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

Open-Source-Projekte entwickeln Software, bei der der Quellcode frei verfügbar ist. Diese Communitys lassen automatisierte Tests gegen jede Code-Änderung laufen, um die beste Software-Qualität zu garantieren. Diese Tests sollen auf unterschiedlichen Architekturen laufen können. Es ist schwierig, Software für grundlegende Hardware ohne Zugriff darauf zu testen. Deshalb muss die in IBM Z verwendete s390x-Architektur für ausgewählte Open-Source-Projekte emuliert (nachgeahmt) und in die entsprechende CI/CD-Pipeline eingefügt werden. Das sollte mit schnellen Deployment-Methoden im Emulator QEMU durchgeführt werden. Kubernetes wird als containerisiertes Beispiel-Projekt als Grundlage für dafür eingeführte Anwendungen eingesetzt. Ein weiteres Open-Source-Projekt, Apache Cassandra, wird verwendet um Tests auf der Anwendungsschicht im Kubernetes-Stack zu repräsentieren.
Zusätzlich müssen die minimalen Systemanforderungen für die Einrichtung innerhalb der CI/CD-Infrastruktur beider Projekte wegen der Minimierung an Festplattenplatz, Memory und CPU-Verbrauch analysiert werden.
Zum Abschluss wird die automatische Emulation beider Projekte in die Test-Infrastruktur integriert, so dass diese Projekte für die s390x-Architektur von IBM Z Systemen aktiviert sind. Allgemein soll diese Methode für weitere Open-Source-Projekte in der Zukunft wiederverwendet werden können.

Abstract

Open-source projects are developing software with freely available source code. These communities are running automated tests against every code change in order to guarantee the best software quality. These tests should be able to run on different architectures. It is difficult to test software for essential hardware without access. Therefore, the s390x architecture used in IBM Z has to be emulated on x86 for chosen open-source projects and included in their CI/CD pipeline. That should be executed with fast deployment methods in the emulator QEMU. Kubernetes is used as a containerized example project as the foundation for instituting applications. Another open-source project, Apache Cassandra, is applied to represent tests on the application layer in the Kubernetes stack.
Additionally, minimal system requirements have to be analyzed for the setup inside of the CI/CD infrastructure of both projects concerning the minimization of disk space, memory and CPU usage for deployments.
Finally, the automated emulation of both projects will be integrated into the test infrastructure, so that these projects are enabled for the s390x architecture of IBM Z systems. Overall, this method should be reusable for further open-source projects in the future.

Contents

Chapter 1.

Introduction

One business introduced by IBM is the mainframe, now known as a Z system. The Z system architecture is called s390x and a default system x86. In the year 1978 x86 was introduced as a microprocessor architecture originally by Intel with the processor "8086". Afterwards, it has been apapted by AMD [Ostl 20]. The acronym x86 has been tagged quickly for this processor family mostly used for home PCs. The mainframe hardware architecture is different. Not every open-source community has got access to such a Z system. It is possible to run Linux on the mainframe since 1999. There is a large community behind Linux and open-source projects. Open source software does not contain only Linux. There exist different applications and other software developed by open-source communities. Most open-source contributors own systems with the x86 architecture. Therefore, it should be possible to test hardware dependencies for s390x on available systems. The goal of this Bachelor Thesis is to integrate emulated IBM Z systems for different open-source projects to test their software for hardware dependencies so that it allows to release new versions in the CI/CD pipeline of the respective project for running on the architecture s390x without the available hardware. Deployments of the latest software version on Github and running tests have to be automated for this case.
The focus is on Kubernetes based open-source projects. These should be emulated for IBM Z systems in the CI/CD test infrastructure by these chosen 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 IBM Z systems on x86 architecture. Afterwards, Kubernetes is installed in a Docker container. Additionally, tests should be able to be run on this system. Kubernetes and Docker are both written in Go. The full system setup will be integrated into the emulation environment for an automated start. In conclusion, the CI/CD system should be able to execute all tests.
The same will be done with the NoSQL database Cassandra, which is Java-based, for the Apache community to represent the whole system stack from Kubernetes until the application layer for container platforms. Other points are minimal systems requirements and minimal systems sizes. Here are different methods evaluated to minimize the system for emulation.

1.1. Mainframe Computers

Mainframes are called as high-performance computers that process billions of simple calculations and transactions in real-time [IBMb]. Some of these computers are part of the Z series[1] by IBM. They are not only used as internet servers, but also for mission-critical apps and blockchain. The IBM Z system is gladly utilized in sectors for banking, finance and insurances. Mainframes can handle a large number (2020: millions [IBMa]) of transactions in one second [Tane 14, p.56]. Thousands of virtual machines can run on such a system [Linu]. IBM Z systems do not use the well known x86 architecture. They are built with s390x. The current architecture version has been introduced in late 2000 by IBM [Bloc 19, p.15]. The Z system as a mainframe has built in the fastest available processor with 5 GHz with on core and on chip cache, extensive memory and dedicated I/O processing [Linu].
IBM Z contains "IBM Z pervasive encryption" for comprehensive protection around the data on the system [Lasc 20, p.4]. Such systems are offering a horizontal and upright scaling of processes, which allows the operation of many hundred virtual systems in parallel [Tsch 09, p.13]. One traditional operating system for IBM Z systems has been z/OS. IBM Z systems have been optimized for open-source software as Linux [Lasc 20, p.8]. The goal by IBM has been to offer a combination of a robust and securable hardware platform with the power of different Linux distributions. The IBM Z system allows running multiple operating systems at the same time with the support of LPARS. An LPAR is a logical partition to separate the mainframe for the corresponding operating system with associated applications.
Ubuntu Linux is used as a base operating system for this Bachelor Thesis.

1.2. Hardware Emulation

Not everybody has access to hardware with essential architecture. The software should be able to run on most relevant hardware architectures. A solution for Software Developers is hardware emulation. You can test with the hardware emulation whether the software is running correctly. Accordingly, you can run different operating systems and applications for specialized hardware in emulators or virtualization software. It is possible to enable other hardware architectures than the one of the host. That will be done with the implementation of a guest system on a host 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] `https://www.ibm.com/it-infrastructure/z/hardware/`

1.3. Kubernetes Based Open-Source Projects

The focus in this Bachelor Thesis is on Kubernetes based open-source projects. Open-source projects are communities developing software with public accessible source code. Contributors are mostly a mixture of paid developers in companies and volunteers with open source as their hobby.

Kubernetes is a container orchestration platform as described in 1.3.2 "Kubernetes". The advantage of containers is that every service can interact isolated in a separate container independent of the development environment. This technology simplifies the management and deployment of new software versions for particular components. Furthermore, the setup is faster with predefined installation workflows.

Kubernetes is developed based on Docker as the first container engine. Docker provides connected setups for different services in detached containers. Kubernetes integrates cluster functionality for containers. 1.3.3 "Apache Cassandra" is selected as an example application for containerization.

1.3.1. Docker

Docker[2] is one possible container engine used by Kubernetes. This technology emerged in 2013 as a base for future container technologies. The difference to existing container technologies (LXC as an example) has been the possibility of distributing systems. Docker Inc. has established a public container registry with the name Docker Hub for public usable Docker images. That should facilitate setups and the entrance into work with containers. Another benefit of Docker is that all installation and configuration steps for a system are described in one reusable text file. This file (Dockerfile) is the foundation of Docker images and most public Docker images on Docker Hub are maintained. Docker images are allocatable systems obtained from a container registry with `docker pull`, or built with

```
docker build -t ${name} .
```

based on a self-written Dockerfile inside of the current directory for the local registry on the server.

`-t` is the tag and defines the name of the docker image listed in the registry with the command `docker images`. Only successful builds of images can be registered in such a container registry.

The **Dockerfile** contains different instructions for building steps. The line with the "FROM" command defines the base image. Every public or local usable Docker image can be used as a base image with the whole operating system incl. pre-installed packages and relevant configuration. Docker images include only a minimal operating system defined in the Dockerfile

[2] https://www.docker.com/

of the base image together with all listed packages for installation. These installations are listed in the Dockerfile with the command "RUN" before apt, yum, pip or other installation commands. After the build, all commands are registered in a Docker manifest within JSON format together containing all information about the Docker image. If a new application should be executed inside of the Docker container, then creating a directory for this application besides the Dockerfile is possible. The command "ADD" can integrate this application into the Docker container during the build process. Such a line has the following structure:

```
ADD ./dir/app.py /app.py
```

This application can be started with the "CMD" command then:

```
CMD ["python","/app.py"]
```

It is recommended to define one service or process for one single container and to connect all containers for a start then. One single container is launchable with the command `docker run ${name}`. In this case, `${name}` can be the name of the image or the image id. The command `docker-compose` is available to automate the setup with multiple connected containers.

The Dockerfile format has been spread as an easily understandable base for most container runtimes to create container images. Therefore, most container runtimes support this format. Docker has turned out to be more like a developer tool for many container technologies in the last years. Consequently, as an example, Kubernetes is using Docker for their community tests as a base container runtime. Most new container orchestration platforms are developed based on Docker. The difference is the replacement of the docker command and the expansion with additional features for clustering and specialized configurations.

1.3.2. Kubernetes

Kubernetes[3] is an open-source project for container orchestration, also known as k8s. Google started this project. A Kubernetes cluster has at least one Master node and one Worker node for availability. One node is one host. Both can be on the same host. The Master node is responsible for managing all Worker nodes with applications. All configurations and distributions are performed from there. New Worker nodes are added to the Kubernetes cluster with "join" on the Master node, too. The Worker node is deploying pods with their containers with different services. Every Worker node can communicate with other Worker nodes in the cluster.
Kubernetes is scalable for using containers as distributed services in a pod. A pod is representing something as a single server split into different connected containers with all services for a running application. Pods can be replicated to multiple nodes for high availability. Kubernetes is configurable with different container runtimes, like Docker, containerd[4] or

[3] https://kubernetes.io/
[4] https://containerd.io/

CRI-O[5], for example. The Container Runtime Interface (CRI) is necessary for managing container images, the life cycle of container pods, networking and help functions [Scho 19, p.16]. The most used container runtime is Docker in the Kubernetes project. The image of a container with the full installation and configuration is described in the format of a Dockerfile for the Docker image.

1.3.3. Apache Cassandra

Apache Cassandra[6] is a NoSQL database management system developed by the Apache Foundation. The project had been started internally at Facebook and has been released as an open-source project in 2008. Cassandra provides continuous availability, high performance, and linear scalability besides offering operational simplicity and effortless replication across data centres and geographies [Data]. It is recommended for mission-critical data. The Cassandra Query Language (CQL) is similar to SQL and includes JSON support. The similarity simplifies migrations from relational database management systems to Apache Cassandra.
CQL includes an abstraction layer that hides implementation details of the structure.
NoSQL database stores are using other methods than relational database management systems for data access and own a different structure. Cassandra is using hashing to fetch rows from the table as a column-oriented database management system (DBMS). Such a DBMS stores data tables by column rather than by row.
NoSQL databases were developed as massively scalable database management systems that can write and read data anywhere while distributing availability to billions of users. This field is a part of Big Data.

[5] https://cri-o.io/
[6] https://cassandra.apache.org/

Chapter 2.

Emulation

2.1. QEMU

QEMU is an open-source emulator introduced by Fabrice Bellard in the year 2003 and available in most Linux distributions now. It is embedded in different virtualization tools as KVM and XEN, too. QEMU is well tested and contains all the necessary features for emulations of other architectures on alternative hardware. QEMU is a generic emulator for 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. It is a processor emulator using binary translation [Butt 11] which is executing and translating emulated instructions based on basic blocks. Each block comprises 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. The TCG is a small compiler replacing GCC because of unlimited releases and code changes. The TCG is converting the blocks of target instructions into a standardized form with machine instructions of the host hardware. Subsequently, it has to be compiled for the host or target architecture for the guest system. 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 dedicated to improving 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 guest is allocated during launch. Then that 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 default package. QEMU does not emulate the entire hardware. That is only possible for the CPU. QEMU is used for emulations in this Bachelor Thesis.

2.2. Full-System Emulation

The full-system emulation emulates a whole system with hardware, the operating system (with the Linux kernel) and the userspace (with application processes). The system (hardware and operating system) will be translated unmodified. The condition of system emulation (compared to user mode emulation) is that you can run privileged instructions [Butt 11, p.2]. This feature enables the translation of the unmodified target code in the operating system. That all leads to a slowdown of the emulation in comparison to the user-mode emulation. Additionally, this emulation 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 should be executed in the guest hardware. However, that has to be emulated in software. That implies multiple host instructions as a result because of the translation for a single guest instruction [Tong 14, p.1]. It is possible to reduce the supported and attached additional devices with additional options. The deactivation of a graphic card can be specified with **-nographic** as an example. Afterwards, less hardware has to be emulated, which has given a better performance.
System emulation benefits from the virtualization support as KVM, if the guest has the same architecture as the host. In this case, CPU operations are not required to emulate. Emulations of alternative architectures (see 2.4 "Emulation of Alternative Architectures" require additional resources for the emulation of the whole system.

2.3. User-Mode Emulation

The user-mode emulation does not emulate the whole system. It is faster than the full-system emulation because it does not engage so much hardware resources. Application processes can be run in QEMU with a minimal system for a specific application. This emulation type is working on a system call level. Therefore, the application has to be runnable as a single process executable itself. The emulator is using the Linux kernel to emulate system calls then. That can be managed by mapping system calls of the target system to an equivalent system call on the host with threading (with a separate virtual CPU) [QEMU]. This process does not require the emulation of the full memory management unit [Butt 11, p.2]. The user-mode emulation can run directly non-privileged instructions or is using system calls to ask for a selected service from the operating system.
The disadvantage of user-mode emulation is the ability to run only single processes. Most services contain different applications with the result of multiple processes. Consequently, these applications need a full system emulation.

2.4. Emulation of Alternative Architectures

It is achievable to emulate alternative architectures on another hardware architecture. The package qemu-user-static has to be installed then, and the chosen architecture has to be registered in binfmt_misc. binfmt_misc is a kernel module. You can register other architectures within that, so that multiple other architectures can be run on one host. Not only QEMU can be used for system emulation. But also, container technologies as Docker include an integrated QEMU compatibility as an additional emulation feature for building images for other architectures. This technology has got the name 2.4.3 "BuildX". Accordingly, a hybrid virtualization approach is practicable with different virtualization and emulation technologies as with QEMU and Docker together. In this case, an external Linux kernel will be integrated into QEMU, and the application can be mounted via a loaded Docker image in a hard disk image. The Linux kernel is required for QEMU and should be built or downloaded (see 3.3 "Fetching Linux Kernel Image").
The initrd should match the Linux kernel and is optional. It is used to start an init process together with the required test scripts.

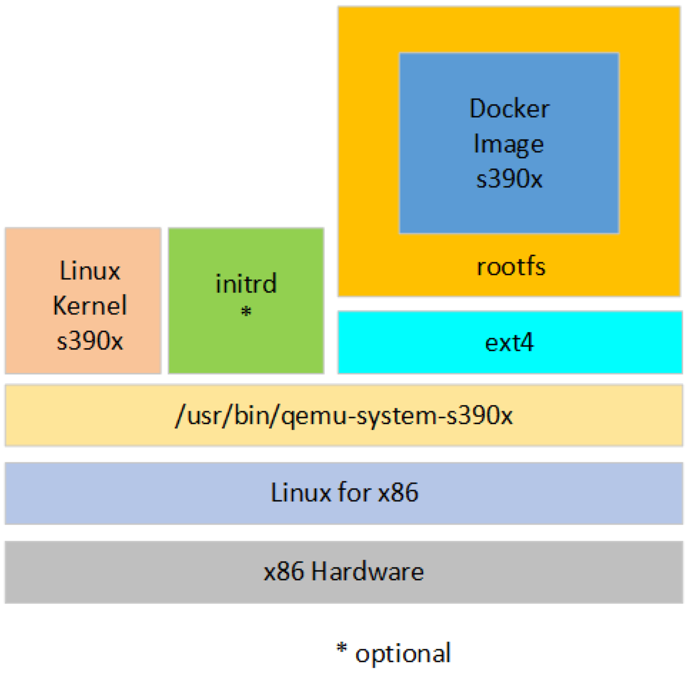

Figure 2.1.: Emulation with QEMU

2.4.1. Binfmt_misc

Binfmt_misc is a kernel module which allows invoking almost every program by simply typing its name in the shell. That permits to execute user space applications such as emulators and virtual machines for other hardware architectures. It recognizes the binary-type by matching some bytes at the beginning of the file with a magic number. 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 utilized 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 it. The field **flag** 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.
In 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 afterwards.
The necessary command for s390x registration can be found under 3.1. The package **binfmt-support** has to be installed first for using this kernel feature.

2.4.2. Qemu-user-static

The package **qemu-user-static**[1] enables the execution of multi-architecture container emulation based on **binfmt_misc** with **QEMU**. This package contained in various Linux distributions includes a set of binfmt configurations for various architectures together with an amount of statically compiled QEMU emulators that applications compiled for alternative architectures can run on another architecture [Yang 19]. When building for one new architecture, static binaries are relevant because of the possibility of an "exec init error" without them.
The installation contains emulators for all available architectures supported by the QEMU project incl. s390x aligned with the specific host architecture. In this manner, you can build and run a binary for different architectures on the same host (see 3.4 "Optimized Qemu Command"). That is the base for 2.4.3 "BuildX" by Docker, too.

[1] `https://github.com/multiarch/qemu-user-static`

2.4.3. BuildX

As mentioned in 2.4.2 "Qemu-user-static", Docker is using QEMU for emulations with BuildX[2]. That is an integrated "experimental Feature" since version Docker 19.03. That implies that it is not enabled as default because it is very new and for experiments. It has to be activated with `DOCKER_CLI_EXPERIMENTAL=enabled` (see 3.1 "Registration of Qemu-S390x"). The disadvantage of the provided BuildX inside of Docker is that you do not receive the up-to-date version of BuildX. There is a version with a more detailed output and better working with an additional **buildx** behind **docker build**. This version can be cloned from `https://github.com/docker/buildx` and installed with **make install**.
In general, BuildX is a Docker CLI plugin for expanding the "docker build" command. That includes the possibility of multi-architecture builds and exceptional output configuration. That involves not only the upload possibility to your local docker registry (see the output of **docker images**). That includes the output of "docker build" (which creates a docker image based on the requirements and installations in the used Dockerfile) into a local directory, a tar archive or a public registry on Docker Hub. Docker Hub is the most used and maintained registry for Docker images.
The special architecture for 2.4.4 multi-architecture builds can be specified with the additional option **–platform**. The option **build create** provides the option to create different build instances for multiple build combinations of architectures.

2.4.4. Multi-Architecture Images

Default Linux distribution providers have to package their software for different system architectures because of different required drivers included in the Linux kernel and dependecies to these. That is the reason that Docker images have to be built equally for different architectures. That can be performed with different base images for the respective architectures or with multi-arch images. It is possible to use one Dockerfile for different architectures then. One example for such a multi-arch Docker image is adoptopenjdk/adoptopenjdk8 based on Ubuntu[3]. In this case hardware dependencies are selected with `dpkg --print-architecture` and included as ARCH into the different cases for downloading the relevant tar archive of OpenJDK:

[2] `https://github.com/docker/buildx`
[3] `https://github.com/AdoptOpenJDK/openjdk-docker/blob/master/8/jdk/ubuntu/Dockerfile.hotspot.nightly.full`

```
RUN set -eux; \
    ARCH="$(dpkg --print-architecture)"; \
    case "${ARCH}" in \
       aarch64|arm64) \
         ESUM='2c6540ff8ea3d89362fd02143b24303e2359be249a6be6cb7e6580472686d863'; \
         BINARY_URL='https://github.com/AdoptOpenJDK/openjdk8-binaries/releases/
         download/jdk8u-2020-08-05-08-20/OpenJDK8U-jdk_aarch64_linux_hotspot_2020-
         08-05-08-20.tar.gz'; \
         ;; \
       armhf|armv7l) \
         ESUM='3c57c121415ef721ba9c73dfa809cd2f689d7369ef3d92d055f7c7c81ed2d697'; \
         BINARY_URL='https://github.com/AdoptOpenJDK/openjdk8-binaries/releases/
         download/jdk8u-2020-07-31-04-43/OpenJDK8U-jdk_arm_linux_hotspot_2020-07-
         31-04-43.tar.gz'; \
         ;; \
       ppc64el|ppc64le) \
         ESUM='668972e8d6d0815c03b32dac3f5c663453b4b0842938d13cfb0c1232a26a8087'; \
         BINARY_URL='https://github.com/AdoptOpenJDK/openjdk8-binaries/releases/
         download/jdk8u-2020-08-05-08-20/OpenJDK8U-jdk_ppc64le_linux_hotspot_2020-
         08-05-08-20.tar.gz'; \
         ;; \
       s390x) \
         ESUM='045aa287c48fd84eef19f2828ed1d15d27059a9328f5ca2cc91f6d42489af956'; \
         BINARY_URL='https://github.com/AdoptOpenJDK/openjdk8-binaries/releases/
         download/jdk8u-2020-08-05-08-20/OpenJDK8U-jdk_s390x_linux_hotspot_2020-
         08-05-08-20.tar.gz'; \
         ;; \
       amd64|x86_64) \
         ESUM='c26124b14c6a42d89278d3ce108b43c1210a317aba0163d985c06a79a695a174'; \
         BINARY_URL='https://github.com/AdoptOpenJDK/openjdk8-binaries/releases/
         download/jdk8u-2020-08-05-08-20/OpenJDK8U-jdk_x64_linux_hotspot_2020-08-
         05-08-20.tar.gz'; \
         ;; \
       *) \
         echo "Unsupported arch: ${ARCH}"; \
         exit 1; \
         ;; \
    esac;
```

Figure 2.2.: Multi-Arch Image AdoptOpenJDK 8

You can see all URLs for arm64, armv7l, ppc64le, s390x and x86 in this case. This RUN command in the Dockerfile is using the architecture given by the option
`--build-arg ARCH=s390x`
as an example in the `docker build` command or the `--platform` option will be used with BuildX. `Docker build` is stacked against BuildX, because only one image can be built with one command. Buildx provides the possibility to consign different architectures (comma separated) to the platform option. Thereby, one command can build multiple images for different architectures.

2.5. File systems

Docker is using the Union File system which is not compatible with QEMU, because it is not based on block devices as ext4 as an example.
QEMU can integrate only hard disk formats for default Linux file systems as ext2, ext3, ext4, XFS and Btrfs. The driver virtio_blk is used for mounting external file systems and emulates read/write in the physical block device[Barb 18]. Following this, it is possible to integrate and start nonnative systems in QEMU. Docker is capable of setting up and starting containers fast.

Linux has got the feature that it is adventitious 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` for the first partition [Whit 20, p.22].

2.5.1. UnionFS

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 performs 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 building binary files. Docker is using only these specified sections of the file system for the container image. 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.
The example Docker image for Ubuntu 18.04 on s390x is using the base image scratch as a super minimal system and is adding the tar archive with the rootfs for Ubuntu Bionic on S390x, which is the version Ubuntu 18.04. That will be installed as a minimal Ubuntu system then.

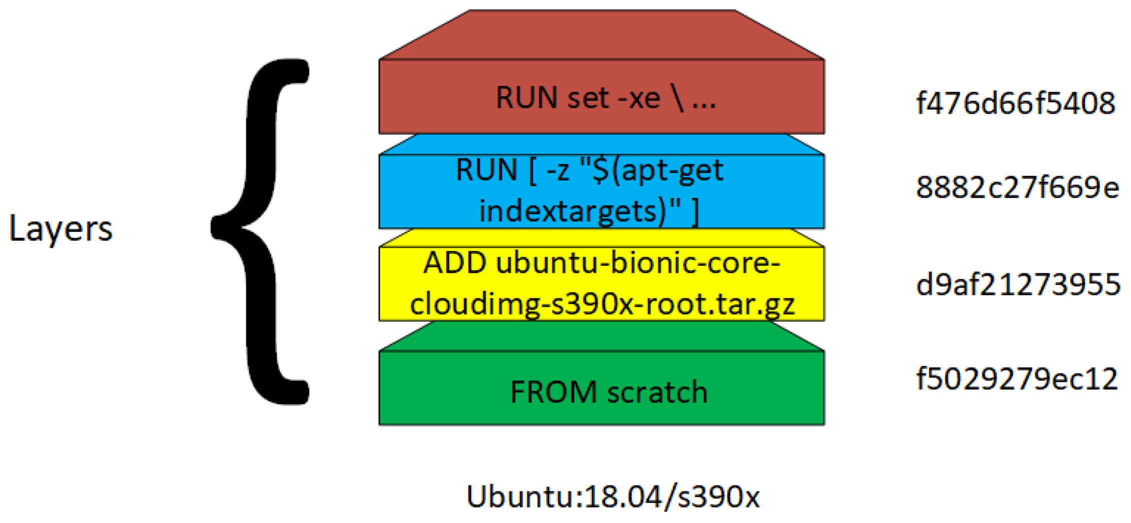

Figure 2.3.: Union File system

2.5.2. Ext4

Ext4 is a journaling file system (just 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 family 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. Flex groups can combine

multiple group blocks - inode bitmaps, inode tables and data blocks - to one logic block group. All data from inodes are only saved in the first flex group then. So the search for files is speedier because meta data are allocated at one place.
All meta data from the journal can be relocated in inodes, if 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 check sums for journal transactions [Seuf15, 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 [Seuf15, p.21]. 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 [Seuf15, p.24].

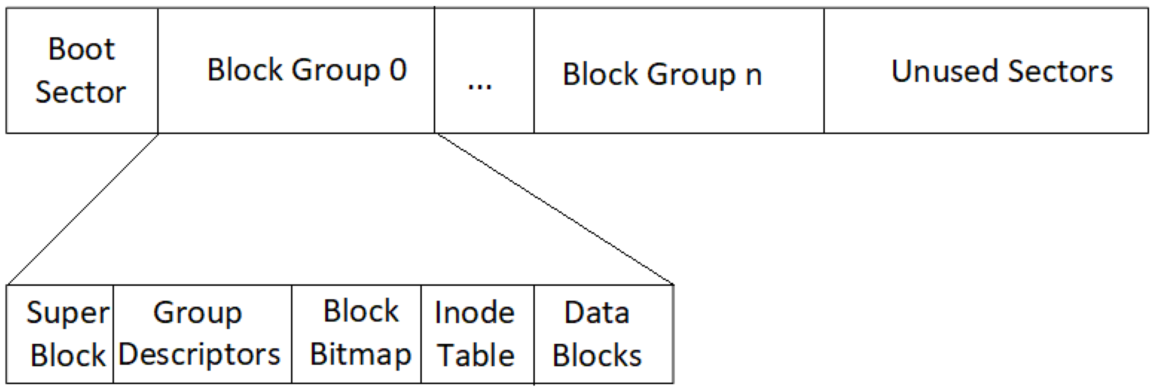

Figure 2.4.: Ext4

2.5.3. Converting Docker Image to External File System

It would be nice to integrate the docker file system into QEMU. After building a docker image, it is realizable to export the file system into a directory with the name rootfs with the command
`docker export $(docker create image-s390x)| tar -C "rootfs"-xvf -`.

The exported file system has got the format of the 2.5.1 "UnionFS" at this moment. QEMU contains the beneficial tool **qemu-img** for creating loadable images of file systems. These require the minimal size of the docker image. The output of the size is given by `docker ↪ images | grep ${id}`. Qemu-img understands only rounded numbers. Therefore, that has to be rounded up to a full number. That will be done automated with the tool **awk**. In the case of Cassandra, there will be used
`docker images | grep 'Cassandra-s390x' | awk '{print int($7+0.5)"G"}'`.

`print $7` would give the output of the size raw of `docker images`, which is a float number. The **int** is rounding down to an integer number. Accordingly, the **0.5** has to be summed up for rounding up. The GB has been removed in this process. `qemu-img` requires a G behind the number. That can be added with "G" inside of the awk command.

Finally, Cassandra is using the command
`qemu-img create -f raw cassandra.img 2G` (see 6.6 "Running the Docker Image")
for the creation of a file system image with a Docker image size about 1.2G then. This size is given by the output of `docker images`. "2G" can be replaced by the command above in the CI/CD pipeline then (see A.1 "CircleCI configuration for Apache Cassandra").
At first, cassandra.img is empty. The content of **rootfs** has to be transferred into this image file. But QEMU does not know UnionFS as a file system. That is the reason for formatting cassandra.img with `mkfs.ext4 -F cassandra.img` to ext4 for an emulated hard disk. Writing into cassandra.img is only executible if the image is mounted into a directory under `/mnt/` with
`mount -o loop cassandra.img /mnt/rootfs`.
In the next step the content of the Docker file system directory can be copied into `/mnt/rootfs/` with
`cp -r rootfs/* /mnt/rootfs/.`.
Afterwards, the UnionFS is converted to ext4. All data of the Docker image are saved into cassandra.img because it is mounted into `/mnt/rootfs/`. In conclusion, the image cassandra.img is usable for deployments with **qemu-system-s390x** (see 6.4 "Run Cassandra"). The Cassandra image can be unmounted with `umount /mnt/rootfs` later.

Chapter 3.

Prerequisites

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 required 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 version for s390x.

3.1. Registration of Qemu-S390x

The packages **qemu-user-static** and **binfmt-support** should be installed because it contains all binfmt configuration files for various architectures in the directory `/usr/lib/binfmt.d/` [Yang 19].

The Linux kernel module binfmt_misc can be mounted with the following command:

```
mount binfmt_misc -t binfmt_misc /proc/sys/fs/binfmt_misc
```

The installed version 3.1 of qemu-user-static on Ubuntu 18.04 contains bugs that Docker images for s390x are not buildable on x86. Therefore, an upgrade to a new release is necessary. Ultimate stable releases of qemu-user-static can be found under `https://github.com/multiarch/qemu-user-static/releases/`. The release v5.1.0-2 is used for the project and downloaded with

```
wget https://github.com/multiarch/qemu-user-static/releases/download/v5.1.0-2/
x86_64_qemu-s390x-static.tar.gz
```

for the specific version of qemu-s390x-static on x86. This static binary in the archive is extracted to the directory `/usr/bin/` with the command

```
sudo tar -xvzf x86_64_qemu-s390x-static.tar.gz -C /usr/bin/
```

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

afterwards. The architecture name before qemu is the architecture of the host system. The name between qemu and static is the emulated architecture. This archive contains only the new version for the unique architecture. The configuration will be used by the installed qemu-user-static.

s390x binaries have to be registered for s390x that they can be executable on x86, as described in refBinfmt "Binfmt_misc". That is executed with the following command: `sudo -i` and

```
echo ':qemu-s390x:M::\x7fELF\x02\x02\x01\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\xff\
xfe\xff\xff:/usr/bin/qemu-s390x-static:OCF' > /proc/sys/fs/binfmt_misc/register
```

Figure 3.1.: Register S390x Binaries

3.2. Enabling Multi-Architecture Images

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. It can be added with the following configuration to the Docker daemon after the installation of docker:

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

Figure 3.2.: Docker Experimental Flag

The configuration of the Docker daemon is written in the JSON format. That is saved in `/etc/docker/daemon.json`. After a restart of the docker daemon, there should be listed this experimental 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.

Additionally, a new version of BuildX is required. The pre-installed version of Docker has got bugs with the consequence of a broken build process with **Illegal instructions** at Go applications. This bug is fixed with the new version on Github.
Therefore, the new version has to be installed with:

```
git clone git://github.com/docker/buildx && cd buildx
make install
```

Figure 3.3.: BuildX Upgrade

3.3. Fetching Signed Linux Kernel Image

QEMU needs a built Linux kernel to start a full system emulation. There are the two options, to build the Linux kernel repeating because of security updates or to fetch the matching Linux kernel for the Docker image from the associated repository of the corresponding Linux distribution.
The build process via CI/CD should be fast. Building the Linux kernel would require a lot of time with a minimum of 1 hour. Therefore, the second option will be used. The Docker file system in rootfs does not contain any Linux kernel in the directory `/boot/`. The prequisite for fetching of a signed Linux kernel image for a minimal system is the installed package **debootstrap**. This command line tool can download the file system of a minimal system to a sub directory on a Debian based system. There are additional options to specify the architecture with `--arch=s390x` in our case and to include the Linux kernel with `--include=linux-generic`. To receive only a minimal system, the additional option `--variant=minbase` should be used. The whole command is the following then:

```
debootstrap --include=linux-generic --arch=s390x --variant=minbase bionic \
kernel-bionic http://ports.ubuntu.com
```

Figure 3.4.: Debootstrap

This command "debootstrap" is validating all downloaded packages additionally. This process is easy to automate and requires only 10 seconds. In conclusion, the generic and signed Linux kernel image can be used in the `/boot/` directory for the integration into QEMU.

3.4. Optimized QEMU Command

Every additional device requires additional resources 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 s390x Linux kernel image. In the given 3.5 "Optimized QEMU command" the qemu command is executed in the main directory with the

Linux kernel. This one is transferred from the boot directory to the main directory during the automation process
The option **-m** is available to add the minimal guest memory matching the system requirements of every open-source project. **-M s390-ccw-virtio** defines the machine type for s390x. **-nodefaults** is deactivating default additional devices activated in QEMU. Only the console is necessary for receiving an output and debugging. So this one is added as a device with **-chardev stdio,id=console,signal=off,mux=on -mon chardev=console** and **-device sclpconsole,chardev=console**. Additionally, the console has to be specified with **console=ttyS0** in the append part.
Cassandra as a project does not need any network interface (-net none) or parallelism (-parallel none). 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 the chapters for considered open-source projects. 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’
```

Figure 3.5.: Optimized QEMU Command

Chapter 4.

Continuous Integration

4.1. CI/CD in the Software Development Lifecycle

CI/CD is the short name for Continuous Integration/Continuous Delivery. This method contains systems and technologies used for Test Driven Development. Continuous Integration provides executing changes in the code, building them to an application and testing it directly. If all is working fine, it can be merged. Continuous Delivery is adding automated releases to the process. The software is tested automatically by the system beforehand. If all is ok, it can be released.

These tests during the software development lifecycle are integrated into all sprints for agile software development. That has got the name DevOps after adapting these practices with expanding with automated deployments for operations. It is playing an important role through all phases in the software development because of the quality improvement of code. The code complexity is growing with the size of the project. Therefore, it is beneficially to have automated tests running directly after the commit.

In the past, all systems had to be setup manually. Automating the software cycle and Test Driven Development (TDD) have promoted software for configuration management, CI/CD and version control. In combination, software can be deployed and tested over the full software development lifecycle. The Developer has the possibility to test his new written software before submitting and is receiving the feedback by the CI/CD system for the integration and system test whether the software is working in cooperation with other modules on multiple systems with their base operating systems.

This process should be automated. "Infrastructure as Code" is a provision for managing the setup of systems, networks and services via written code [Scho 19, p.110]. Deployable systems are described in configuration files how to be launched eith this method. These servers are installable on real servers in any data centre or in the public cloud. Using code as a base ensures that all systems have been installed and configured on the same way matching the system requirements.

4.2. Test Levels

Test suites can be executed repeatable and consistent with an point of view on the functionality, new requirements and software quality. These automated tests are part of the CI/CD pipeline with integrated deployments into system environments [Scho 19, p.112].
Test suites can be splitted into different test levels:

- **Component Test**
  The component test can be called additionally unit test, module test or class test. It is testing the functionality of a special software component developed by the Developer [Spil 19, p.66]. He has to extend the unit test with tests for new features for code coverage. These tests are mostly based on requirements and design documents [Spil 19, p.63]. In addition, the developed test cases of the component test have an effect on maintainability and efficiency. The preparation with "Test First" (writing tests before software development) is called Test Driven Development and is improving the code quality continuously.

- **Integration Test**
  The integration test is testing the matching of different modules. This process is called software integration [Spil 19, p.71]. Some modules are using APIs (Application Programming Interface) for communication or have to correspond with other services. Devices, the output and performance have to be checked in this level. This test level can be performed with multiple connected containers for different services/ modules. Containers can represent sub systems. Bugs in interfaces and communication issues can be detected on this manner. All in all, the system architecture of the complete system can be tested with integration tests.

- **System Test**
  The system test integrates all hardware and system requirement for the product [Spil 19, p.79]. Every customer can use different hardware or services as a foundation. That can expect special system settings and can have an effect on the behavior of the system. This test level includes tests for functional sustainability [Spil 19, p.87] and non-fuctional sustainability. Performance testing, usability testing, security testing and compliance testing are part of non-functional sustainability. But also the system compatibility and tests with multiple configurations suit to non-functional tests.
  Many software manufacturers are using live environments by their customers because of missing hardware for this test level. Integrated system emulations into the CI/CD pipeline would solve this problem.

- **Acceptance Test**
  The acceptance test requires users. The User Experience has to be tested on this test level. Mostly the Product Management and/ or the Customer Support are appointed

for this task after the Alpha and Beta release. These release phases are executed before the final release for the possibility to fix critical or major bugs. Some software companies are offering such releases for free, if a customer wants to test the software in the customer environment.

This order of test levels has to be kept. 3 of 4 phases can be automated with CI/CD.

4.3. CI/CD Systems

There is no one CI/CD system. These systems exist as open-source software and commercial software. They can integrate version control systems for automated tests of new software code or for their releases. Most common used CI/CD systems are Jenkins, CircleCI, Bamboo, TeamCity, GitLab CI and Travis CI[1].

CI/CD systems can split their tests in different test cases or 4.2 "levels". Developers can define which test should run on which branch and in which phase. A branch is a code copy in a version control system as Github, Gitlab or SVN. It supports teams working parallel on the same code and to manage that. The code contribution by a single Developer should be tested and approved before the merge into the master branch. That can be triggered via CI/CD after the Pull Request. The Pull Request is a request for merging the offered code contribution to the master branch. This test before the merge is in the CI/CD stage for component tests. Afterwards, integration tests can be adjoined in the next stage.

Repeating test cases after software changes is called regression test [Spil 19, p.98]. This practice is checking whether new code is matching the requirements and does not implicate accidentally side effects. That is the best release method with using the CI/CD system in comparison with manual releases.

Most developed CI/CD systems are web applications connected to the code base. All test cases are listed in the application. If the code matches to the respective test, the test case has the output "Passed" and the next test case will be executed. If the representation indicates "Failed", a bug has been found or the code does not fit the requirements of the functionality any more.
Some systems can handle only the Continuous Integration part. Others are developed to manage all in the CI/CD pipeline. The Release Manager or Product Manager can determine which test cases should be iterated in the CI/CD pipeline. Open-source projects have been appointing maintainers for this role and managing the CI/CD system.

It is recommended to prepare separate environments for testing and production in the background [Scho 19, p.120]. Usually that will be done with virtual machines or containers

[1] `https://medium.com/devops-dudes/top-7-best-ci-cd-tools-you-should-get-your-hands-on-in-2020-832c29db936`

on multiple hosts. The profit is that new base software versions can be tested together with the self developed software in the test environment during the system test before running in production.

Some CI/CD systems are connected interrelated to the version control system that the creation of a release branch is possible via CI/CD. All functionality tests have to be passed. If system tests are integrated, these tests "can" have to be matched. But this proceeding is no requirement. The goal of this Bachelor Thesis is to integrate first system tests for IBM Z into the CI/CD pipeline of open-source projects before the release. As a result, the software will be tested for the architecture s390x continuously and IBM Z will be supported by referred projects.

4.4. Emulation for System Tests

Traditionally development environments have been running on virtual machines or locally on development systems as the CI/CD system, as an example [Scho 19, p.123]. Progressively CI/CD systems are providing automated Docker builds and support for running containers alongside virtual machines. Shell scripts and Python code are required decreasingly for system setups in separate scripts. Some systems allow the integration of shell commands into the job configuration.

System tests should integrate tests for system dependencies for multiple hardware architectures. Hosting of such systems should not be required. QEMU and other emulation software are providing features to emulate multiple system architectures. That can be integrated into the CI/CD pipeline. Host systems or test systems in the cloud have to be prepared with the listed 3 "Prerequisites". That can be automated with any configuration management system.

QEMU in combination with container technologies exhibits a method of fast sytem deployments with all required system configuration with the possibility to emulate most used system architectures. That can keep software providers (and open-source communities) reassured that the developed software is supporting all tested architectures. In conclusion, they gain confidence into their software on various platforms.

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). 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. These are using a lot of space. One 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. Docker Multi-Arch Image

As mentioned in 1.3.1 "Introduction of Docker", Docker is used as a base container engine for Kubernetes tests by the community.
Docker Inc. is maintaining a Docker image with the name "Docker-in-Docker" for multiple architectures, incl. s390x. This Docker image was introduced by the Docker contributor Jerome Petazzoni (Tianon) with the name dind[2]. Docker can be installed on Ubuntu, openSUSE, Fedora, ArchLinux and Alpine on this way. The base Docker image on Docker Hub[3] is based on Ubuntu. The base image for a specified architecture can be integrated with 2 methods. Firstly, the s390x base image can be pulled to the local container registry with **docker pull s390x/docker**. The specification **docker** behind pull alone would download the Docker image for the host architecture or would choose the platform for a multi-arch build with the platform option of BuildX in the command **docker build**. The prefix **s390x**

[1]`https://kubetest.readthedocs.io/en/latest/`
[2]`https://github.com/jpetazzo/dind`
[3]`https://hub.docker.com/_/docker`

before docker specifies a specific available architecture. The second option is the integration of **s390x/docker** into the FROM command inside of the Dockerfile. Both methods are downloading the latest Docker version with Ubuntu from `docker.io/s390x/docker:latest` and register this Docker image in the local registry. Therefore, this image is used as a base image in the self-developed Dockerfile for Kubernetes.

5.3. 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 cloning Github repositories and building binary files based on it. On this way, a lot of space is saved. Concluding, 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 (see 3.2 "Multi-Architecture Images").
Default one image is receiving the name build and the other one a name of what will be done with that. In our case, the second image has got the name work. The second image is copying with "COPY –from=build" all required built data from the first image that it can be used for running the application. On this way, it is possible to reduce the size of a docker image.
Multi-staging Dockerfiles are an approved method for executing binaries based on Github repositories.

5.4. Installation

Kubernetes requires 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 with it.
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 something as one "logical host".
Kubeadm is the administration tool to set up clusters. It is necessary to upgrade Kubernetes to other versions. 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

5.5. 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 last versioned Kubernetes tests. It is possible to receive the version number of the last 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

5.6. 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 name of the most relevant tests for the Kubernetes community is "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[4].

```
&& 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

5.6.1. End To End Testing

The e2e.test suite is a end-to-end testing framework for Kubernetes. It is testing Kubernetes for all required functionality [Ohly 19]. This framework is written in Go and is using the Ginkgo Testing framework[5] for expressive and comprehensive tests with the style of Behavior Driven Development (BDD). Expected behaviors are described in specs inside of the Kubernetes directory for e2e-tests[6]. These tests exist for every Kubernetes version in specified Github branches.
The test is built with the installed Go based on the file **e2e_test.go**. Kubernetes has been importing all multiple providers, in-tree tests, configuration support, and bindata file lookup in the file e2e_test.go. The vendoring code and their dependencies are available

[4] https://github.com/cncf/k8s-conformance
[5] https://onsi.github.io/ginkgo/
[6] https://github.com/kubernetes/kubernetes/tree/master/test/e2e

under `k8s.io/kubernetes/vendor/`. Additionally, these can be tested limited to necessary dependencies.
Conformance tests can be included with:

```
export KUBERNETES_CONFORMANCE_TEST=y
export KUBECONFIG="$HOME/.kube/config"
go run hack/e2e.go -- --provider=skeleton --test \
--test_args="--ginkgo.focus=\[Conformance\]"
```

Figure 5.4.: E2e-Test

These conformance tests are a subset of e2e-tests [Omic 18, p.8]. Every vendor is receiving a certification by Kubernetes with passing all required conformance tests. 167 of 999 tests had been such conformance tests in the year 2018 [Omic 18, p.9]. A working Kubernetes test setup is required for running tests against it.

5.7. Run in the Main Dockerfile

5.8. Integration Tests

5.9. Integration into CI/CD

The Kubernetes project has been using Prow as a CI/CD system. Prow is a Kubernetes based CI/CD system further developed based on Jenkins [Born 19]. That implifies, that the existing CI/CD pipeline can work only with containers. Before the integration into the community pipeline, it is possible to use a default Jenkins system because of identical configuration files. An additional advantage in this case is the chance to test the integration of Kubernetes into QEMU before providing the IBM Z environment container based for test purposes.

5.9.1. Jenkins versus Prow

As mentioned in 4.3 "CI/CD Systems", **Jenkins**[7] is the most popular open-source CI/CD system. This open-source project has started with a default CI/CD workflow based on real hardware and later on virtual machines to test software before the release. Jenkins

[7] https://www.jenkins.io/

provides many plugins[8] for git integration about release management support until container deployments.
There exists a special **Kubernetes plugin**[9], that Jenkins can understand yaml configuration files for Kubernetes deployments which are used for integration tests by the Kubernetes community as an example. On this way, Jenkins can execute the same tests as Prow together with emulations because the Kubernetes environment is deployed inside of a emulated system. Jenkins can handle emulated systems the same as virtual machines. The Kubernetes plugin makes automated Kubernetes deployments possible. Therefore, a Jenkins installation is the best choice for the transition from emulated systems together with container technologies to a working test strategy with container based test environments in Prow.

Jenkins X[10] has been developed based on Jenkins specialized on Kubernetes and cloud native environments. That means, that something as the Kubernetes plugin is integrated and the system can work with public cloud providers and other cloud technologies for test and production deployment pipelines. All in all, Jenkins X is an improved version of Jenkins for cloud environments. The system can work with container registries, understands yaml configuration files without additional plugins and is able to deploy scalable systems based on containers in the local data centre as well as in the public cloud (AWS, Google Cloud, IBM cloud, Azure).

Prow[11] is a Kubernetes-native solution based on Jenkins X. This CI/CD system is used by projects as Kubernetes, Istio und OpenShift which are all based on Kubernetes. This system is something as a Jenkins X expanded with special Kubernetes APIs and cloud specific features. It can understand and deploy only Kubernetes based environments. Other configurations are not supported. The benefit in comparison to Jenkins is, that yaml configurations for Kubernetes setups are transferable between Jenkins and Prow.

5.9.2. System Requirements

Jenkins can include all required installation of integration tests and the whole system setup into the specified Docker image. Therefore, the disk image size can be defined based on the Docker image and the specified command in 2.5.3 "Mounting an External File system" with

```
docker images | grep 'Kubernetes-Test' | awk '{print int($7+0.5)"G"}'
```

The Kubernetes community is referencing[12] a minimum about 2GB of memory and 2 CPUs per system. It has to be observerd, that Kubernetes is watching master and worker node as a

[8] https://plugins.jenkins.io/
[9] https://plugins.jenkins.io/kubernetes/
[10] https://jenkins-x.io/
[11] https://jenkins-x.io/docs/reference/components/prow/
[12] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#before-you-begin

own system. Both are on one system in our case. Additionally, some RAM and CPU should be added for running tests. Therefore, 6GB for memory and 5 CPUs will be calculated for the QEMU setup.

Chapter 6.

Cassandra

6.1. Overview

Apache Cassandra is a Java based NoSQL database management system. It contains the commandline interface CQL for Cassandra database commands instead of SQL. It has to be recognized that the supported Java version by the open-source project Apache Cassandra is able to be changed with every version. That has to be considered during the integration into the CI/CD pipeline of the project. There are different multi-arch Dockerfiles for all versions of AdoptOpenJDK for public usage. These can be applied for the base Cassandra image. Afterwards, a multi-arch Docker image for Cassandra, which includes the s390x architecture, can be published. A 6.5 "nagios monitoring check" - usable for containerized environments - is chosen for tests. This one has to be assimilated in the setup.
This Apache project is using CircleCI for Continuous Integration.

6.2. Installation

A Dockerfile[1] for a multi-arch image should be created in this project with the reason for using it for multiple architectures in the future. Apache Cassandra is a Java based application. Therefore, the research has contained the search for a multi-arch image with pre-installed Java. The Cassandra version 3.11.6 is supporting Java 8. AdoptOpenJDK is maintaining multi-arch images for different Linux distributions with pre-installed AdoptOpenJDK. In the description of 2.2 "Multi-Arch Image AdoptOpenJDK 8" is explained how the recommended Java version will be chosen for the specific architecture in this base image. This base image is integrated with `FROM adoptopenjdk:8-jdk-hotspot` on top of the Dockerfile.

Apache Cassandra will be cloned from the Github repository with
`git clone https://github.com/apache/cassandra.git`.
The special version is set with an agument with the ARG variable. On this way, it is

[1]https://github.com/s390x-container-samples/s390x-cassandra-ci-cd/blob/master/Dockerfile

possible to checkout other versions with a given argument as `--build-arg CASSANDRA_VERSION=3.12.4` in the docker build command.
Apache Cassandra is using JNA (Java Native Library)[2] as an extension of Java. JNA provides Java programs easy access to native shared libraries for an improved communication with the database system. The ant command is building the application and the JNA subsequently. The command `sed -i ' s/Xss256k/Xss32m/'` is setting the size of the frame stack (JVM stack) used by each thread to store local variables. This frame size is reduced to a minimum and stored in both files build.xml and conf/jvm.options. A soft link (**ln -s**) will be created to save space in the container. Such a soft link refers to a symbolic path of another file or directory. Following, all files inside of the build directory `/root/cassandra` are additionally available under `/usr/share/cassandra`. Another benefit is the low time for the creation of a soft link in comparison to a copy / move command.

```
FROM adoptopenjdk:8-jdk-hotspot
ARG CASSANDRA_VERSION=3.11.6
git clone https://github.com/java-native-access/jna.git \
&& cd jna \
&& git checkout 4.2.2 \
&& ant native jar \
# Build and install Apache Cassandra
&& cd $SOURCE_ROOT \
&& git clone https://github.com/apache/cassandra.git \
&& cd cassandra \
&& git checkout cassandra-${CASSANDRA_VERSION} \
&& sed -i ' s/Xss256k/Xss32m/' build.xml conf/jvm.options \
&& ant \
&& rm lib/snappy-java-1.1.1.7.jar \
&& wget -O lib/snappy-java-1.1.2.6.jar https://repo1.maven.org/maven2/org/xerial/snappy/
snappy-java/1.1.2.6/snappy-java-1.1.2.6.jar \
&& rm lib/jna-4.2.2.jar \
&& ln -s $SOURCE_ROOT/cassandra /usr/share/cassandra \
&& rm -rf  $SOURCE_ROOT/jna $SOURCE_ROOT/*.tar.gz  \
&& rm -rf /usr/share/cassandra/test \
```

Figure 6.1.: Cassandra Installation

Cassandra can be started with `CMD ["cassandra", "-R", "-f"]` with **f** for force and **R** so start the service as root after setting the environment variables with inside of the Docker container.

[2] https://github.com/java-native-access/jna

6.3. Java Optimization

Apache Cassandra should be allowed to use all available memory of the minimal system effectively. Default the JVM is configured that only a part of that can be used. These configurations are set in the Java Options. Afterwards, Apache Cassandra can work with a good performance.
Every major Java release can differ from another one with such options for Java optimization.

Following Java optimizations are available for Java 8:

- **-XX:+UseCGroupMemoryLimitForHeap**
  This option tells the JVM to look into `/sys/fs/cgroup/memory/memory.limit` after the available memory and to use the whole memory if necessary [Floo 17].

- **-XX:ParallelGCThreads=2**
  Apache Cassandra is referencing 2 required CPUs as a minimum in their documentation [Cass]. This JVM option permits 2 GC (Garbage Collection) threads parallel on different (virtual) processors.

All these Java optimization options can be set as an environment variable with `ENV JVM_OPTIONS` in the Dockerfile. Then they will be used for every Java process like Apache Cassandra as an example. As a result, Apache Cassandra can use the whole memory and delivered CPU, which is specified in the QEMU options.

6.4. Workaround Because of a JVM Issue

Apache Cassandra is using OpenJDK 8 for running as default. There are some discussions[3] about supporting OpenJDK 11 at the moment. The Dockerfile mentioned above is using AdoptOpenJDK 8 at the build process is working without any problems on Z systems. There exists a JVM issue during the build process on x86 which is the reason for non successful builds of Java applications there. That counts for all more complex Java applications.

This case has been tested with AdoptOpenJDK 8 and AdoptOpenJDK 11. The library hsdis-s390.so is not loadable for both Java versions for emulated s390x guests on x86, but during the build process on s390x[4]. This issue does not exist with the version AdoptOpenJDK 14, but Apache Cassandra does not work with this version.

[3] `https://lists.apache.org/thread.html/r38f6beaa22e247cb38212f476f5f79efdc6587f83af0397406c06d7c%40<dev.cassandra.apache.org>`
[4] `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=951764`

This emulation bug of qemu has to be fixed and the issue has been reported to KVM Developers inside of IBM.

There exists a workaround for the development process for finishing the project. The Docker image is buildable on s390x. Therefore, the Docker image has been built for s390x on s390x with
`docker build --squash -t cassandra .`.
Afterwards, this image can be saved as a tar archive with
`docker save cassandra > cassandra.tar`
on the host. It has to be transferred with rsync to the x86 system then. The image inside of the tar archive can be included to all existing images of `docker images` with the command
`docker load --input cassandra.tar`.
After this action, the Docker image is usable as it would have been built on this system. It can be integrated into QEMU and is able to be started without any problems afterwards.

6.5. Nagios Monitoring Check for Tests

Nagios is an open-source monitoring system to check the status of systems with their services. There are many checks for different services available. One monitoring check[5] exists for Apache Cassandra. It is testing whether Java would be working, the service Apache Cassandra would be up and running, and whether CQL commands would be possible to execute without any issues. This Perl script is using **nodetool** for monitoring the JVM and the application. Nodetool is delivered together with Cassandra. It is connecting with the database and can dispense statistics about the Cassandra cluster / host [Carp 20, p.256]. The result contains the status, information about memory usage and other capacities. This tool can indicate issues resulting in messages via the monitoring plugin in our case.
This monitoring check is written in Perl and is a nice choice for automated tests with error messages in the case if any output would not give an **OK**.
All monitoring plugins in the referenced Github repository are developed additionally for integration into Docker. Following, these scripts do not require any installed nagios system for monitoring. Summarily, these monitoring plugins can be used for tests inside of containerized environments for testing Docker images.

[5] `https://github.com/skyscrapers/monitoring-plugins/blob/master/check_cassandra.pl`

For a better maintenance, **cassandra_check.pl** is integrated with a separate Dockerfile with the installation Dockerfile as a base image:

```
FROM cassandra-s390
RUN apt-get update && apt-get install -y \
    perl \
    perl-base \
    libimage-magick-perl
WORKDIR /bin
COPY cassandra_check.pl /bin/cassandra_check.pl
ENTRYPOINT ["perl", "cassandra_check.pl"]
RUN apt-get remove -y \
    perl \
    libimage-magick-perl\
    perl-base  \
&& apt autoremove -y \
&& apt-get clean && rm -rf /var/lib/apt/lists/*
```

Figure 6.2.: Cassandra Monitoring Check

The separation of both Dockerfiles profits by the reutilization of the main Dockerfile. The base Docker image can be published in different container registries for public usage without the monitoring check. Accessorily, the build process for the base image of Cassandra and the test with the monitoring check can be splitted in the CI/CD pipeline. The possibility of bug identification is simplified by this process.
The test Dockerfile in the monitoring directory[6] can be reused for every Cassandra container image based on Ubuntu or Debian. The unique requirement is that the FROM line is matching the name of the tested Cassandra image name.

6.6. Run Docker Image in QEMU

The Docker image will be built with the command
`docker build buildx --platform=linux/s390x --squash -t cassandra-s390 .`
in the directory with the Cassandra Dockerfile for the architecture s390x. That is the base Cassandra image.
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 during the emulation preparation. The command `docker images` has to show the registered Docker image with

[6] https://github.com/s390x-container-samples/s390x-cassandra-ci-cd/blob/master/monitoring/Dockerfile

the name cassandra-390 in the local container registry. Afterwards, the monitoring check has to be added. The same procedure will be executed in the monitoring directory. The FROM command is referencing the self built Cassandra image with cassandra-s390. With the following command the monitoring check can be attached to the exiting Cassandra image to test it:

`docker build buildx --platform=linux/s390x --squash -t cassandra-check .`

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-s390x.img 2G`

can be used. This image needs to be converted to any Linux file system because QEMU does not know the Docker file system. Therefore, the image is formated with the command `mkfs.ext4 -F cassandra-s390x.img`. A directory with the name rootfs has to be created and the command for receiving the file system of the Docker image.

`docker export $(docker create cassandra-check)| tar -C "rootfs"-xvf -`

is exporting the docker image into the directory rootfs. Afterwards, the following command transfers the content of rootfs into the image cassandra.img.

```
mkdir /mnt/rootfs
mount -o loop cassandra-s390x.img /mnt/rootfs
cp -r rootfs/* /mnt/rootfs/.
```

Figure 6.3.: Mount Rootfs

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'
```

Figure 6.4.: Run Cassandra

6.7. Integration into CI/CD

The community of Apache Cassandra is using CircleCI[7] as a Continious Integration system. This CI/CD system is connected with Github. The tests are running after every commit in the master repository. These tests can run in a container or in a virtual machine. These options make it possible to build the Docker image for the architecture s390x and to integrate it into QEMU via the CI/CD pipeline without additional scripting or workarounds.
CircleCI is working with a yaml configuration[8]. An enhancement is the opportunity to define shell commands with "command" in the yml file. Therefore, 3.3 "Fetching Signed Linux Kernel Image" can be integrated. Additionally, dependencies can be created. On this way, it can be assured that the build process for the Docker image is triggered before the file system creation. The same counts for the kernel fetching process and the file system creation before the QEMU start.
This configuration file with the whole test process has got the name **config.yml**.
Workflows defines the order of different process steps with their dependencies with **- requires**. The different processes are specified separately above then. The advantage of different steps with dependencies is, that every step can be tested an the build process will be aborted after one failed process and this step will be listed with the error message in the CI/CD system.
The configuration of CircleCI is clearly structured. The workspace can be declared. Running steps can receive names. Shell commands can be executed. Additionally, binaries and other files can be stored as artifacts.
The implementation of the job configuration can be found under A.1 "CircleCI configuration for Apache Cassandra". It can only work with a bug fix for the 6.4 "JVM Issue".
The system is configured that the version number of Apache Cassandra can be replaced by a new version with `--build-arg CASSANDRA_VERSION=$CASSANDRA_VERSION` in the docker build command. With that, the checkout of the special Cassandra branch will be executed instead of the configured last one.

[7] `https://circleci.com/`
[8] `https://circleci.com/docs/2.0/sample-config/`

Chapter 7.

Outlook

This project has been combinining container technologies with an emulator which is used as a foundation for different virtualization technologies. Additionally, it has represented a method to deploy applications fast with an emulated system for alternative system architectures. Emulation bugs fur QEMU have to be fixed. Afterwards, this method can be used in all CI/CD pipelines of open-source projects and on computers by Developers. New features can be tested without high effort. Following, the software development process can become faster. There is the possibility to embed the QEMU deployment of Docker images into virtualization technologies as KVM and XEN. But that requires additional development for alignments.
All in all, this emulation method is usable for fast deployments in CI/CD pipelines and system deployments for all system architectures.

Chapter 8.

Summary

Hier kommt eine Zusammenfassung

Appendix A.

Supplemental Information

A.1. CircleCI configuration for Apache Cassandra

```
jobs:
  build-docker-image:
    working_directory: ~/s390x
    docker:
      - image: Dockerfile
    steps:
      - attach_workspace:
          at: .

      - run:
            name: Build for s390x
            command: docker build buildx --platform=linux/s390x --squash \
                     --build-arg CASSANDRA_VERSION=$CASSANDRA_VERSION \
                     -t cassandra:s390x .
      - docker/check:
         registry: $DOCKER_REGISTRY
  prepare-kernel:
    steps:
        - run:
              name: Fetching the Linux kernel
              command: |
                      mkdir ~/s390x/kernel-bionic; debootstrap --include=linux-generic \
                      --arch=s390x --variant=minbase bionic kernel-bionic \
                      http://ports.ubuntu.com
        - store_artifacts:
              path: ~/s390x/kernel-bionic/boot/vmlinuz-4.15.0-20-generic
        - persist_to_workspace:
                  root: .
                  paths:
                      - .
  prepare-file system:
    working_directory: ~/s390x
      steps:
        - attach_workspace:
                at: .
        - run:
              name: Creating a ext4 file system with integration of the Cassandra image
              command: |
                      mkdir rootfs; qemu-img create -f raw cassandra-s390x.img \
                      $(docker images | grep 'cassandra:s390x' | \
                      awk '{print int($7+0.5)"G"}'); \
                      docker export $(docker create cassandra:s390x)| \
                      tar -C "rootfs"-xvf -;  mkfs.ext4 -F cassandra-s390x.img;\
                       mount -o loop cassandra-s390x.img /mnt/rootfs; \
                      cp -r rootfs/* /mnt/rootfs/.
        - store_artifacts:
              path: ~/s390x/cassandra-s390x.img
  test:
  working_directory: ~/s390x
    steps:
      - attach_workspace:
                at: .
      - run:
             name: Run test with QEMU
             command: |
                     /usr/bin/qemu-system-s390x -kernel vmlinuz-4.15.0-20-generic \
                     -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 cassandra-s390x.img \
                     --append 'root=/dev/vda rw console=ttyS0 rdinit=/bin/bash'
workflows:
    version: 1
    build-test:
      jobs:
        - prepare-kernel
        - build-docker-image
              requires:
                  - prepare-kernel
        - prepare-file system
              requires:
                  - build-docker-image
        - test
             requires:
                 - prepare-kernel
                 - prepare-file system
```

List of Figures

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].

[Born 19] D. Bornkessel and A. Kammer. “Kubernetes und seine CI/CD-Generationen”. November 2019. https://jaxenter.de/kubernetes/kubernetes-cicd-generationen-88174 [Accessed 24 September 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.

[Carp 20] J. Carpenter and E. Hewitt. *Cassandra: The Definitive Guide*. "O’Reilly Media, Inc.", 2020.

[Cass] A. Cassandra. “Apache Cassandra Documentation: Hardware Choices”. https://cassandra.apache.org/doc/latest/operating/hardware.html [Accessed 10 September 2020].

[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.

[Data] Datastax. “What is Cassandra?”. https://www.datastax.com/cassandra [Accessed 16 July 2020].

[Floo 17] C. Flood. “OpenJDK and Containers”. April 2017. https://developers.redhat.com/blog/2017/04/04/openjdk-and-containers/#more-433899 [Accessed 10 September 2020].

[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].

[IBMa] IBM. “Transaction processing”. https://www.ibm.com/it-infrastructure/z/capabilities/transaction-processing [Accessed 10 September 2020].

[IBMb] IBM. “What is a mainframe?”. https://www.ibm.com/it-infrastructure/z/education/what-is-a-mainframe [Accessed 10 September 2020].

[Lasc 20] O. Lascu, B. White, J. Troy, and F. Packheiser. *IBM z15 Technical Instruction*. IBM Redbooks, 2020.

[Linu] O. M. P. a Linux Foundation Project. “The Modern Mainframe”. https://www.openmainframeproject.org/modern-mainframe [Accessed 04 September 2020].

[Ohly 19] P. Ohly. “Kubernetes End-to-end Testing for Everyone”. March 2019. https://kubernetes.io/blog/2019/03/22/kubernetes-end-to-end-testing-for-everyone/ [Accessed 16 September 2020].

[Omic 18] K. Omichi, Ed. *Verify Your Kubernetes Clusters with Upstream e2e Tests*, Linux Foundation, Open Source Summit NA 2018, August 2018. https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Verify-Your-Kubernetes-Clusters-with-Upstream-e2e-Tests-Kenichi-Omichi-NEC.pdf.

[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.

[Ostl 20] U. Ostler and J. Höfling. “Was ist x86?: Ein Methusalem unter den Prozessor-Befehlssätzen”. July 2020. https://www.datacenter-insider.de/was-ist-x86-a-946188/ [Accessed 19 September 2020].

[QEMU] QEMU. “QEMU User space emulator”. https://www.qemu.org/docs/master/user/main.html [Accessed 10 September 2020].

[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].

[Spil 19] A. Spillner and T. Linz. *Basiswissen Softwaretest.* dpunkt.verlag, 2019.

[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.

[Tsch 09] A. Tschoeke, Ed. *Linux on IBM System z10*, IBM Deutschland Research & Development GmbH, TU Kaiserslautern, 2009. http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/users/kschmidt/mainframe/documentation2009/Linux_on_IBM_System_z.pdf [Accessed 21 August 2020].

[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.

[Yang 19] R. Yang. "Docker Multiarch Builds Quick and Easy". November 2019. https://renlord.com/posts/2019-11-09-docker-multiarch/ [Accessed 21 August 2020].

Glossary

application layer The layer in the software stack wich is responsible for running applications. 1, v, 1

bash Unix shell and command language integrated in all Linux operating systems. 1

binary Executable application built in a single file. 1

CI/CD Continuous Integration/Continuous Delivery is a method for automated tests in the software development. 1, v, 1

configuration Saved settings for an application or computer program. 1

container A standard unit with the minimum of mandatory software. 1

container engine The foundation for managing and organizing containers. 1

containerized Splitting programs into different services and deploying it connected in many containers as a single application. 1, v

CPU The Central Processing Unit is a main component of computers for executing instructions. 1

cross-compilation Method for compiling code for a different computer system or architecture. 1

deployment Fast automated setup of a system with applications and configurations. 1

Docker daemon Manages and organizes objects inside of the container engine Docker. 1

Docker image A built system listed under "docker images" which is runnable. 1

Dockerfile Base file with all installations and configurations for a container image. 1

emulator Software for emulating different architectures of other systems for running software on it. 1, v

Github Software version control system by Microsoft for free usage. 1

Go Programming language much used for container software. 1

hard disk The whole operating system with data is written and saved on this component of the computer system. 1

high availability cluster Group of systems which is representing one system with the same software for a minimum amount of down-time. 1

hypervisor Software that deploys and run multiple guest virtual machines on real hardware. 1

IBM Z systems Mainframe hardware by IBM. 1, v, 1

issue A bug report because of a failure in the software. 1

Java Programming language much used for Enterprise software development. 1

JSON The JavaScript Object Notation is an open standard file format for saving data structured. 1

JVM Java Virtual Machine enables running Java applications compiled to Java bytecode. 1

Kubernetes stack Kubernetes splitted into different layers from the container environment until the application layer. 1

library A suite of reusable code inside of a programming language for software development. 1

Linux Free operating system developed by different communities and available as different distributions. 1

Linux kernel Software by the Linux community containing all important drivers for running the operating system. 1

LPAR Logical partition to separate a mainframe for different running operating systems. 1

mainframe Large computer which is able to execute millions of transactions in parallel. 1

memory The place inside of a computer for saving data before writing on the hard disk. 1

mounted Integrated into a system with the mount command. 1

multi-architecture images Images based on system builds usable for multiple system architectures. 1

package Software offered comprimized and with dependencies to other software by different Linux distributions. 1

pod An association of multiple containers with services for one application. 1

registry Built container images are registered and maintained there for general usage. 1

repository The location in the internet where software packages are downloadable for installations provided by Linux distributions and other providers. 1

root permissions Administrator access permissions with all privileges for a computer system based on Linux/Unix. 1

s390x Mainframe system architecture. 1, v, 1

scaling Distributing processes to different cores of a system and executing there. 1

shell Terminal of a Linux/Unix system for entering commands. 1

Ubuntu Linux distribution maintained by Canonical. 1

virtual machine A virtual machine is a system running in a hypervisor as a separate system on another system. 1

x86 Architecture of a default PC. 1

z/OS Operating system by IBM for Z systems. 1