TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Informatik

Enablement of Kubernetes Based Open-Source Projects on IBM Z

Bachelorarbeit im Studiengang Informatik

vorgelegt von

Sarah Julia Kriesch

Matrikelnummer 303 6764

Erstgutachter:	Prof. Dr. Ralf-Ulrich Kern
Zweitgutachter:	Prof. Dr. Tobias Bocklet
Betreuer:	M.Sc. Alice Frosi
Unternehmen:	IBM Deutschland R & D GmbH

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Vorname: Matrikel-Nr.:

Fakultät: Studiengang:

Semester:

Titel der Abschlussarbeit:

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☐ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,

☐ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Ort, Datum, Unterschrift Studierende/Studierender

Kurzdarstellung

Kubernetes ist eine Container-Plattform zur Orchestrierung mit unterschiedlichen Container-Runtimes, wie Docker, CRI-O und containerd für Hochverfügbarkeits-Cluster. Es gibt Hardware-Abhängigkeiten für die Software. Deshalb sollen alle Dienste und Container-Applikationen darauf basierend auch auf IBM Z Systemen laufen können. Die meisten Kubernetes-basierten Projekte sind Open-Source-Projekte. Sie haben ihre eigene Infrastruktur für automatisierte Tests mit Continuous Integration. IBM Z Hardware kann mit QEMU/libvirt als Hypervisor emuliert werden. So kann der Quellcode immer getestetet werden, so wie es auch schon für die Architektur x86 durchgeführt wird. Außerdem werden auch Fehler schneller erkannt. Das erleichert es den einzelnen Communities Software für spezielle Hardware zu entwickeln. Die Emulation soll in sämtliche CI/CD-Umgebungen der unterschiedlichen Open-Source-Projekte integriert werden können. Mainframes, wie IBM Z, verwenden die s390x-Architektur. Normalerweise haben Open-Source Communities keinen Zugriff für Tests zu solchen Systemen. Deshalb ist Emulation ein durchführbarer Weg diese Communities mit der Befähigung alternativer Hardware zu unterstützen.

Abstract

Kubernetes exists as a container orchestration platform with different container runtimes as Docker, CRI-O and containerd for clustering. There is some hardware dependency for the software. Therefore, all services and container applications should be able to run based on that on IBM Z systems, too. Most Kubernetes based projects are open-source-projects. They have their own infrastructure for automated test environments (Continuous Integration). IBM Z hardware can be emulated with QEMU/libvirt as a hypervisor. In this way, the source code can be tested continuously as it is for x86 architecture and can discover earlier possible bugs. That makes it easier for communities to develop software for special hardware. Emulation should be able to be integrated in all CI/CD environments by different open-source projects. Mainframes as IBM Z are using the s390x architecture. Usually open-source communities don't have access to such systems for builds and tests. For this reason, emulation is a viable way to enable those communities supporting alternative architecture. The first step is to understand their CI/CD infrastructure and how to integrate emulation in their configuration. The investigation on various open-source-projects can help to identify a common pattern how to integrate emulation. Hardware emulation requires a lot of performance. Therefore, minimal system requirements have to be analyzed for the emulation in the next step. This hardware emulation will added into the test environment of both open-source projects Kubernetes and Apache Cassandra then. The goal of this Bachelor Thesis is to apply and integrate emulation for s390x architecture into the infrastructure for various open-source projects. That can be reapplied for other open-source projects then, too.

Contents

Chapter 1.

Introduction

One business introduced by IBM is the mainframe, now known as a Z system. It is possible to run Linux on it. There is a large community behind Linux and open-source projects. Open source does not contain only Linux. There are different applications and other software developed by open-source communities. Mainframes have got a different hardware architecture than a home PC. The Z system architecture has got the name s390x and a default system x86. Z systems are really expensive and not every open-souce community can pay such a system. Most open source Contributers have got systems with the architecture x86 at home. Therefore, it should be possible to test hardware dependencies for s390x on x86.
Our focus is on Kubernetes-based open-source projects in this Bachelor Thesis. Some Kubernetes-based projects should be emulated for Z systems in the CI/CD test infrastructure by these open-source projects. That will be done on systems by these communities. As the first step, the emulator will be chosen with the focus on functionality for Z systems on x86 architecture. After that, Kubernetes is installed in a Docker container. Tests should be able to be run on this system, too. That will be integrated into the emulation environment for an automated start. The CI/CD system should be able to execute all tests then.
The same will be done with the NoSQL database Cassandra for the Apache community to represent the whole system stack from Kubernetes until the application layer for container platforms. Another point are minimal systems requirements and minimal systems sizes. Here are different methods evaluated to minimize the system for emulation.
The goal of this Bachelor Thesis is to offer emulated Z systems for different open-source projects to test their software for hardware dependencies, so that it is possible to release new versions in the CI/CD pipeline of the respective project for running on the hardware architecture s390x without the available hardware. Deployments of the latest software version on Github and running tests have to be automated for that.

1.1. Mainframe Computers

Mainframe computers are large computers. Some of them are part of the Z series[1] by IBM. They are not only used as internet servers or for banking systems. They can handle large numbers of transactions in one second for e-commerce[Tane 14, p.56]. Such Z systems do not use the well known x86 architecture. They are built with s390x. This architecture has been developed by IBM. The current architecture has been introduced in late 2000 and supported by the Linux Kernel since late 1999 [Bloc 19, p.15]. The traditional operating system for mainframes has been z/OS. Linux is used as a base operating system for this Bachelor Thesis.

1.2. Hardware Emulation

Not everybody has access to expensive hardware or hardware with specific architecture. Software should be able to run on the most important hardware architectures. The solution for Software Developers is hardware emulation. You can test based on hypervisors with the hardware emulation whether the software is running correctly. So you can run different operating systems and applications for special hardware in virtualization software. It is possible to enable other hardware architectures than the host has got. That will be done with the implementation of a VM on a host system with a different target architecture[Rose 15, p.3]. In our case it should be possible to run applications for mainframes with the target architecture s390x on a host system with the architecture x86.

1.3. Open Source Projects

1.3.1. Kubernetes

Kubernetes[2] is an open-source project for container orchestration. That is well known as k8s, too. This project was started by Google. A Kubernetes cluster has at least one Master node and one Worker node for high availability. This container platform is portable for private und public clouds. Kubernetes is available as a managed platform by different cloud providers as same as different Kubernetes distributions exist to download or installations from scratch are possible. It is configurable with different container runtimes, as Docker[3], containerd[4] or CRI-O[5] for example. The Container Runtime Interface (CRI) is necessary for managing

[1] https://www.ibm.com/it-infrastructure/z/hardware/
[2] https://kubernetes.io/
[3] https://www.docker.com/
[4] https://containerd.io/
[5] https://cri-o.io/

container images, the life cycle of container pods, networking and help functions [Scho 19, p.16].

1.3.2. Apache Cassandra

Chapter 2.

Emulation

2.1. QEMU

QEMU is an open-source emulator available in most Linux distributions. It is embeeded in different virtualization tools as KVM and XEN, too. So QEMU is well tested and has got all necessary features for emulations. Additionally, you can emulate other architectures on different hardware. QEMU is a generic emulator and can emulate different system architectures. It can also be used for emulation of obsolete hardware[Opsa 13, p.24]. It has been extended for various architectures as x86, ARM, PowerPC, Sparc32, Sparc64, MIPS and s390x. That is a processor emulator using binary translation[Butt 11] which is executing and translating emulated code based on blocks. Each block has got one entry and one exit point[Wang 10, p.5]. The binary translation will be executed with the "Tiny Code Generator" (TCG) inside of QEMU. That is a small compiler replacing GCC because of plenty releases and changes withit. The TCG is converting the blocks of target instructions into a default intermediate form. After that, it has to be compiled for the host or target architecture. If a binary for a new target architecture is necessary, the frontend of TCG will be converted while QEMU is ported to a new architecture. The TCG integrates new code for the new host architecture in the background then. It is also dedicted to improve performance with avoiding repeated translations by buffering already translated code[Cota 17]. The TCG takes care of the emulation of the guest processor running as a thread launched by QEMU. The memory of a quest is allocated during launch and is mapped into the address space of the QEMU process[Opsa 13, p.29].
It is possible to run unmodified guest operating systems. The open-source projects can use any Linux distribution as their base operating system then because QEMU is integrated as a package as default. QEMU does not emulate the whole hardware. That is only possible for the CPU. Therefore, QEMU is used for emulations in this Bachelor Thesis.

2.2. System Emulation

The System Emulation emulates a whole system with hardware, the operating system (with the kernel) and the user space (with application processes). The system (hardware and operating system) will be translated unmodified. The benefit of system emulation (compared to user mode emulation) is that you can run priviledged instructions[Butt 11]. Additionally, it can be used as an application development platform where specific hardware is not available. System emulation with emulated hardware is slower than a real machine because instructions executed directly in the guest hardware have to be emulated in software. That has got multiple host instructions because of the translation for a single guest instruction as a result[Tong 14, p.1]. It is possible to reduce the supported and attached additional devices with special options. The deactivation of a graphic card can be specified with **-nographic** as an example. Afterwards less hardware has to be emulated which has got the result of better performance.
System emulation benefits from the virtualization support as KVM. So most CPU operations are not required to emulate.

2.3. User Mode Emulation

The User Mode Emulation does not emulate the whole system. It is possible to reproduce application processes in QEMU with a minimal system for a special application. This emulation type is working on a syscall level. Therefore, the application has to be runnable as a process executable itself. An external Linux kernel will be built and the application can be mounted via a loaded Docker image in a hard disk image. The emulator is using the Linux kernel to emulate system signal calls then. That can be managed by mapping system calls of the target to an equivalent system call on the host. The user mode emulation can run directly non-priviledged instructions or is using system calls to ask for a special service from the operating system[Butt 11].

2.4. Emulation of different architectures

It is achievable to emulate different architectures on another hardware architecture. The package qemu-user-static has to be installed then and the special architecture has to be registered in binfmt. binfmt_misc is a kernel module. You can register other architectures within that, that multiple other architectures can be run on this host. Not only QEMU can be used for system emulation. Container technologies as Docker have got an integrated QEMU compatibility as an additional emulation feature for building images for other architectures.

That has got the name BuildX. Accordingly a hybrid virtualization approach is practicable with different virtualization technologies as with QEMU and Docker together.

2.4.1. binfmt_misc

Binfmt_misc is a feature in the kernel which allows invoking almost every program by simply typing its name in the shell. That makes it possible to execute user space applications such as emulators and virtual machines for other harware architectures. It recognises the binary-type by matching some bytes at the beginning of the file with a magic byte sequence. These executable formats have to be registered in the file `/proc/sys/fs/binfmt_misc/register`. The structure of the configuration for the registration is the following:
:name:type:offset:magic:mask:interpreter:flags

The **name** is the name of the architecture for the binary format. The **type** can be **E** or **M**. E is for executable file formats as .exe for example. M is used for a format identified by a **magic** number at an absolute **offset** in the file and the **mask** is a bitmask indicating which bits in the number are significant[Slac] (which is used in our case). The offset can be kept empty. The magic is a byte sequence of hexadecimal numbers. The **interpreter** is the program that should be invoked with the binary[Gün]. The path has to be specified for that. The field **flags** is optional. It checks different aspects of the invocation of the interpreter. The **F** flag is necessary in our case for "fix binary". That supposes that a new binary has to be spawned if the misc file format has been invoked.
On this way it is possible to register another system architecture (s390x on x86) on the host system and it is realizable to emulate the other architecture then.
The package **binfmt-support** has to be installed first for using this kernel feature.

2.4.2. qemu-user-static

2.4.3. BuildX

2.5. File Systems

Docker is using the Union File System which is not compatible with QEMU. QEMU can integrate only hard disks formated for default Linux file systems as ext2, ext3, ext4, XFS and Btrfs. The driver virtio_blk is used to mount external file systems and emulates read/write in the physical block device[Barb 18]. Following it is possible to integrate and start nonnative systems in QEMU. Docker is advantageous to setup and start systems fast. So it would be nice to integrate the docker file system into QEMU. After building a docker image, it is

possible export the file system into a directory with the name **rootfs** with the command `docker export $(docker create initrd-s390x)| tar -C "rootfs"-xvf -`.

Linux has got the feature that it is possible to reformat directories for file systems and to copy/ mount content into this one. Reformatting the default docker file system UnionFS to another one as ext4 for example can be done then.
QEMU is accepting the new file system as a block device for the guest system then. The default path to the mounted file system as a hard disk is `/dev/vda` as the first partition[Whit 20, p.22].

2.5.1. Union File System

Docker does not use any default Linux file system. Docker images are based on the Union File System (UnionFS)[Ashr 15, p.21]. This file system is using different file system layers with grouping directories and files in branches. The first layer is the typical Linux boot file system with the name bootfs. That is working the same as in a Linux virtualization stack with using memory at first and unmounting to receiving RAM free by the initrd disk image. So the bootfs can be used inside of another Linux file system to mount in an virtualization stack for a successful boot process with QEMU. The next layer contains the base image with the operating system given by the "FROM" command. Then every docker command inside of the Dockerfile is adding an additional layer with the installation of applications or buildin binary files. That is the reason that every executed docker command is receiving an own id in the disk memory during the build process. Every separate docker command is using his own disk space. So a docker image can grow really fast. It is reasonable to compress so much as possible of different routines into one docker command. All sizes of different docker layers will exist continuously inside of the new file system.

2.5.2. Ext4

Ext4 is a journaling file system (the same as ext3). The journal is registering transactional all changes on the operating system with meta data. So no data are lost after a system failure. They can be restored based on the journal without a save procedure by the user. Such journal file systems in the ext file system familiy are working with blocks[Seuf 15, p.20]. The journal is splitted into the journal super block, descriptor blocks, commit blocks and revoke blocks. The super block contains all meta data of the journal. Descriptor blocks include the special destination adress and a sequence number. Commit blocks are available to flag logged transactions. Revoke blocks are flagging blocks not logged any more.
All meta data from the journal can be relocated in inodes because changes of file system meta data are registered, too.

The difference to the journal in ext3 are the option writing asynchronous commit blocks and additional ckeck sums for journal transactions[Seuf 15, p.28].

Ext4 provides a better performance than ext3. This file system can be formated with mkfs.ext4 which is available in every Linux distribution. This tool is creating a group despriptor table for further group descriptors what allows an expanding of the file system. The file system can grow a maximum of the multiple 1024 of his existing size because of the saved space[Seuf 15, p.21]. Flex groups are used in ext4 for merging different block groups to one logic block group. All data from inodes are only saved in the first flex group then. So the search for files is more powerful because meta data are allocated at one place.

Another feature of ext4 is the possibility of inline files and inline directories. So small files and directories can be saved directly in inodes instead of data blocks. From this follows less disk space consumption[Seuf 15, p.24].

Chapter 3.

Prequisites

Different software is necessary to run qemu or docker for multiple architectures. Therefore, docker and qemu should be installed. Additionally, qemu-user-static [1] and binfmt_misc [2] are important for running multi-architecture containers.
Root permissions are necessary for installation, configuration and running emulated systems.

It is possible to install packages as binfmt-support and qemu-user-static by different Linux distributions, but it is recommended to use the latest possible version for s390x.

3.1. Registration of qemu-s390x

The kernel module binfmt_misc can be mounted with the following command:
`mount binfmt_misc -t binfmt_misc /proc/sys/fs/binfmt_misc`

Latest stable releases of qemu-user-static can be found under https://github.com/multiarch/qemu-user-static/releases/. The release v5.0.0-2 is used for the project and downloaded with `wget https://github.com/multiarch/qemu-user-static/releases/download/v5.0.0-2/x86_64_qemu-s390x-static.tar.gz` for the special version of qemu-s390x-static on x86. That is extracted to the directory `/usr/bin/` with the command `sudo tar -xvzf x86_64_qemu-s390x-static.tar.gz -C /usr/bin/` then.

s390x binaries have to be registered for s390x. That is done with the following commands: `sudo -i` and

```
echo ':qemu-s390x:M::
    \x7fELF\x02\x02\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x02\x00\x16:
    \xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff\xff\xff\xff\xff\xff\
```

[1] https://github.com/multiarch/qemu-user-static
[2] https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html

```
xff\xfe\xff\xff:/usr/bin/qemu-s390x-static:OCF' > /proc/sys/fs/binfmt_
    misc/register
```

Listing 3.1: Register s390x binaries

Docker is configured to build only for the own architecture, which is x86 at open-source projects. The "Experimental" flag exists for new available features which are not ready for production. So you can build docker images for s390x on x86 then. That can be added with the following:

```
{
"experimental": enabled
} >> /etc/docker/daemon.json
```

Listing 3.2: Docker Experimental Flag

After a restart of the docker daemon there should be listed this flag in the command `docker version`. An alternative way is to export this flag as an environment variable in the shell with `export DOCKER_CLI_EXPERIMENTAL=enabled` to enable it. Now it is possible to build docker images for s390x on x86 with `docker build --platform=linux/s390x -t image-example:s390x .` based on a Dockerfile in the existing directory.

3.2. Cross-Compilation of a s390x Linux Kernel on x86

QEMU needs a built Linux kernel to start a system. These packages are necessary:
bison, flex, gcc, gcc-s390x-linux-gnu, libssl-dev
One kernel source code for a certain version has to be downloaded from kernel.org with the following command then: `wget https://git.kernel.org/torvalds/t/linux-5.7-rc7.tar.gz` It can be extracted with `tar -xf linux-5.7-rc7.tar.gz`. The Makefile is in the directory linux-5.7-rc7. Therefore the command `make ARCH=s390 defconfig localyesconfig` has to be executed to create the default configuration for s390x and `make ARCH=s390 CROSS_COMPILE=s390x-linux-gnu- -j6` for the compilation. The kernel has the name bzImage in the directory `arch/s390/boot/` then and can be used in the qemu command.

3.3. Optimized QEMU Command

Every additional device requires additional performance and time for starting the system. So the systems requirements had to be figured out to be minimal for every given open-source project and for running tests on it. That counts for the number of CPUs, too.

The kernel option is receiving the path to the built s390x kernel with the name bzImage. In the given "Optimized QEMU command" the qemu command is executed in the directory with the built Linux kernel and the path is unimportant.
The option **-m** is available to add the minimal guest memory matching the system requirements of every open-source project. **-nodefaults** is deactivating default additional devices activated in QEMU. Only the console is necessary for receiving an output and debugging. So that is added as a device. Cassandra as a project does not need any network interface or parallelism. The option **-nographic** is responsible for not adding any graphical interface. So we save system requirements. The option **-smp** is the minimal number of CPUs for the guest. The file system of containers can be loaded as a hard disk with the option **-hda** which is explained in every chapter of the Bachelor Thesis for a considered open-source project. That is the ideal option to mount a minimal file system for every application or system. `/dev/vda` is the partition name and rdinit is used for using bash as a default shell.
This command can be used for user mode emulation with Apache Cassandra as an example.

```
/usr/bin/qemu-system-s390x -kernel bzImage -m 4G -M s390-ccw-virtio
    ↪ -nodefaults -device sclpconsole,chardev=console -parallel none -net
    ↪ none -chardev stdio,id=console,signal=off,mux=on -mon
    ↪ chardev=console -nographic -smp 3 -hda /data/cassandra.img --append
    ↪ 'root=/dev/vda rw console=ttyS0 rdinit=/bin/bash'
```

Listing 3.3: Optimized QEMU Command

Chapter 4.

Continuous Integration

Hier kommt ein Teil zu CI/CD

Chapter 5.

Kubernetes

5.1. Overview

Kubernetes is a container platform for high availability clusters. There exist plugins for integration tests for Kubernetes with the name kubetest[1]. They contain conformance tests, as well as e2e tests (end-to-end), too. That can be all built and executed on the system. Therefore, a Dockerfile for setting up Kubernetes and building tests with Go is necessary. The challenge is, that 2 big Github repositories have to be cloned and integrated into the docker image. That is using a lot of space. The solution is using a multi staging Dockerfile. So 2 different Dockerfiles are used in one Dockerfile and one of them is used for building. The other one is used for the installation and testing with built tests. At the end the size of the docker image has got only the size of the test image regardless of the repository size in the mother Dockerfile.

5.2. Multi Staging Dockerfile

A multi-staging Dockerfile is using different systems in one Dockerfile for different stages. These systems are receiving special names as indicators with "AS" behind the "FROM" with the base image name. Default this feature is used for building applications in one stage and executing the copied application in another stage. The same counts for clonining Github repositories and building binary files based on it. On this way a lot of space is saved. At the end the docker image has got only the size of the executing system with the application file (without all the code). That is an "experimental feature" at the moment. Therefore the **experimental flag** is necessary to export or set before using it.

[1] https://kubetest.readthedocs.io/en/latest/

5.3. Installation

Kubernetes needs a lot of packages for running and for tests. That will be all installed with the RUN command. `apt.kubernetes.io` has got later versions of Kubernetes than the Ubuntu repository. Therefore this repository has to be added to Ubuntu. kub-build is the name of the mother Dockerfile to be able to copy needed files and directories from there.
In the intallation part of the Dockerfile the Kubernetes repository `https://apt.kubernetes.io` for Debian packages has to be registered together with with the gpg key used by `https://packages.cloud.google.com/apt/doc/apt-key.gpg`. A Dockerfile is installing only necessary packages. Therefore, apt-transport-https, apt-utils, curl, git, ca-certificates, gnupg-agent and software-properties-common have to be installed first for the following installation. The system requires the lates update with `apt-get update` after the registration of the additional repository besides of the default Ubuntu repositories imported with the base Ubuntu image "s390x/ubuntu:18.04" in the FROM command. After that the packages docker.io, kubelet and kubeadm can be installed from kubernetes.io. **Docker.io** contains the container engine docker with all docker commands. CRI/O or containerd would be allowed, too. The Docker daemon will be used because that is the main used container engine of the Kubernetes project and all tests are running withit.
Kubelet is the primary node agent running on each node. He is responsible that different containers can run together in a pod. Pods are deployable units defined in JSON or a yaml file. They include one or a group of containers with shared storage and network resources. Hosting of distributed systems with different services in different containers can work together. Consequential one pod is soething as one "logical host".
Kubeadm is the administration tool to setup clusters. It is necessary to upgrade Kubernetes to other versions, too. Clusters can be initialized. The network can be configured and the command **kubectl** (Kubernetes Control Plane) for adding additional nodes to a cluster can be initialized.
`apt-mark hold` is keeping these special versions of kubelet, kubeadm and kubectl.

```
FROM s390x/ubuntu:18.04 AS kub-build

# The author
MAINTAINER Sarah Julia Kriesch <sarah.kriesch@ibm.com>

#Installation
RUN echo "Installing necessary packages" && \
apt-get update && apt-get install -y \
apt-transport-https \
apt-utils \
systemd \
curl \
git \
ca-certificates \
gnupg-agent \
software-properties-common \
&& curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add - \
&& echo "deb https://apt.kubernetes.io/ kubernetes-xenial main"
> /etc/apt/sources.list.d/kubernetes.list \
&& apt-get update && apt-get install -y \
docker.io \
kubelet \
kubeadm \
&& apt-mark hold kubelet kubeadm kubectl \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
&& systemctl enable docker
```

Figure 5.1.: Kubernetes Installation

5.4. Installation of the latest Go

There were some issues with older Go versions as 1.10 during building tests for Kubernetes. Therefore a higher version (min. 1.13) should be used. It is recommended to use the latest go version for latest Kubernetes tests. It is possible to receive the version number of the latest go release with the command `curl https://golang.org/VERSION?m=text`. This version number has to be included before linux-s390x.tar.gz for downloading the special s390x archive from the go directory by `dl.google.com`. Then the version number has to be called with curl inside of another curl command with the whole path to the special tar

archive on dl.google.com. Every tar archive has got the same structure for every version (`$version.$architecture.tar.gz`). On this way the latest version of Go is integrable into the curl command that it can be installed. Directories for bin, pkg and src have to be created after extracting this tar archive in the `/root/` directory. They are not integrated in the tar archive.

The environment variables for GOROOT, GOPATH and PATH have to be set with ENV on the top of the Dockerfile for successful builds later. PWD is added because Github repositories have to be cloned to this directory.

The ENV variables will be on the top of the Dockerfile. The part for the "Installation of Go" will be attached to the end of the Kubernetes installation part.

```
ENV GOROOT=/root/go
ENV GOPATH=/root/go
ENV PATH=$GOPATH/bin:$PATH
ENV PATH=$PATH:$GOROOT/bin
ENV PWD=/root/go/src/

#Installation of latest GO
&& echo "Installation of latest GO" && \
curl "https://dl.google.com/go/ \
$(curl https://golang.org/VERSION?m=text).linux-s390x.tar.gz" \
| tar -C /root/ -xz \
&& mkdir -p /root/go/{bin,pkg,src}
```

Figure 5.2.: Go Installation

5.5. Building Tests

After a successful installation of go, it is possible to build and install the Kubernetes test environment. At first the directory k8s.io has to be created because kubernetes-tests are looking for this directory as a mother directory. The repository test-infra by the Kubernetes project has to be cloned to there. The repository **test-infra** contains all tests for Kubernetes provided by the community. These can be used of course and are updated continuously. That is the reason to clone this repository inside of the Dockerfile. Inside of this test-infra directory kubetest can be installed with **go install**. That is downloading all available Kubernetes-Tests. So you can use them to test the own Kubernetes cluster and the used software.

The most important tests for the Kubernetes community have got the name conformance tests. These conformance tests are executed with e2e.test which can be built with make inside of the kubernetes repository. Therefore this repository has to be cloned to k8s.io, too. These tests certify the software to comply regular standards. Only with complying these standards, Kubernetes software is allowed to become Kubernetes certified[2].

```
&& cd $PWD \
#Clone test-infra
&& mkdir -p $GOPATH/src/k8s.io \
&& cd $GOPATH/src/k8s.io \
&& git clone https://github.com/kubernetes/test-infra.git
/root/go/src/k8s.io/test-infra \
&& cd /root/go/src/k8s.io/test-infra/ \
#Install kubetest
&& GO111MODULE=on go install ./kubetest \
#Build test binary
&& git clone https://github.com/kubernetes/kubernetes.git
/root/go/src/k8s.io/kubernetes \
&& cd /root/go/src/k8s.io/kubernetes/

CMD make WHAT="test/e2e/e2e.test vendor/github.com/onsi/ginkgo/ginkgo cmd/kubectl"
```

Figure 5.3.: Test Building for Kubernestes

[2]https://github.com/cncf/k8s-conformance

Chapter 6.

Cassandra

6.1. Overview

6.2. Deployment

IBM is offering a Dockerfile[1] for Apache Cassandra especially for Z systems with the latest version on Github. This file can be cloned to the system and the Docker image will be built with the command `docker build --platform=linux/s390x --squash -t cassandra:s390x .` in the directory with the Cassandra Dockerfile for the architecture s390x.

Squash is an option to compress a Docker image and combine commands in a Dockerfile automatically. The prequisites for building s390x images on x86 are set duiring the emulation preparation. The command `docker images` has to show the registered Dockerimage with the name cassandra:390x then. It should be possible to integrate this Docker image into the qemu command. Therefore, a qemu-image will be created with an rounded given size besides of the Docker image in the `docker images` command. So the command `qemu-img create -f raw cassandra.img 2G` can be used. This image needs any Linux file system because QEMU does not know the Docker file system. The image is formated with the command `mkfs.ext4 -F cassandra.img` then. For receiving the file system of the docker image a directory with the name rootfs has to be crated and the command `docker export $(docker create cassandra:s390x)| tar -C "rootfs"-xvf -` is exporting the docker image into the directory rootfs. Following transfers the content of rootfs into the image cassandra.img.

```
mkdir /mnt/rootfs
mount -o loop cassandra.img /mnt/rootfs
cp -r rootfs/* /mnt/rootfs/.
```

Listing 6.1: Mount rootfs

[1] https://github.com/linux-on-ibm-z/dockerfile-examples/tree/master/ApacheCassandra

Now it is possible to run the system with Cassandra:

```
/usr/bin/qemu-system-s390x -kernel bzImage -m 40G -M s390-ccw-virtio
    ↪ -nodefaults -device sclpconsole,chardev=console -parallel none -net
    ↪ none -chardev stdio,id=console,signal=off,mux=on -mon
    ↪ chardev=console -nographic -smp 3 -hda
    ↪ /data/dockerfile-examples/ApacheCassandra/cassandra.img --append
    ↪ 'root=/dev/vda rw console=ttyS0 rdinit=/bin/bash'
```

Listing 6.2: Run Cassandra

6.3. Workaround because of an Issue with JVMs during the Build with BuildX

6.4. Start of the application and tests

A script has been written to start Cassandra and to run tests. Java and Cassandra have to be started here. The Icinga monitoring check has been used for experiments.

Chapter 7.

Outlook

Ich kommt ein Teil zur Zukunft zum Projekt

Chapter 8.

Summary

Hier kommt eine Zusammenfassung

Appendix A.

Supplemental Information

Zusatzinformation

List of Figures

List of Tables

List of Listings

Bibliography

[Ashr 15] W. Ashraf. *The Docker EcoSystem*. Gitbook, 2015.

[Barb 18] D. H. Barboza. "Enhancing QEMU virtio-scsi with Block Limits vital product data (VPD) emulation". October 2018. https://developer.ibm.com/technologies/linux/articles/enhancing-qemu-virtio-scsi-with-block-limits-vpd-emulation/#sec3.1 [Accessed 17 August 2020].

[Bloc 19] B. Block, Ed. *Modern Mainframes & Linux Running on Them*, IBM Deutschland Research & Development GmbH, Chemnitzer Linux-Tage, March 2019. https://chemnitzer.linux-tage.de/2019/media/programm/folien/173.pdf [Accessed 07 February 2020].

[Butt 11] K. Butt, A. Qadeer, and A. Waheed. "MIPS64 User Mode Emulation: A Case Study in Open Source Software Engineering". In: *2011 7th International Conference on Emerging Technologies*, pp. 1–6, 2011.

[Cota 17] E. Cota, P. Bonzini, A. Bennée, and L. Carloni. "Cross-ISA machine emulation for multicores". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 210–220, February 2017.

[Gün] R. Günther. "Kernel Support for miscellaneous Binary Formats (bintfmt_misc)". https://www.kernel.org/doc/Documentation/admin-guide/binfmt-misc.rst [Accessed 22 August 2020].

[Opsa 13] J. M. Opsahl. *Open-source virtualization - Functionality and performance of Qemu/KVM, Xen, Libvirt and VirtualBox*. Master's thesis, University of Oslo, May 2013.

[Rose 15] D. S. Rosenthal. *Emulation & Virtualization as Preservation Strategies*. Andrew W. Mellon Foundation, 2015.

[Scho 19] B. Scholl, T. Swanson, and P. Jausovec. *Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications*. "O'Reilly Media, Inc.", 2019.

[Seuf 15] S. Seufert. *Implementierung eines forensischen Ext4-Incode-Carvers*. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, September 2015.

[Slac] Slackware. "Howtos : Emulators : binfmt_misc". https://docs.slackware.com/howtos:emulators:binfmt_misc [Accessed 20 August 2020].

[Tane 14] A. S. Tanenbaum and A. Todd. *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*. Pearson Studium ein Imprint von Pearson Deutschland, 2014.

[Tong 14] X. Tong, T. Koju, and M. Kawahito, Eds. *Optimizing Memory Translation Emulation in Full System Emulators*, IBM Tokyo Research Laboratory, IBM Japan, Ltd., February 2014.

[Wang 10] Z. Wang, Ed. *COREMU: A Scalable and Portable Parallel Full-system Emulator*, Fudan University, Parallel Processing Institute, August 2010.

[Whit 20] B. White, S. C. Mariselli, D. B. de Sousa, E. S. Franco, and P. Paniagua. *Virtualization Cookbook for IBM Z Volume 5 - KVM*. IBM Redbooks, 2020.