageable heterogeneous testbeds requires the consideration of different aspects of the complete system. The power management of the nodes can be solved with different approaches and concepts ranging from individual solutions, e.g., the usage of Smart Plugs, to standardized technologies, e.g., Power Over Ethernet and remote management interfaces.

Installing operating systems on other devices inside a testbed requires knowledge of their hardware specifications and control over different network services such as DHCP and TFTP. The other devices must support some form of management or remote access to provide a possibility to install operating systems from a remote location.

The installed operating system inside the testbed must be able to boot and execute the configured services without user inputs to allow the execution of remotely deployed experiments, thus the collection of the generated output during the deployments is invaluable for the evaluation of the software. The use of different bare-metal devices allows the testbed users to create different test scenarios with multiple devices and evaluate their software on heterogeneous hardware.

4 Implementation

To demonstrate the usability of the defined requirements and components, we implement a testbed management software. The implementation is realized with the programming language *Rust*. Rust is a statically typed programming language designed for developing reliable and efficient systems. Rust's type system and runtime guarantee the absence of data races, buffer overflows, stack overflows, and accesses to uninitialized or deallocated memory [Boe15]. The programming language was chosen for this implementation due to its memory safety features, speed, and great community support.

4.1 Image Creation And Deployment

The requirements in Section 3.1.1 and 3.1.2 state that we need a user-facing application that handles the creation of the image of the operating system with a concise syntax and allows the deployment of the created images. This Section explains how the images are created and deployed using this implementation of a testbed management software. The build application chosen for this implementation is HashiCorp Packer. To incorporate the chosen build application into our system, we need a parser that takes files with Docker-like syntax and returns the configuration needed to build images with HashiCorp Packer.

4.1.1 Parser

The parser for the Imagefile needs to translate the commands of the Imagefile into the format accepted by Packer. *HCL2* (.hcl) is the HashiCorp Configuration Language and is used for the configuration of the software products of HashiCorp, including Packer. The format is too verbose for our usage; thus, we need the Imagefile to abstract the configuration given to Packer. The conversion of the Imagefile requires several default values and varies from architecture to architecture, as different builders require different fields in the HCL2 file. For the creation of ARM images, we are using the mentioned Packer Builder ARM plugin, and for X86 targets, the officially supported VirtualBox builder. As stated in Section 3.3.3, the Imagefile does not only contain information for the building process, it also contains deployment information not handed over to Packer but stored for the eventual deployment of the created image. If the command `PREBUILT` is present in the Imagefile, the building process is skipped, and the archive with the configuration and the operating system image is created, allowing completely custom operating systems. The parser is dependent on other software to be installed on the build host. The building of ARM images can be realized with a containerized Packer

installation provided by the plugin's maintainer, requiring the installation of the Docker engine. The official VirtualBox builder requires the installation of Oracle's VirtualBox. With the usage of different builders, the parser is designed to allow custom commands for the different supported architectures. The parsing application combines the parsing of the Imagefile with the image creation and can also push the image to the repository of the testbed.

```
1  FROM 2021-05-07-raspios-buster-armhf-lite.zip
2  ARCH ARM64
3  FS vfat /boot boot 256M 8192 c
4  FS ext4 / root 0 532480 83
5  CONFIG /boot/cmdline.txt
6
7  RUN echo -n 'console=serial0,115200 ' > /boot/cmdline.txt
8  RUN echo -n 'console=tty root=/dev/nfs ' >> /boot/cmdline.txt
9  RUN echo -n 'nfsroot=%SERVER_IP%:%NFS_ROOT%' >> /boot/cmdline.txt
10 RUN echo -n ',vers=3 rw ip=dhcp' >> /boot/cmdline.txt
11 RUN echo 'rootwait elevator=deadline' >> /boot/cmdline.txt
12 RUN sed -i /UUID/d /etc/fstab
13 RUN touch /boot/ssh
14 RUN echo 'localhost' > /etc/hostname
15 RUN sudo apt-get update
```

Figure 4.1: Imagefile for a Rasbian installation with a remote filesystem

Input

The command set for the Imagefiles is designed with simplicity in mind. We present the most important parts of the command set and their intended usage. Due to the inspiration from the Docker environment, the syntax of the commands in the Imagefile is similar to the commands of the Dockerfile. All the following commands use the syntax `COMMAND [ARGUMENT]` and are usable for both supported processor architectures. The example in Figure 4.1.1 illustrates a working Imagefile that, after parsing, produces an operating system that is deployable inside the created testbed. The commands used in the Imagefile and other commands are explained in the following list:

- `FROM` specifies the original image the operating system will be built upon. Both Packer plugins require a checksum of the provided image. When no other argument is given to the command, the checksum file is assumed to have the same name as the image file and defaults to `sha256`. If no checksum file is available, the command `CHECKSUM` can be used to supply a precalculated checksum. Packer accepts filepaths and URLs as an argument. If a URL is passed, the application will retrieve the image from the remote location.

- `RUN` allows the user to add a new provisioner to the build process. Each `RUN` command is executed in order of appearance inside the Imagefile.
- `FILE` is used to add files from the host environment to the new operating system. The first argument specifies the original filepath on the host, and the second argument defines the location inside the created system.
- `ARCH` is a required command and sets the target architecture for the new operating system. The value of the architecture dictates the used builder of Packer and the usable hardware in the testbed.
- `FS` is the command used to create a new partition within the created operating system. The user must supply the filesystem, the mountpoint, a name, the desired size, its start sector, and partition type.
- `CONFIG` is used to collect the files inside the operating system that need to be rewritten with testbed-specific values such as the internal network address of the managing server.
- `ON-DEVICE` is used to signal that the created image is intended to be installed on the local storage device of the node in the testbed.

The other commands implemented in the parser software are platform-specific and depend on the needed builder for the respective image.

Packer Builder ARM The plugin developed by Kaczanowski is tested with different ARM-powered SBCs such as the Raspberry Pi 4 and the BananaPi R1 [Kac21]. The plugin is easy to configure and allows the creation of running images with the mentioned commands. Internally, the plugin creates a virtual machine using *QEMU*, a fast machine emulator using an original portable dynamic translator [Bel05].

VirtualBox The VirtualBox builder supports the complete configuration of the operating system with the system's installer. If the image file requires the completion of an installer, the required user input can be handled with the special `BOOTCMD` command in the Imagefile, which spoofs keystrokes in the intermediately created Virtual Machine in order to install and configure the operating system. The VirtualBox builder requires a multitude of custom commands to successfully create the guest system. The builder needs the complete specifications of the Virtual Machine, such as the time Packer should wait before beginning to send the keystrokes for the installation. Furthermore, it requires credentials for an SSH connection, as the provisioners are executed after the Virtual Machine has gone through the initial installation and a subsequently completed reboot. The initial installation needs to account for a possible configuration of the SSH service and allow the connection with the given credentials. Thus the needed input to create images for X86 architectures is more verbose and a little more sophisticated than the commands needed for an image for ARM processors.

Output

The parsing application generates the image of the operating system and writes the deployment configuration in the format of a JSON file. The complete output of Packer and the parser is collected in an output directory that contains the files illustrated in Figure 4.1.1. Each builder generates a different image type; ARM images are exported using the Image (.img) format, whereas the result of the VirtualBox builder is the complete Virtual Machine Disk (.vmdk) and the configuration for the generated Virtual Machine (.ovf). The final output of each build is a compressed file with all the required files, ready to be moved to the testbed management server and await usage in an experiment.

```
os-build/
├── configuration.json
├── generated.img
└── image.pkr.hcl
```

(a) Content of an ARM image

```
os-build/
├── configuration.json
├── generated.ovf
├── generated.vmdk
└── image.pkr.hcl
```

(b) Content of a X86-64 image

Figure 4.2: Output of the image creation for different architectures

4.1.2 Deployments

The creation of deployments is achieved with a separate file that resembles the structure and syntax of docker-compose files, as described in Section 3.1.2. Figure 4.3 illustrates the amount of code that is be needed to deploy a previously created operating system. The deployment of the images, as they are preconfigured, does not require more configuration than shown in this figure. Specifying the node and the used IP Address for the deployment is optional. If no preferences are given, then the management software will randomly choose an available node. To automatically free the nodes when all the required data is collected, the deploymentfile offers the possibility to specify criterias that signal the completion of the deployment to the management system. In Figure 4.3 the deployment will automatically stop after the message `shutdown` is found in the logs.

4.2 Management Software

The heart of the testbed is the management software, the cluster-manager, located on the management server of the testbed. After an operating system has been created and pushed to the server as described in Section 4.1, a deployment is possible. The cluster-manager controls the deployments and is responsible for the clean commissioning and decommissioning of operating systems, as well as the correct configuration of all

```
services:
  raspbian:
    image: raspbian
    node: dc-a6-32-76-3d-71
    hostname: raspbian
    replicas: 1
    ipv4-address: 10.0.0.10
stop:
  log:
    - message: "shutdown"
      occurrence: 1
```

Figure 4.3: Example of a bare-metal Raspbian deployment

the required services to fulfill the requirements mentioned in Section 3.2.3 and 3.2.2. This implementation uses the DNS and DHCP service `dnsmasq` to manage the internal network and provide Preboot eXecution Environments and other boot options to the connected nodes, and the uses an implementation of a Network File System (NFS) server to serve the remote filesystems.

The management software consists of a command-line interface to handle the configuration of the testbed and a RESTful API that allows the remote deployment of experiments and the ability to upload new operating system images to the management server.

4.2.1 Configuration

The management software does not only need control over the services used but also needs a configuration of its own. The configuration for the testbed in this implementation is written in YAML and contains all the information needed for the creation of operational remote filesystems and the configuration of each node in the testbed.

Dnsmasq was chosen due to its relatively easy configuration and its capability to be a DNS, DHCP, and TFTP server at once, making it especially useful to provide PXE services inside the testbed. Figure 4.2.1 shows the complete required configuration for dnsmasq in this testbed. The initial configuration of the dnsmasq service requires the choice of the network-wide accessible TFTP directory with the option `tftp-root`. The advertising of a boot server in the local network is achieved with `dhcp-boot`. The deployment of every operating system requires the cluster-manager to restart the dnsmasq service, after its configuration is adjusted to accommodate the new hostname and the internal IPv4 Address of the node. Additionally, the determination of the supported Preboot eXecution Environments inside the testbed is necessary using the configuration `pxe-service`. The configuration of dnsmasq further allows the definition of allowed clients that can receive a DHCP lease, restricting access to the internal network. To provide different boot files

```
1 dhcp-authoritative
2 dhcp-boot=pxelinux.0,10.0.0.1
3 enable-tftp
4 tftp-root=/tftpboot
5 pxe-service=tag:!pxe,0,"Raspberry Pi Boot"
6 pxe-service=tag:pxe,x86PC,"x86-UEFI",pxelinux
7 dhcp-ignore=tag:!known
8 dhcp-host=dc:a6:32:86:a0:a9,10.0.0.10,rpi1
9 dhcp-host=set:pxe,b8:ae:ed:eb:6c:6f,10.0.0.54,nuc
```

Figure 4.4: Dnsmasq configuration for the testbed

to different hardware, dnsmasq can assign tags to the clients. The configuration example in Figure 4.2.1 uses these tags to send the PXELINUX bootloader only to devices with the tag `pxe`. The other devices, in this instance, a Raspberry Pi, do not receive this bootloader as their boot process does not require the PXELINUX bootloader.

Nodes

The hardware that acts as nodes for the testbeds needs to fulfill the requirements in section 3.2.1. To meet these requirements, the hardware must be configured manually, as most devices are not preconfigured to boot from the network first. For example, the boot configuration of the firmware on the Raspberry Pi Model 4 must be modified before joining the testbed.

After the required modifications to the boot sequence of each node are applied, the addition of new hardware needs to be handled. However, before deployments can use a new node, the command-line interface must be explicitly called to configure the NFS and the TFTP-server to prepare them for eventual deployments on the newly added node. Once the node is added to the testbed by the cluster-manager, it is available for deployments. Figure 4.5 displays a sample configuration for a Raspberry Pi 4 Model B. The node is internally identified by the string `dc-a6-32-86-a0-a9`. The values `mac-address`, `tftp-prefix`, `ipv4-address`, and `pxe` are used to identify the node in the network and to create the remote booting environments. The values of the fields `architecture`, `default-os`, `default-user`, and `storage-device` are used to allow the correct assignment of the images to the node, the architecture of the image and of the node must be identical. With the default values it is possible to create local installations on the specified blockdevice in `storage-device`. To create a local installation of an operating system the access to the node must be established via an intermediately deployed system specified by `default-os` that is accessible using a SSH connection using the user supplied by `default-user`.

The power management is not directly achieved by the management software, but is outsourced. This implementation of a testbed management system uses five different

```yaml
nodes:
  dc-a6-32-86-a0-a9:
    name: rpi1
    mac-address: dc:a6:32:86:a0:a9
    tftp-prefix: dc-a6-32-86-a0-a9
    ipv4-address: 10.0.0.10
    architecture: ARM64
    serial-number: 0d26a1f14
    default-os: raspbian
    default-user: pi
    pxe: false
    storage-device: mmcblk0
    log-inputs:
      serial:
        - ttyUSB0
    power:
      on: bash snmp-poe-switch.sh on 24
      off: bash snmp-poe-switch.sh off 24
      reboot: bash snmp-poe-switch.sh reboot 24
```

Figure 4.5: Example for a configuration of a node in YAML

bash scripts to demonstrate the usability of various power management solutions. The first script calls a HTTP endpoint hosted on a Raspberry Pi 4, that toggles the state of an attached relay, forcing a reboot of a Raspberry Pi 4 node by connecting two pins on the SBC designed to hard reset the device. The second and third approaches make use of the PoE capability of the Raspberry Pi and a PoE capable switch. One instance of the script manages the power state over the web interface of the managed switch, and the other implementation uses the *Simple Network Management Protocol* (SNMP) to modify the port configuration. This approach is used in the example 4.5. The fourth script is similar to the first one, but its endpoint is a smart plug running the open-source firmware Tasmota, which allows the management of the status of the smart plug over HTTP calls [Are21].

The possibility to use shell commands as values for the power management commands in the configuration of the nodes allows fast additions of new ways to power a devices and to be able to control its status. The created scripts are essentially wrappers around a simple command such as `curl` and `snmpset` to control another device inside the testbed.

4.2.2 Deployments

To create the deployments inside the testbed, the management server must modify the given images and provide the target filesystem over the network. Every deployment

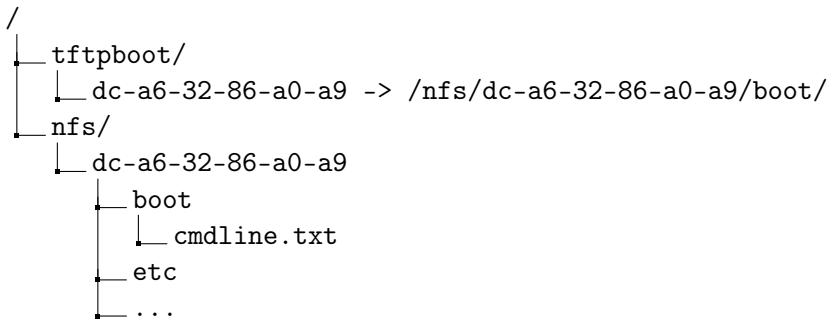

Figure 4.6: Structure of a provided filesystem for a remote deployment

starts with the unpacking of the compressed image. If a remote filesystem is required, the management server temporarily mounts the generated image as a *loopdevice*, creating a readable block device from the content of the image. Using the mount points from the configuration of the image, the loopdevice's content is recreated inside the network share for the node. The creation of the loopdevice requires the existence of an image in the format IMG. As explained in Section 4.1.1 the build process for X86 images produces a different format. Using QEMU, we can convert the VirtualBox Disk Image to a regular, mountable image. If the complete image is a *Logical Volume Manager* (LVM), the recreation of the filesystem is more complex, as the partitions can not be accessed directly but need to be activated to allow the management software to mount the partitions.

To allow the netbooting of the nodes, the TFTP server needs to provide the required boot files. To differentiate between the booting nodes, we use the MAC-Addresses as prefixes for all the configured nodes. A node that requires the bootloader offered by dnsmasq will get a single default option that provides the location of the bootable kernel and eventual kernel options. Hardware that can netboot without pxelinux directly accesses their boot files from the TFTP directory. Here we also use prefixes that are configurable on the client nodes themselves. For example, the Raspberry Pi 4 can request its boot files via TFTP using its MAC-Address as a prefix. With the prefixes in place, the TFTP directory has a subdirectory for each node. In this subdirectory, the boot partition needs to be available for the connecting node. To achieve this, we mount the boot partition again in this directory via a bind mount. A bind mount allows the mounting of a directory inside the filesystem to another point in the filesystem. The final structure for each deployment that needs a remote filesystem is illustrated in Figure 4.2.2 and contains the recreated filesystem inside the network share of the node on the management server and its boot configuration written in the TFTP directory with the boot partition residing in both directories.

As illustrated in Figure 4.1.1, the boot file `/etc/cmdline.txt` contains the variables `%SERVER_IP%` and `%NFS_ROOT%`. These values are replaced during the deployment to direct the booting node to the IP Address of the management server and the shared network filesystem.

The deployment of local installations is more elaborate than remote installations. Since each node is an independent device, the management server has no access when the hardware is powered off. To deploy local installations, we need to first deploy an operating system with a remote filesystem, wait for a complete reboot and move the target operating system to the node. By accessing the deployed operating system over SSH, the management server has to access the node and can overwrite the local storage with the target operating system.

4.2.3 Logging

As defined in Section 3.2.2, the testbed provides a centralized logging component to allow the collection of data during the deployments. The logging service inside the testbed is realized using Logstash, an open-source data collection engine developed by Elastic [Ela21]. Logstash creates an HTTP-Endpoint on the management server and groups the recorded messages by the sending host. Thus, all the messages that a node sends to the specified logging endpoint during the deployment are gathered automatically by Logstash and can be kept after the decommissioning of the deployment for further evaluation. To also provide an alternative logging approach, the usage of serial devices is possible. Nodes that possess a serial interface can be connected to the management server to gather the output of the serial connection. With larger testbeds, the serial connection can not be established directly with the management server. To extend the alternative log source, specially programmed microcontrollers can read the serial output and send it to the management server.

The collection of logs stored on the device itself is achieved after the completed experiment and in a similar fashion to the deployment of operating systems with local filesystems. To gain access to the node, the cluster-manager deploys a predefined, controllable operating system onto the node. The cluster-manager then establishes an SSH connection to the node and mounts the old root filesystem to access the log data in the directory `/results` on the local storage and transfers it to the management server.

4.3 Usage

With the RESTful API provided by the management software, the usage of the testbed can take place completely remotely. The images for the deployments are created using the parser application on the user's device and can be pushed to the operating system image repository on the management server. Deployments of experiments can also be started using the RESTful API, and when a deployment is ended, the collected data can be downloaded by the users to evaluate their application.

The usage of the RESTful API is facilitated with a small client application implemented using Rust. The client application provides a command-line interface similar to the interface on the management server, however, with less possibilities. The application uses different subcommands to allow the management of the testbed, the available

commands and their intended usage of the management software are presented in Table 4.3. The client application features only commands that allow the usage of the testbed and does not modify the configuration of the nodes. However, with the client application, it is possible to retrieve the logs of a terminated deployment from the management server.

Command	**Usage**
`check`	Checks if all the internally used command-line tools and network services are installed.
`deploy`	Can be used to deploy single operating systems or multiple systems simultaneously.
`deployment`	Can display a list of deployments from the database and can be used to stop running deployments.
`image`	Lists the available operating system images.
`install`	Creates the required directories and configurations to start a new testbed.
`node`	Can be used to add or remove nodes from the testbed and can display all configured nodes in a list.
`server`	Used to start the RESTful API to allow the remote use of the testbed.
`watch`	Allows to display a live feed of all logs belonging to a node or a service during a running deployment.

Table 4.1: Subcommands of the management software and their usage.

5 Evaluation

To determine if the designed and created system is beneficial for the management of operating system installations in heterogeneous testbeds, we compare its workflow with manually created deployments.

5.1 Hardware

Due to the popularity of the Raspberry Pi boards, the testbed created and evaluated throughout this chapter is composed of five Raspberry Pi 4 Model B with 4 GB of RAM and equipped with a 32 GB SD card (SanDisk Extreme Pro microSDHC, Class 10). To provide and test the manageability of heterogeneous nodes, an Intel NUC with 16 GB of RAM and a 256 GB Solid-state drive (SSD) is added to the cluster. The six nodes are connected to a Dell Optiplex 7050 with 8 GB of RAM, 4 CPUs (Intel(R) Core(TM) i5-7500T CPU @ 2.70GHz), and a 128 GB SSD (SanDisk X400 M.2) running Debian 10, which acts as the management server of the testbed. The nodes and the management server are connected with a managed PoE Aruba Instant On 1930 Switch, providing power to four Raspberry Pi nodes, which are also equipped with a PoE hat. To test the heterogeneity in the peripherals and their manageability, the power management for the Intel NUC node is achieved using the Smart Plug Sonoff S26, running the open-source firmware Tasmota. One Raspberry Pi is powered by a normal power adapter instead of PoE. To manage the power state of this node, a sixth Raspberry Pi is used, which controls a 5V Relay that can trigger a hard reset to the connected Raspberry Pi. The Dell Optiplex takes up all the tasks discussed in Section 3.3.1, and will be the centerpiece of our evaluations. To allow the remote management of the Smart Plug, the management server creates a wireless network.

5.2 Workflow

To make assumptions over the benefit of the designed and created system, we need to evaluate the time saved with its usage, compared to the manual creation of a testbed. The first metric we look into is the time needed to create deployable operating system images.

5.2.1 Image Creation

To evaluate the image creation process of the created system, we compare the time needed to create a working operating system for a Raspberry Pi 4 Model B. The operating system used in both approaches is Raspbian, a derivation of Debian, and the configuration of the system is the installation of a webserver.

To create a manual installation of Raspbian, the operating system first needs to be downloaded and written onto an SD card. The configuration of the operating system can only be done after the storage device has been inserted into a Raspberry Pi and the device has started. With the image creation tool of our system, the process of downloading and and configuring the operating system is automated and can be done without the access to a Raspberry Pi.

The time needed for the download of the operating system is the same for both approaches. The average time after ten iterations of the creation and configuration bootable image using the image creation tool of our system is 4 minutes and 9 seconds. With the same amount of iterations, the average time required for the image to be written on an SD card is 2 minutes and 35 seconds. To be able to compare the complete process of the image creation, the time needed to boot the Raspberry Pi and configure it manually must be considered. The average time it takes for a Raspberry Pi to boot Raspbian and be accessible over the network is 31 seconds, based on the observation of 100 boot processes. After the average boot time, the device must be accessed and the required software needs to be installed. The installation of a simple webserver on a Raspberry Pi requires 27 seconds on average. With an accumulated average time of 3 minutes and 33 seconds, is the manual approach slightly faster than the image creation tool.

While our system is slower than the manual approach, the image creation and configuration is possible without access to a bare-metal device. When deploying multiple systems at once, the time needed to prepare the bare-metal devices is substantially reduced.

With the manual approach, each system must be flashed and configured separately, requiring the developer or researcher to be present at all times. Using image cloning, like FOG, the manual procedure can be accelerated by capturing the first functional system and replicating the image for the subsequent deployments. This advantage also applies to our system, as the bootable image can be flashed directly onto storage devices without using a remote testbed.

The image creation for a single device is slower than the manual approach. However, when used to create images for multiple devices, the time difference between the manual approach and our system shrinks with every additional device that needs to be configured, as the required configuration is automated and does not require manual actions other than the initial creation of the image configuration. Additionally, the recreation of a past experiment just requires the redeployment of the already existing and configured images, whereas with a manual testbed, the complete physical and logical setup must be reconstructed.

5.2.2 Deployments

To evaluate the deployments of the images, we again use the time used to complete a deployment as a metric. With a manual setup, the differentiation between image creation and deployment is not separatable; the deployment begins as soon as the image is created on the bare-metal device, as the configured applications are already installed. With the usage of the heterogeneous testbed, the time of the deployment can be measured exactly. Thus, we compare different strategies of deploying the operating systems inside the testbed to conclude which approach is better suited for creating deployments of multiple images at once.

The deployment strategies used throughout the testing of the created system were (1) sequential deployments of each node and (2) concurrent deployments of all the required nodes. Measuring the time consumed by the management interface to complete the deployments, we found that, on average, the sequential deployment is faster than the concurrent approach. The comparison in Figure 5.1 shows that with a growing number of nodes needed for a deployment, the time required by the sequential deployment is lower than the time needed for the concurrent deployment.

The advantage over the sequential approach can be explained with the workload of the management server deploying the operating systems. With parallel deployments, the workload on the processors of the management host is substantially higher and thus diminishes the time saved by deploying each instance of the images simultaneously. The creation of a new deployment requires the management software to decompress the image, extract the content of the operating system image, and move the filesystem of the management server.

Observing the workload of the management host with four CPUs during the deployments, the sequential method did not overstress the host. In comparison, the concurrent deployments regularly resulted in the management server being unresponsive to other commands. The concurrent processes that decompress the images and recreate the filesystem of the node on the server require to much processing power for the management server to be responsive for other tasks. This observation indicates that, with the used hardware, the concurrent deployment of operating systems is not a good approach to keep the rest of the testbed operational.

The graph in Figure 5.1 shows the average time measured with deployments of the operating system Raspbian using a remote filesystem. The image has a size of 508 MB when compressed. The deployment times with the usage of different operating systems will vary with the size of the images, as the management server has to decompress the image, mount it, and transfer the data to the network share of the node used for the deployment. The measured data shows that the deployment time grows almost linearly with the number of nodes with both strategies, sequential or concurrent deployments. The sequential method allows the management host to deploy the images faster and be responsive for other tasks during the deployment.

The time measurements so far indicate the speed of deployments with an operating system using a remote filesystem. To broaden the spectrum of supported operating

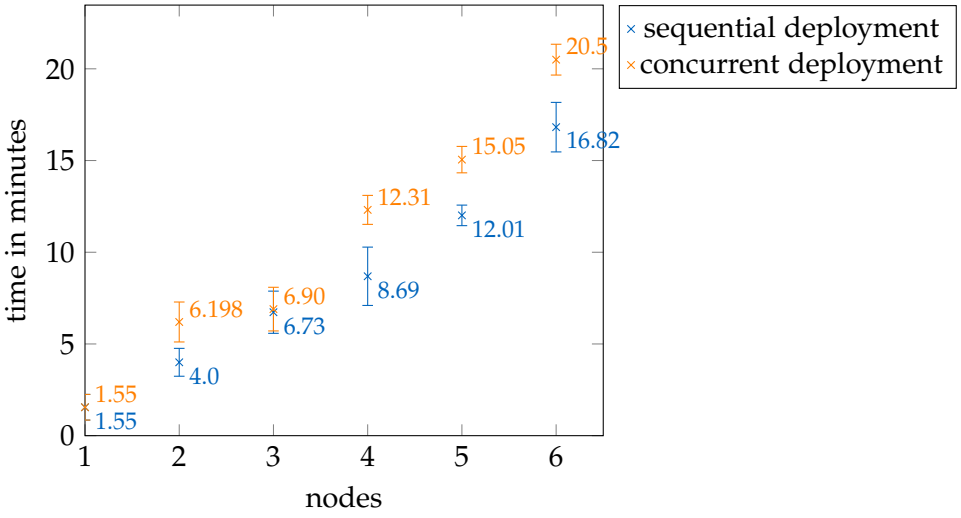

Figure 5.1: Comparison between sequential and parallel deployments.

systems, the deployment of systems with local filesystems is needed and needs to be evaluated. The graph in Figure 5.2 shows that the deployments of images on the local

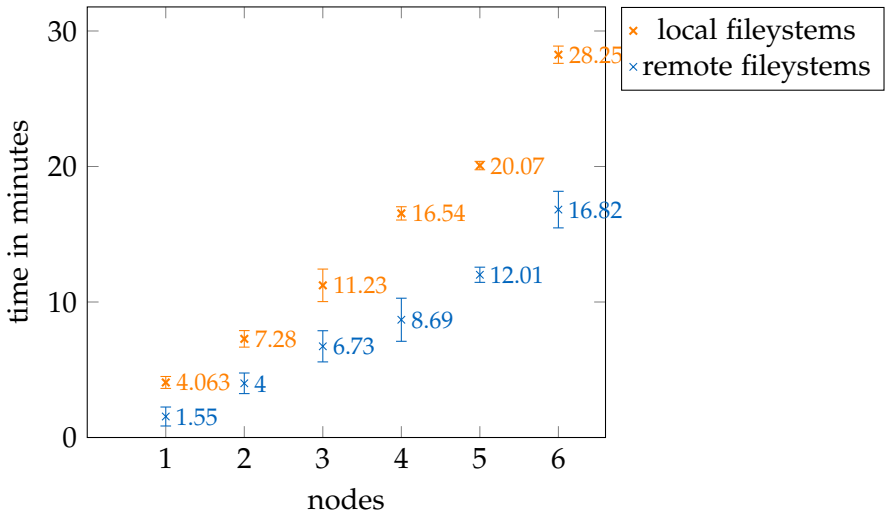

Figure 5.2: Comparison between local and remote fileystem deployments

filesystems of the nodes takes up to three times as long on average as an equivalent deployment with a remote filesystem. This difference comes from several factors that elongate the time needed for a local deployment. The created management system does not have direct access to the storage devices of the nodes inside the testbeds. Due to this restriction, the only possible way to deploy a local operating system is to first deploy an operating system with a remote filesystem on the node and modify the local storage after the intermediate deployment has booted. Thus, the time for the deployment of a

local operating system includes the time needed for the node to be accessible over the network in order for the management server to issue the commands to overwrite the local storage on the node with the desired operating system. The task of overwriting the local operating system is also time-consuming. It can heavily fluctuate between different hardware due to different sizes of the images and the capability of the local storage device being written to. During the deployments of the Raspbian image with a size of 1.8 GB, the average writing speed that could be achieved on local SD cards was 44 MB/s, and on the SSD of the used Intel NUC, the writing speed was 110 MB/s with a similarly sized image. With the flashing of the operating system under a minute, the main part of the time is spent with the deployment and the time waiting for the chosen node to boot the intermediate operating system.

5.3 Hardware Performance

The testbeds should be used to deploy images and allow its users to test their application in a cluster of heterogeneous hardware. The experiments can be used to test the developed application's speed and to compute different types of data. To ensure that the tests are carried out in a performant environment and reflect the real performance of the applications, we need to measure how resource-hungry applications adapt and run inside the testbed. To test the effect of simultaneous deployments and the resulting performance loss, we tested the benchmark `fio`, a tool to measure the performance of the storage device by reading and writing large quantities of data. The conducted benchmarks were configured to write and read a 4 GB file and repeated five times for each amount of nodes.

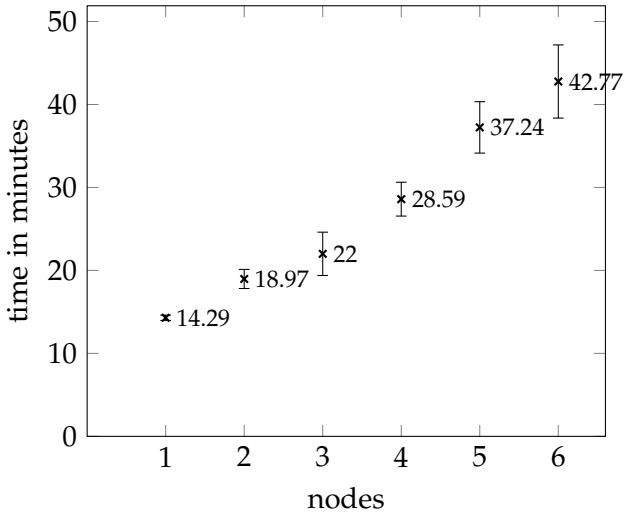

Figure 5.3: Time used to complete the benchmark with different amount of nodes.

Running the benchmark with a single node resulted in an average runtime of 14

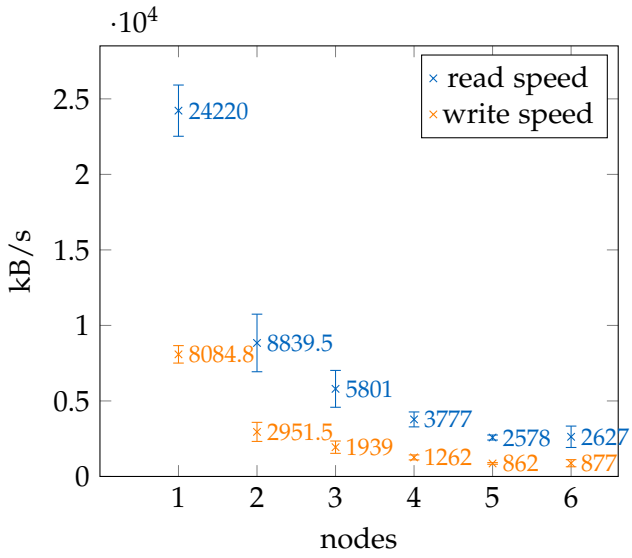

Figure 5.4: Progression of read and write speed with different amount of nodes.

minutes and 17 seconds and an average of 24220 kB/s read, and 8084.8 kB/s written. With every additional node, the performance of the read and write operations declined as expected, as illustrated in Figure 5.4. The average time needed to complete the benchmarks is shown in Figure 5.3, the time increased almost linearly with every additional node, with a significant time difference between the experiments with 4 and 5 nodes. The storage device on the management server also seemed to be overwhelmed by the amount of read and write operations. During the deployments with five nodes, the metric iowait, measuring the time a CPU is idle due to outstanding I/O requests, routinely hit 100%. The data gathered with these benchmarks shows that the testbed's execution speed depends on the concurrent deployments and their usage of the internally provided hardware. Using operating systems with local filesystems, the execution of the same application does not depend on the internal network. It can run without having to compete over the storage device and the processing power of the management server. However, running the benchmark on an operating system with a local filesystem shows why a remote filesystem can still be useful. On Average, `fio` took 15 minutes 10 seconds to execute the same write and read operations locally on a single Raspberry Pi. Secondly, the read and write averages are 4294.5 kB/s and 1435.5 kB/s, respectively, in the range of the remote deployments with four different instances simultaneously, highlighting a problem with the usage of Single Board Computers. SD cards are not as fast and reliable as Solid-state drives.

5.4 Network Performance

The internal network is an essential part of the testbed. It allows the management server to manage the remote installations. Therefore, the functioning of the created internal network is crucial to an operational testbed. The network's performance indicates the speed at which new deployments can be created and existing deployments performed. Applications deployed with multiple instances across the network and need the internal infrastructure to send and receive messages over their network interfaces can force degradation of the network's performance by intensively using it.

To observe the degradation of the network's performance during active experiments we utilize the tool `iperf` to measure the achievable throughput of the internal network. To simulate an active usage of the testbed we deployed three instances of Raspberry Pis executing the benchmark `fio` simultaneously with a remote filesystem located on the management server.

The measurements during the simulation showed that the execution of the simultaneous read and write operations did noticeably impact the performance of the internal network. The measurements showed that during the simulation of the usage of the testbed the reachable throughput of a Gigabit network interface dropped temporarily to around 110 Mbits/sec. The same measurements using `iperf` were taken on a second network interface of the management server and showed no noticeably impact caused by the generated traffic on the main network device.

To allow multiple simultaneously running deployments inside the testbed, the usage of multiple network interfaces on the management server is advisable. With the management server being the controlling instance inside the network, the bottleneck of the network also resides on the management server. With multiple network interfaces, the generated traffic can be separated by its usage. Traffic generated by the initial deployment can be routed over a single network interface. The usage of remote filesystems could potentially require multiple network interfaces, given that the storage device on the management server is able to handle the additional traffic. Additionally, the usage of a faster network interface controller would enhance the performance of the internal network. The hardware used in the evaluated testbed is capable of using a one Gigabit network interface.

5.5 Summary

The measurements in Section 5.3 show that a fast management server is needed to allow performant testing of operating systems within a heterogeneous testbed. Observing the workload on the management server during the evaluations showed that the server requires fast network interfaces and storage devices that can keep up with the generated traffic. The shared network filesystems also require a lot of processing power; a deployment of four different simultaneous deployments of nodes with intensive usage of the filesystems consistently resulted in the used management server to become

unresponsive.

Surprisingly, the system does not require a lot of Random Access Memory (RAM). During the height of the stresstests for the system with the server being unresponsive at times, the usage of the RAM never spiked and averaged at around 1 out of 8 GB used in the management server. Thus, to provide a manageable testbed with several nodes concurrently using the internal network, the resource that is used the least is the RAM. The hardware for a possible management server should therefore contain enough powerful processors to handle simultaneous deployments and a fast storage device to allow fast and consistent read and write operations. Furthermore, the use of different network interfaces to split the network traffic used for remote filesystems and the creation of new deployments from the traffic generated by the running deployments is advisable.

In conclusion we can say that the performance of the created system is improvable, the reliance on a single management server creates a possible performance bottleneck that reduces the achievable throughput of experiments of the testbed.

6 Future Work

The results of the conducted evaluations in Chapter 5 show that the current implementation of the testbed management software requires several minutes to deploy multiple operating systems.

The current implementation of the deployment of operating systems copies the complete content of the image to the management server's filesystem. This can be prevented with the use of OverlayFS. OverlayFS is a union filesystem that merges the content of two filesystems, the 'lower' filesystem is readonly while the 'upper' writable, every file that is part of the content of the 'lower' filesystem and must be written is copied to the 'upper' filesystem. With the operating system acting as the 'lower' filesystem and the network share of the node as the 'upper' filesystem, the deployment time would be drastically reduced [Bro].

In the current version, direct access to the nodes is not possible. Adding a feature that allows users of the testbeds to access the nodes via an SSH connection would allow a more interactive usage of the testbeds. The feature would enable access to hardware for users with simple test cases that do not warrant the creation of a new operating system image but allows to interactively test software on heterogeneous hardware.

In combination with broadening the access to the testbeds to multiple users it should be considered to introduce security measures to the system and require user authentication. The current implementation focused on the creation of a useable system rather than a secure one.

To improve not only the deployment speed but also the configuration of the running operating systems the creation of an internal API could allow an improved user experience. The API should be able to provide additional services such as remote access to filesystems and deliver additional information about the hardware on which the operating system is executed. For example, an operating system with a local filesystem is deployed on a node and requires more storage space than available to conduct its experiments, with the availability of an API the system could automatically request more storage and resume its experiment.

7 Conclusion

Designing and implementing a new solution for managing operating system installations in heterogeneous testbeds required the inspiration from various existing solutions.

The Docker ecosystem was used as an inspiration as it provides a good approach to the creation and management of deployments of reproducible images.

To adapt the deployment approach of Docker to the use of bare-metal devices we defined the requirements for a management solution that allows the installation of operating systems on heterogeneous hardware. Furthermore, we defined requirements to the usable hardware inside the testbeds and the required software to create a manageable system. To create the operating system images the system required a build application that handles the configuration of the system.

In the end we created an operational management software that allows the creation of operating systems using a single configuration file with an easy-to-use syntax inspired bt the Dockerfile and allows the installation of the created systems in a heterogeneous testbed.

The implemented system shows that it is possible to manage different operating system installations in a heterogeneous testbed with the help of different standardized service The creation of similar system to the Docker ecosystem was successful, as the creation and deployment of operating systems is now possible with a concise configuration.

During the evaluation of the system it became clear that the created system allows room for improvement but allows the management, creation, and installation of operating systems inside heterogeneous testbeds.

List of Figures

List of Tables

Bibliography

[Abr+13] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkilä, X. Wang, K. Hamily, and S. Bugoloni. "Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment." In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 2. 2013, pp. 170–175. DOI: 10.1109/CloudCom.2013.121.

[Alb13] R. Alber. *Testimonials*. 2013. URL: https://wiki.fogproject.org/wiki/index.php?title=Testimonials (visited on 08/20/2021).

[Anv08] H. P. Anvin. *PXELINUX*. 2008. URL: http://download.bmsoft.de/archiv/boot/syslinux/doc/pxelinux.txt (visited on 09/10/2021).

[Are21] T. Arends. *Tasmota*. 2021. URL: https://github.com/arendst/tasmota (visited on 07/24/2021).

[Bel05] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. California, USA. 2005, p. 46.

[Boe15] C. Boettiger. "An Introduction to Docker for Reproducible Research." In: *SIGOPS Oper. Syst. Rev.* 49.1 (Jan. 2015), pp. 71–79. ISSN: 0163-5980. DOI: 10.1145/2723872.2723882.

[Bro] N. Brown. *Overlay Filesystem*. URL: https://www.kernel.org/doc/html/v5.14/filesystems/overlayfs.html (visited on 09/10/2021).

[Cha16] e. a. Chad Swanson Wayne Workman. *FOGUserGuide*. 2016. URL: https://wiki.fogproject.org/wiki/index.php?title=FOGUserGuide (visited on 08/20/2021).

[Cox+14] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. O'Brien. "Iridis-pi: a low-cost, compact demonstration cluster." In: *Cluster Computing* 17.2 (2014), pp. 349–358. DOI: 10.1007/s10586-013-0282-7.

[Ela21] Elasticsearch. *Logstash Introduction*. 2021. URL: https://www.elastic.co/guide/en/logstash/7.14/introduction.html (visited on 08/17/2021).

[Esc20] J. Escalante. *Introduction to Packer*. 2020.

[Han07] D. Hankins. *Dynamic Host Configuration Protocol Options Used by PXELINUX*. RFC 5071. Dec. 2007. DOI: 10.17487/RFC5071.

[Inc21] D. Inc. *Overview of Docker Compose*. 2021. URL: https://web.archive.org/web/20210901202112/https://docs.docker.com/compose/ (visited on 09/01/2021).

[Int99] P. Intel. "Preboot execution environment (PXE) specification." In: *Intel Corporation* (1999).

[Joh+18] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox, and J. Singer. "Commodity single board computer clusters and their applications." In: *Future Generation Computer Systems* 89 (2018), pp. 201–212.

[Jor04] H. Jordan. "An introduction to the Intelligent Platform Management Interface." In: *Magazine of Dell Power Solutions* (2004).

[Kac21] M. Kaczanowski. *Packer builder ARM*. 2021.

[Mer14] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[Rot16] S. Roth. *Introduction*. 2016. URL: https://wiki.fogproject.org/wiki/index.php?title=Introduction (visited on 08/20/2021).

[Sol92] D. K. R. Sollins. *The TFTP Protocol (Revision 2)*. RFC 1350. July 1992. DOI: 10.17487/RFC1350.

[YGR19] A. K. Yadav, M. L. Garg, and Ritika. "Docker Containers Versus Virtual Machine-Based Virtualization." In: *Emerging Technologies in Data Mining and Information Security*. Ed. by A. Abraham, P. Dutta, J. K. Mandal, A. Bhattacharya, and S. Dutta. Singapore: Springer Singapore, 2019, pp. 141–150. ISBN: 978-981-13-1501-5.