Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Containerizing a user-space storage framework for reproducibility

vorgelegt von

Marcel Papenfuss

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	7065349
Erstgutachter:	Jun.-Prof. Dr. Michael Kuhn
Zweitgutachter:	Kira Duwe
Betreuer:	Jun.-Prof. Dr. Michael Kuhn und Kira Duwe

Hamburg, 2021-03-11

Abstract

Container solutions are becoming increasingly popular in the High Performance Computing (HPC) field. They support points such as reproducibility and mobility in the form that complete system environments can be reproduced and executed on other systems. In this thesis, JULEA, a flexible user-space storage system, is used to show how such a container solution can look like for such an application. Different container engines like Docker, Podman and Singularity are introduced and a concrete implementation approach for each is shown.
Besides concrete solutions, the planning and design of workflow plays an important role. With the help of GitHub Actions, a completely automated workflow is designed, which is responsible for the image creation.
In addition, an evaluation of the implementation problems will be presented and a comparison between existing and new container solutions will be shown.

Contents

1. Introduction

In this chapter, a short introduction to the thesis is given. It is highlighted why the thesis is done and which goals are expected from it.

1.1. Motivation

In recent years, containers have become increasingly important and very popular. Especially in enterprise solutions, containers have become an essential part of development and operation. Also in High-Performance Computing (HPC) environments and generally in the scientific field, such solutions are more in demand [GWR20].

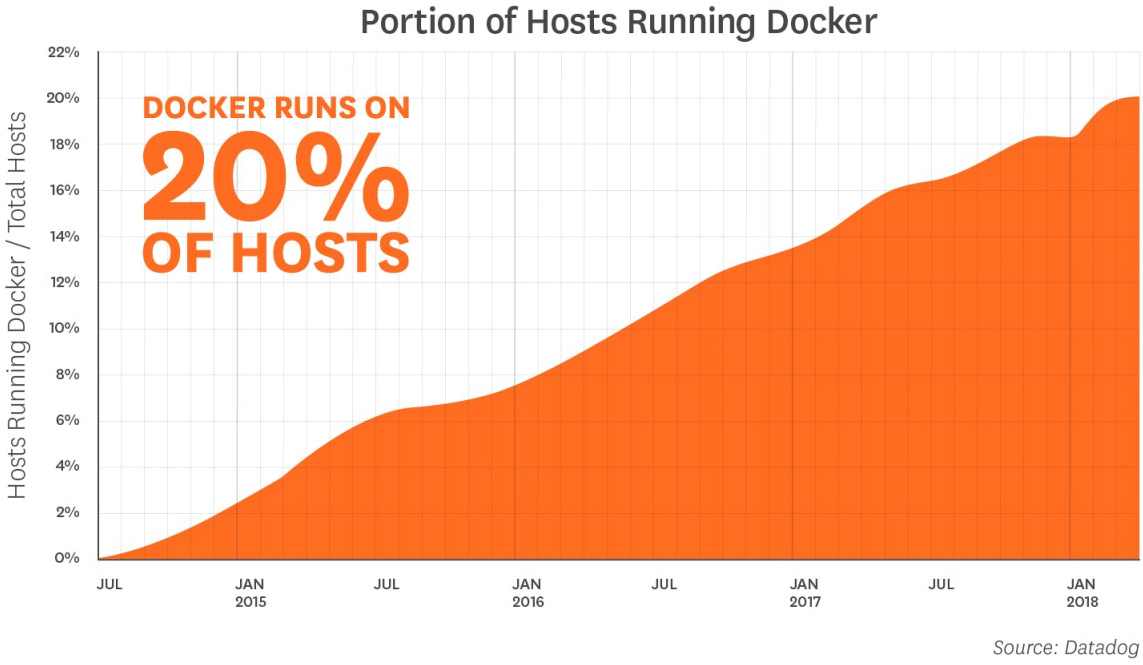

Figure 1.1.: Docker over the last years [Dat18]

Experiments and their solutions are often published in papers in the scientific field. In many fields, these results must be reproducible or even formally proven. This standard has not been met for a long time in computer-based experiments. In many publications, the results are only briefly described and the code used for them is rarely available, which can raise questions about reliability. An important component for the reproducibility

of experiments is therefore an accumulation of all data, source code, parameters and environment [FS12].
Intel describes how important the specification of the used software and operating system can be in a contribution for its software tools in connection with floating point reproducibility. Thus, calculations under different conditions can lead to different results, which can cause problems in certain areas. Currently, there is no possibility to perform reproducibility tests on different operating systems like Windows and Linux [MCP21].
Another problem that often occurs during application development is the use of dependencies. Applications often have many external libraries that are required for them to be compiled or used. If the application is to be used on another system, all dependencies must be present. However, the user often does not know which necessary dependencies one has to install for this. If the user simply installs any version of a library, this can be decisive for the result of the experiment. Maybe a function in the library was changed in the background, which leads to a different result of the experiment.
Furthermore, containers are being used more and more in cloud computing. The user can run the applications in an isolated area and is provided with certain resources for this. In addition to the efficient use of the operating system, a container in the cloud can be used for version control or contribute to more productive use, as it is easier to work on different applications, which can be achieved on the one hand through a fast boot time [Var20].

1.2. Thesis Goals

The goal of this thesis is to containerize a user-space storage framework to address the issues described in Section 1.1. JULEA is used as an example for such a framework.
Due to the large number of different container engines, it is necessary to find the best possible solution and implement it. Thereby the specificity of the different systems has to be taken into account.
In addition, it should be possible in the future that a container image is automatically build, so that a current version is always available for use.

2. Background

This chapter gives a brief overview on the storage framework JULEA. A special focus is placed on container based virtualization which is the main aspect of the thesis.

2.1. Prior knowledge

In Section 1.1 it was written that the reproducibility of results plays an important role. For these reasons, more and more publishers of papers offer to submit so-called artifacts. An artifact is described by ACM as follows:

> *By "artifact" we mean a digital object that was either created by the authors to be used as part of the study or generated by the experiment itself. For example, artifacts can be software systems, scripts used to run experiments, input datasets, raw data collected in the experiment, or scripts used to analyze results [Inc21].*

Many journals now offer a formal process that makes the provision of artifacts a standard practice. If the artifacts are made publicly available to everyone, interested parties can check the result and thus improve the robustness. Furthermore, the reusability can be improved and other works can build their experiment on the artifacts [Inc20a]. In computational experiments, reproducibility means that an independent person or group can achieve the same result with the artifacts provided. It is important that the same measuring methods and the same operating conditions are used. It must be irrelevant whether the experiment is executed once or several times, the result must always be the same [Inc20a].

2.2. JULEA

JULEA [Kuh17] is a flexible storage framework that makes it possible to offer different client interfaces to applications. To quickly implement new approaches, e.g. in research or teaching, JULEA offers object and key-value backends, which can be located either on the client or on the server side. Common storage technologies such as POSIX, LevelDB and MongoDB are currently supported as backends. The backend is loaded at runtime and thus enables to be flexibly exchanged and adapted.
To make it easier to use in teaching, among other things, JULEA was written entirely in the user space. This leads to a simplified development and debugging process and no need of knowledge of kernel space development.

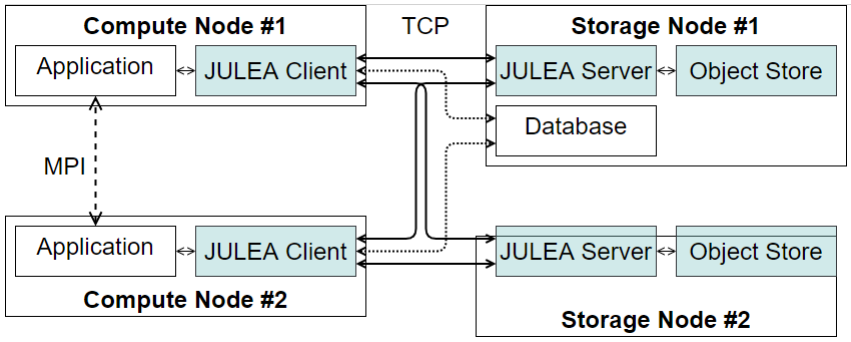

Figure 2.1.: JULEA-Client-Server architecture [KD21]

JULEA was built according to the client-server design. Figure 2.1 shows an exemplary setup. On the left side there are two compute nodes, each of which has started a JULEA-Client. The fact that the two applications communicate here via the Message Passing Interface (MPI) will not be important for the project. On the right side, there are two storage nodes, each of which has started a JULEA-Server. Both use an object store as backend in this example. The connection between a client and server is done using Transmission Control Protocol (TCP). A JULEA-Client can also communicate directly with infrastructures that already have the necessary requirements to communicate with clients, such as MongoDB or SQL. The JULEA-Server as a "middleman" is then not necessary.

2.3. Container virtualization

The two best-known virtualization technologies are hypervisor and container-based virtualization which are visualized in Figure 2.2.
Hypervisor virtualization requires a hypervisor that provides the resources for the virtual machine (VM). Each VM has its own operating system (OS), which means that there is a complete abstraction between the VM and host system. It is also possible that several different operating systems can be run on one host system.
In comparison, the container-based virtualization, also known as operating system level virtualization, has a container engine instead of the hypervisor. The biggest difference, however, is that with this technology, the host's OS is used and there is no longer a separate OS in the individual containers, which leads to a weaker isolation compared to the hypervisor virtualization. The container engine divides the physical resources and creates an isolated user space for each container [CMF+16]. Because the container engine uses the Linux kernel of the host system, container virtualization is lightweight and higher performing than hypervisor virtualization [Boe15].

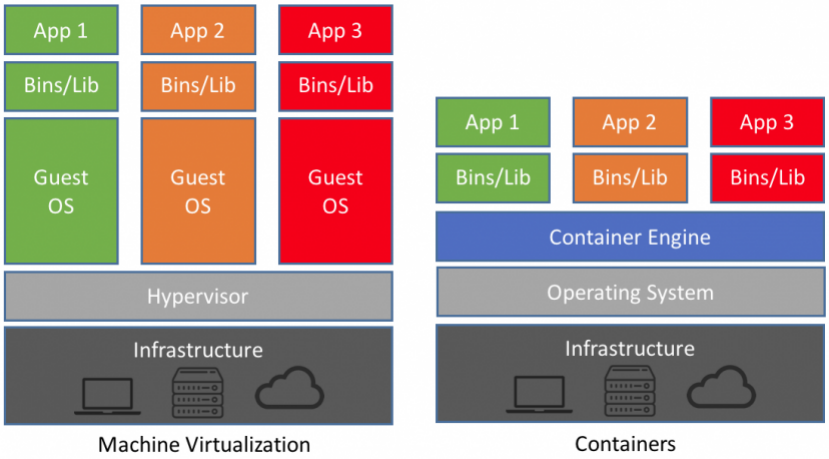

Figure 2.2.: Virutal Machine compared to Container structure [Cha18]

2.3.1. Docker

Docker is an open source project that provides container virtualization. It builds on various techniques that have long been known, such as Linux containers (LXC). One goal of Docker is to simplify the development and delivery of applications with the help of containers [Ber14].

The kernel-level namespaces provided by LXC and thus used by Docker separate the containers from the host. The user namespaces ensures that the root user of the container has no corresponding rights on the host system. Process and networking namespaces ensure, on the one hand, that the container only sees processes on the container, and on the other hand, its own network is created. Control groups (cgroups) are used to limit and manage resources. This means that memory space and I/O can be limited [Mer14].

Dockerfile

Docker helps to solve the problem with used dependencies and environments. To keep track of which software has been installed and how, e.g. with which parameters or on which paths, Docker provided so-called Dockerfiles. This is a script with simple syntax and a set of instructions that define how an image can be built [Boe15].
Listing 2.1 shows an example Dockerfile, which was taken from the following source [Inc20c].

```
1 FROM ubuntu:18.04
2 COPY . /app
3 RUN make /app
4 CMD python /app/app.py
```

Listing 2.1: Example Dockerfile

This shows the basic structure and syntax. First a new layer is created, which is built on the Docker image of Ubuntu 18.04. Then files from the current directory are copied into the image and built using *RUN*. Finally, it defines which command should be executed inside the container.

Docker images

The image is created from the instructions given in the Dockerfile and is a type of snapshot of the current environment. It consists of several read-only layers, which are built on top of each other. All layers together contain all necessary files to start a Docker Container [ZRT+18]. In the end, a container is just a label for the fact that an instance is running from an image [Boe15].

Docker registry

Similar to a code repository, Docker images are stored in the Docker registry. Images can be pushed or pulled from this registry. A registry can be either public or private. The best known example of a public registry is Docker Hub [RBA17].

Docker container

The first step when creating a container is to check if the required image is available on the local host. If it is available, nothing has to happen. However, if it does not exist, the image is downloaded from the registry and made available locally.
The second step is the actual creation of the container. For this all read-only layers of the image are connected to one layer. Then a writable layer is created on the top of it. This layer is also called the container layer. All changes within the container are written to this layer. To change files, they have to be loaded from the read-only layer into the writeable layer [ZRT+18].
If a container is deleted, the image is not changed. The container layer, however, is deleted as well. By this mechanism it is possible that several containers use the same image [Inc20b].

Docker Compose

In larger system landscapes, it is quite possible that several services have to run at once. Usually, for each service a separate Docker container is created. This makes administration, e.g. creating each service and starting it, very complex and laborious. Docker Compose was developed to react to this.
With Compose all services can be configured by a YAML file, typically named *docker-compose.yml*. Within the definition, docker files are used so that the environment can be reproduced everywhere. Then all services can be created, started or shut down at same time with a simple command. [Inc20f]

```
services:
    server:
        build:
            context: ./server
        ports:
            - "1234:1234"
    client:
        build:
            context: ./client
        volumes:
            - "./code:/build"
```

Listing 2.2: Example Docker Compose YAML file

Listing 2.2 shows an example for a Docker Compose file. Two services are created. The *server* service uses an image that is built from a Dockerfile in die */server* directory. Moreover, it binds the container to the exposed port *1234*. In the *client* service it is shown how an volume can be mounted from the host machine (*/code*) to the container (*/build*).

2.3.2. Podman

Although Docker has a great popularity, it can not meet all requirements, especially in the HPCs field. This is the reason why the Podman project was created. An example is the client-server model, which is a potential security risk [GWR20].
Podman is daemonless and open source. With its own CLI, which is very similar to Docker's, it is easy to find, build and run containers. Because Podman, like Docker, is based on the Open Containers Initiative (OCI) Runtime Standard, Podman's containers are hardly distinguishable from those of others [wha20].
One of the biggest features of Podman is that containers can be run with either root or non-root privileges. This is achieved with the help of namespaces. Also, container images can be created from Dockerfiles without starting a process or having root privileges [GWR20].
Podman offers support for Checkpoint/Restore in Userspace (CRIU), which might be interesting for HPC. This makes it possible to restore a container at a later time. Currently it is not possible to use this feature with rootless containers or with MPI and InfiniBand [GWR20].

2.3.3. Singularity

Singularity is another container platform that was developed for the needs of science. Singularity pays special attention to the mobility of workflows and software. This is achieved by a single portable image file. The file contains all files of the container [KSB17].
A further development criterion is reproducibility. After a workflow has been defined, the image can be archived or snapshot so that the user knows that the files have not

changed within the container. Hashing provides a method to ensure that the image has not been changed or edited [KSB17].
With singularity rights system there is no user escalation, which means that a container gets the rights that the user has when the container is started [ADS17].
The container images are read-only by default. Only the home directory and other automatic remounted folders are writable. Applications can access the data inside the container and on the host system [K+15].

2.4. GitHub Actions

With the help of GitHub Actions automated tasks can be created for a repository. Different actions can be triggered depending on the type of event.
An event, e.g. a push of a commit to a repository, automatically triggers a workflow consisting of one job. This job is then again divided into steps. The individual steps are defined by the respective actions. Each action can have a different command. There are already a number of actions available from the GitHub community that can be used or you can create your own [Inc20h].

```
1  name: test-action
2  on: [push]
3  jobs:
4    run-test-script:
5      runs-on: ubuntu-latest
6      steps:
7        - name: Checkout
8          uses: actions/checkout@v2
9        - name: Tests
10         run: ./scripts/test.sh
```

Listing 2.3: Example GitHub Action

Listing 2.3 shows an example workflow. On each push-event (line 2) the *test-action* is executed. The job in the example is grouped by the name *run-test-script* (line 4). Line 5 indicates that the job is running on a fresh virtual machine running an Ubuntu. The job is further divided in line 6 by the steps. In the example there are two steps. One is the Checkout Step (lines 7+8), which uses the community action *actions/checkout@v2*, which checks out the current repository and makes it available to the virtual machine. This action must be used every time a workflow is used for the code in the repository. The other step (lines 9+10) uses *run* to execute a command in the VM. Here a script is called that contains tests [Inc20h].

3. Design

This chapter describes the design options available and how they will be implemented. At first, the different user requirements are discussed. Secondly, the various container engines are examined in more detail and an individual solution is described for each.

3.1. Current JULEA installation steps

Before the individual steps for the container engines can be explained, it is briefly explained which steps are necessary so far to install and configure JULEA [JUL20].

Firstly, the project must be cloned from GitHub. Then all necessary dependencies have to be downloaded and installed. This is done by a dependencies script Listing 3.1.

```
1 ./scripts/install-dependencies.sh
```

Listing 3.1: Install dependencies for JULEA

Afterwards, the JULEA environment can be loaded, which is necessary so that all JULEA binaries and libraries can be found Listing 3.2.

```
1 . scripts/environment.sh
```

Listing 3.2: Load JULEA environment variables

The configuration is done with Meson[1] and the compilation with Ninja[2] as seen in Listing 3.3.

```
1 meson setup --prefix="${HOME}/julea-install"
    ↪ -Db_sanitize=address,undefined bld
2 ninja -C bld
```

Listing 3.3: Configure and compile JULEA using Meson and Ninja

Finally, the actual JULEA configuration can be created in Listing 3.4.

[1] https://mesonbuild.com/
[2] https://ninja-build.org/

```
1 julea-config --user \
2 --object-servers="$(hostname)" --kv-servers="$(hostname)"
    ↪ --db-servers="$(hostname)" \
3 --object-backend=posix --object-component=server
    ↪ --object-path="/tmp/julea-$(id -u)/posix" \
4 --kv-backend=lmdb --kv-component=server
    ↪ --kv-path="/tmp/julea-$(id -u)/lmdb" \
5 --db-backend=sqlite --db-component=server
    ↪ --db-path="/tmp/julea-$(id -u)/sqlite"
```

Listing 3.4: Setting JULEA configuration

3.2. JULEA users types

JULEA basically has two different types of users, which are briefly described here. These should be included in the design process.

- **JULEA-Developer**: Users who want to further develop JULEA have to deal with all development steps. This includes checking out the project from GitHub and installing all required dependencies. Furthermore, Meson must be used for configuring and Ninja for compiling. This requires experience or at least understanding of the commands in the documentation.
- **JULEA-User**: The user who only wants to use JULEA would be provided a way to install JULEA via a package manager using spack [3]. For this user, less prior knowledge about the configuration of JULEA is required.

3.3. Technical Challenges

At the beginning, it should be clarified which technical challenges [Boe15] exactly are supposed to be improved or solved with the presented solution.

- **"Dependency Hell"**: To install or build software, for example to reproduce a result or just to continue developing on the project, can be challenging. Each developer or user has a different user environment, which results in different software packages, environment variables and configurations.
  To improve this problem, images can be used that are made available to other users. The image contains the installed and configured software.
- **Documentation**: In a software life cycle, an application is further developed and modified. In the process, the documentation must also always be adapted,

[3] https://spack.readthedocs.io/en/latest/

which is often neglected. Even a small forgotten step, e.g. the skip of a necessary dependency, can lead to the failure of the installation.
Definition files, such as a Dockerfile, are used to define how a given image was built. This simple script specifies how the software was installed and configured in a particular environment.

- **Reproducibility**: As written in the documentation point, software changes. Fixing bugs or adding new features can affect the result of a different experimental setup. A solution must be found that allows to show with which version and configuration of the software a result was generated, so that others can reproduce the results.

3.4. Concept

First, we introduce two general concepts of how container images could be constructed. After that, the individual container engines will be discussed in more detail.
As an example for consideration, a server and client are created for each container engine. The client should connect to the server and use the server as a backend.

3.4.1. General Concept for container images

Every single step specified in 3.1 must also be used to create the container images. There are two potential ways to do this:

- **One image:** The first and naive approach is to specify all steps in a single configuration file, e.g. Dockerfile, and create a container image from it. This means that both the download of the dependencies and the individual configuration steps are performed one after the other in this file. The only dependency on another image is the base image, which could be an Ubuntu or CentOS version, for example.

- **Two images:** The other approach is to create one image for the dependencies and another for the configuration. The configuration file for the dependencies will again be built on a base image and then download all necessary dependencies. The second image will then import the first and then only need to configure JULEA. The only difference here is the separation of the individual steps.

3.4.2. Docker

For the design of Docker images to make the best use of them, there are recommendations from Docker, which are used for the construction of the images and are explained in more detail in this section [Inc20d].
A special focus here is on the use of the cache. Steps that have already been performed once should ideally not be repeated again in the development process. Another important factor is the image size which should be as minimal as possible.

- **Order of build steps**: The steps should be arranged so that the commands that contain content that changes less often, such as installing packages, are at the beginning. Commands that, e.g. copy content to the image, should be further at the end because this data potentially changes more often during development. If the cache behavior of a step changes, all steps below it also become invalid.
- **Reduce number of layer**: Each *RUN* command creates a new layer in the image and the number of layers should always be kept as less as possible [Inc20d]. Therefore *RUN* commands should be combined if possible. This can be applied particularly well to the installation of packages, e.g. by means of *apt*. All packages should be installed within one *RUN* command. Another advantage is that the cache behavior improves again.
- **Specific COPY**: Everything that is copied into the image using *COPY* should be well considered. Only the files or folders that are really needed for the single step or for the current image should be copied. Any change within the files will result in the cache no longer being usable.
- **COPY/ADD**: The use of *ADD* should be avoided. Both offer almost the same functionality, but *ADD* offers additional features like downloading and unpacking sources, which can lead to unwanted behavior. This way, a zipped folder could be unzipped automatically instead of just being copied into the image [CSW+17].
- **Reduce image size**: Package managers like *apt* offer the possibility to not install unneeded dependencies with the flag *–no-install-recommends*. In the same *RUN* statement the cache of the package manager should be cleared, so that it is not written to the image.
- **Specific tags**: Whenever possible, concrete information about the versions used should be given and the *latest* tag should be omitted.
- **Persistent data**: If possible, data should not be stored in the writable layer of the container, because this increases the size of the image. Instead, *volumes* should be used. These also offer better I/O performance compared to layers. In addition, *bind mounts* should only be used during the development and *volumes* for production [Inc20d].

These points give some clues that will be used for the later implementation. Another point raised in the documentation is the advice to build an own base image if there are multiple images that have a lot in common. Likewise, multistage builds are a good option. These suggested options are helpful for planning the actual build of the images. Furthermore, it is described that a container should only serve one purpose, making it easier to reuse and scale that container in multiple places. For JULEA this will mean that each container will have only one JULEA process running, e.g. one container a client and the other a server [Inc20c].
A single Dockerfile is created for the server and client, which will be built as a base image

on a configured JULEA. Everyone has the possibility to configure the respective node individually. This will happen via a startup script. As soon as the container is started, the JULEA configuration is set at the server, which includes the hostname and the port. Also, the server must be started at the end. When the client is started, it must include these two parameters in its startup process and set them in its configuration.

3.4.3. Podman

Podman offers practically all possibilities that the files created and used for Docker can also be used. This should allow Docker images or Docker files to be reused. Since Docker will be considered for the thesis and images will also be created in this process, the best approach here will be that all implementations are first created with the help of Docker. The created images are attempted to be reused for Podman in the next step. It is assumed that no separate images or configuration files need to be created for Podman.

3.4.4. Singularity

In the text Kurtzer et al. a typical workflow with Singularity [KSB17] is described. The workflow starts with a user developing the image on his own system. On this system, the user usually has root privileges, which are necessary to make changes to the image. Once development on the image is complete, it can be shared with other resources.
On shared computational resources, which can be a server, for example, the image can be viewed as its own program, allowing it to run normally. On these resources, users usually do not have root privileges, which means that no changes can be made within the image. It can only be used. If changes are to happen, it must be switched back to the local user endpoint and the image must be re-uploaded again for everyone.

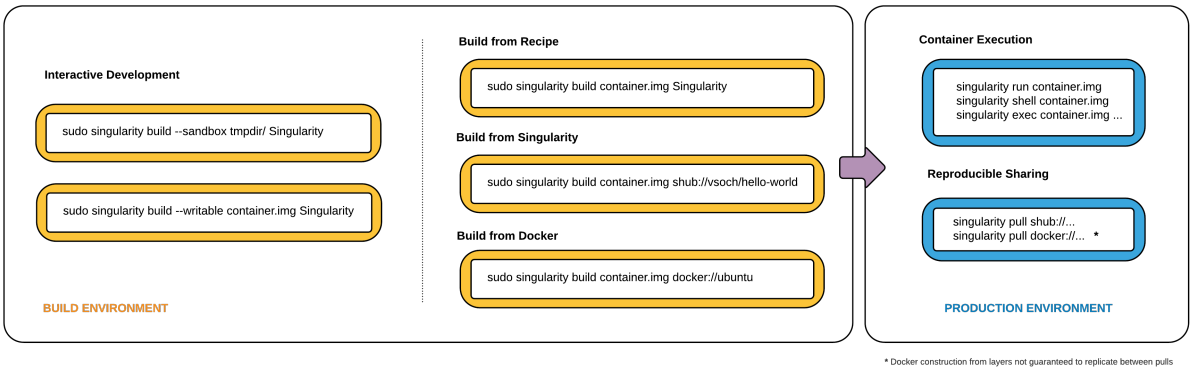

Figure 3.1.: Singularity usage workflow [Sin18]

Singularity follows the approach that a container does not run completely isolated from the host system. This aspect is especially noticeable in the file system. Because

Singularity replaces the file system of the host with the one in the container, none of the host files are accessible within the container. In order to still be able to read files from the host system, there are two possible procedures.

- **System-defined bind paths**: The administrator of the system can specify which directoires should be included automatically. Some are defaulted, which include the following: *$HOME, /tmp, /proc* and */sys*.
- **User-defined bind paths**: If the administrator has allowed it, the user has the possibility to define own bind points. This can be done via the *–bind* flag.

Singularity offers so-called bootstrapping, which follows the idea of creating an image from a previously created template. This can either be created from scratch or existing Docker images can be used.

Kurtzer et al. described that on a shared resource no root rights are needed to run Singularity. This is only partially correct. For a sample implementation, a client-server model is to be set up, where it is necessary to establish a connection between client and server for communication. In the documentation of Singularity it is written that currently root permissions are necessary for this.

The standard workflow described would theoretically also be an option for JULEA. However, a certain degree of flexibility would be lost because a singularity image would have to be available at a central location for both a client and a server, which a user would then have to download in order to use it. This would be the only way to prevent a user from having to build a container themselves. Since root rights are required to use a network with Singularity, the standard workflow can also be modified here so that the user can build and use the image himself on a system.
Also, care must be taken as to where the JULEA configuration is stored. So far it is stored in the path *./config/julea/julea* in the home directory of the user. If the container is now to be shared with other users the configuration is no longer present because the user's home directory is only mounted in the image and is not permanently there. This is only a limited problem. The Docker image with a preconfigured JULEA is used as the base image, which is used by a Singularity description file. This contains a startup script that sets the appropriate JULEA configuration for the user. To use the container on another system, the user only need to build a container using the Singularity description file.
One must be aware that it is not possible to write data within the container. It must be written either via the system paths or user defined endpoints. The data is, therefore, always located on the host system. The workflow, e.g. a user wants to compile files with the help of the container, could be done as follows. The user places the files to be compiled in a directory that is mounted. In order to create the binaries, one of the mounted paths must be specified as output directory in the JULEA configuration.

Singularity provides several ways to interact with processes in the container. For the implementation, instances are mainly used, which is an isolated and persistent version of the container that runs in the background. This service can then be used by multiple clients, which is suitable for the example implementation of a client-server-structure [Inc20i]. It must be figured out if the instance can be started without root rights and the application works without problems.

If JULEA is to be built and used on a system without the appropriate permissions and without the need for a network connection between individual containers, a container can be built with the help of the *–fakeroot* option. An instance can then simply be started.

3.4.5. Image structure

By taking a closer look at the individual container engines, the image structure for the implementation can be specified more precisely. Due to the widespread use of Docker and the resulting possibility for Podman and Singularity to use Docker images, the structure is determined primarily from the description in Section 3.4.2.

To use JULEA, a large number of dependencies have to be installed at the beginning, which takes the most time depending on the mode and system. These dependencies are changed only in a large period. From the points explained in Section 3.4.1 and this further aspect the approach of the two images is pursued. The possibility of using only one image is not considered in detail.

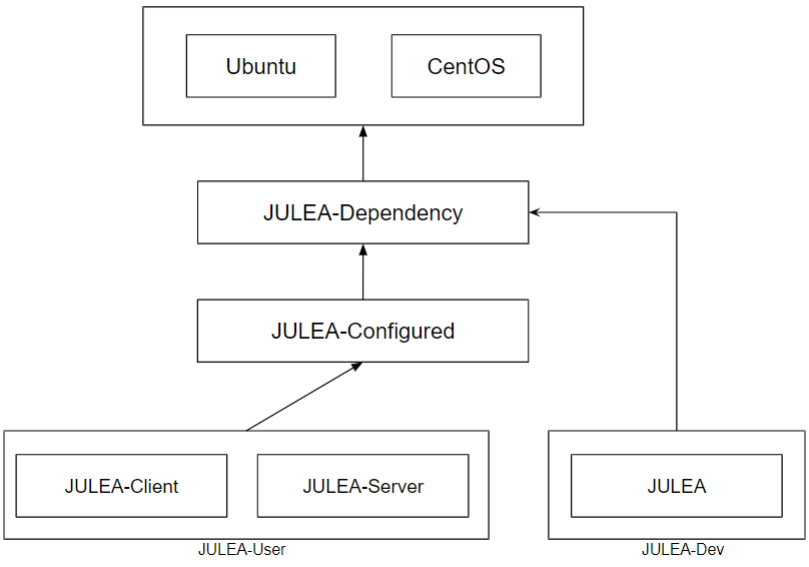

Figure 3.2.: JULEA images hierarchy

Figure 3.2 shows the more detailed structure of the image hierarchy used for implementation.

A separate dependency image is created for each Linux distribution on which JULEA will run in the future. This image contains all dependencies that will be needed for the configuration in the future, e.g. a Python version. Furthermore, this image is used to install the dependencies that are specified in the installation script of JULEA.
On this image, depending on which distribution is needed, the configuration can then be built. This image will configure and compile a JULEA.
The end user only has to decide which image to use. The JULEA-Developer might want to use the dependency image so that he has all the necessary dependencies, but wants to take care of compiling the application himself. The pure user will fall back on the ready configured JULEA image, which only has to be started with a container.

In the following *JULEA-Dependency* and *JULEA-Configured* are used to refer to the JULEA dependency image and to the final configured JULEA image.

3.4.6. GitHub Actions

The JULEA project is managed on GitHub and is publicly available to everyone. Currently, there are already some actions that are used for testing and building JULEA. By introducing containers, new possibilities arise to extend the actions further.
So far Docker Hub[4] is the best way to use and publish Docker images for other users to benefit from. Still in beta, but already usable on GitHub is the new container registry from GitHub [Nga20]. This creates the possibility to publish container images directly on GitHub, which would simplify the administration in this case, since no further system has to be integrated. In addition, it is immediately visible to users that containers have been created and can be used.

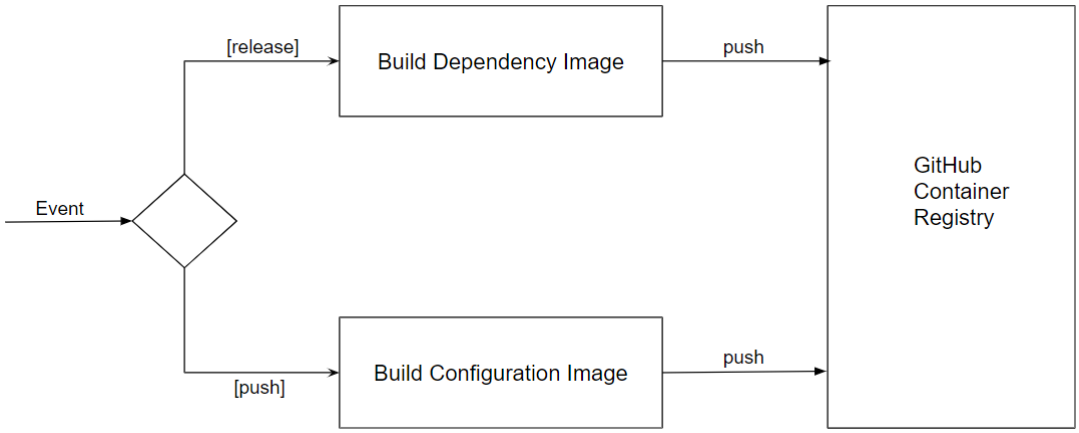

Figure 3.3.: GitHub Action workflow for JULEA

The GitHub developers already offer an action that creates a new container image of the repository on a selected event and then uploads it to the container registry. In

[4]https://hub.docker.com/

Section 3.4.5 a structure of two images was introduced. This should also be done in the actions.
Since it is assumed that dependencies rarely change, *JULEA-Dependency* does not need to be generated for each push event. For example, a workflow for this would be to change the image only on a release, assuming that dependencies only change on a new release version. Other event options would also be possible.
JULEA-Configured could be rebuilt on every push event, because it is assumed that this happens in a short time. So, there would always be a current image of JULEA available.

4. Implementation

In this chapter, the realized implementation to the design plan is explained further. It also discusses problems and explains some technical aspects in more detail.
All results presented in the implementation can be found in the following repository: `https://github.com/vittel1/julea`.

4.1. Docker

In this section, the implementation using Docker is explained. At first, the structure of the Dockerfiles will be shown. A distinction is made between a JULEA-Server and a JULEA-Client, which will also be reflected in the structure of the Dockerfiles. Then it will be shown how the images can be created and used. It will also be explained how the two container can be connected to each other via a network so that the client applications can later reach the server.

In addition, the following section uses the documentation for Docker commands [Inc20e].

Since the decision was made to create several images, the structure of the image for the dependencies is shown first in Listing 4.1.

```
1  FROM ubuntu:20.04
2
3  COPY container/docker/github/install-packages.sh .
4  RUN ./install-packages.sh && rm install-packages.sh
5
6  WORKDIR /julea
7  COPY scripts/ ./scripts/
8
9  SHELL ["/bin/bash", "-c"]
10 RUN ./scripts/install-dependencies.sh
```

Listing 4.1: JULEA-Dependency Dockerfile with Ubuntu 20.04

The Dockerfile in Listing 4.1 is built on an Ubuntu 20.04. In order to use the *install-packages.sh* script in the *RUN* statement, it must be copied from the host to the container filesystem. The installation of necessary packages is outsourced to a script to be more flexible. As described in the design, the image should be kept as small as possible using

–no-install-recommends. However, the Git package must also be installed, which could be installed with this flag, but it would cause problems later and prevents further correct installation. In order to only have to use one *RUN* instruction, the script procedure was used. Since the script is no longer needed afterwards, it is deleted directly.
Line 6 specifies the directory that is to be used for all of the following commands. Since the directory did not exist before, Docker creates it in the same step.
JULEA already has a script that installs all dependencies. For this purpose, the entire *scripts* folder is copied into the image in order to provide all necessary additional files that the script uses in the image. It must be ensured that a folder is specified for the destination, because COPY only copies the contents of the folder. Finally, the installation script of JULEA must be executed.

```
1 sudo docker build -t julea-ubuntu-deps-standard:1.0 -f
    ↪ container/docker/github/Dockerfile_Ubuntu_Deps_Standard
    ↪ .
```

Listing 4.2: Docker build commands for *JULEA-Dependency*

Depending on the system and mode, the Docker build process Listing 4.2 could take some minutes.

The image just created is now available in the local Docker Registry. From there it can be started or used by other images. The second step, creating the image for configuring JULEA, uses the image as seen in Listing 4.3.

```
1 FROM julea-ubuntu-deps-standard:1.0
2
3 WORKDIR /julea
4 COPY . .
5
6 SHELL ["/bin/bash", "-c"]
7 RUN . scripts/environment.sh && \
8 meson setup --prefix="${HOME}/julea-install"
    ↪ -Db_sanitize=address,undefined bld && \
9 ninja -C bld
```

Listing 4.3: JULEA-Configured Dockerfile

Here, the same installation directory is specified as for the dependency image. The difference here is that the entire JULEA directory is copied into the image. This is necessary so that the application can be used completely later.
In the *RUN* command, the environment variables are set by the existing script. In addition, the program is configured and compiled in the same course. In fact, it is important to point out here that the setting of the environment variable and the *meson*

and *ninja* commands must be executed together, i.e. via the *&&* link. First, it was tried to execute each command as a single *RUN* statement. As a result, the environment variables are not available in the second step. This can probably be attributed to the fact that Docker creates a new layer for each *RUN* command and the environment variables are no longer available in the new layer. Docker provides the ability to set environment variables for all subsequent commands via *ENV*. However, this is currently not possible via a script, because of that this option is out of the question. Thus, it is currently necessary to always make sure that the environment variables are set, which will also be seen later in various scripts.

After also building *JULEA-Configured*, the image is ready to be used. As already written, a JULEA-Server and a JULEA-Client are created for the example. First the configuration of the server is shown and explained in Listing 4.4.

```
1 FROM julea-config:1.0
2
3 EXPOSE 9876/tcp
4 COPY docker-entrypoint.sh /entrypoint.sh
5
6 ENTRYPOINT ["/entrypoint.sh"]
7 CMD ["juleaServer", "9876"]
```

Listing 4.4: JULEA-Server Dockerfile

Docker provides with the *EXPOSE* instruction, a command that can be used to specify that the container listens on a specific port at runtime. It can also be specified whether the port listens to TCP or UDP. However, the port is not published by the command, this specification serves more as documentation for the users who will later use the container.
In the same directory where the Dockerfile for the server is located, a script is created that is first copied into the image.
As briefly described in Section 3.4.2, a container should be designed so that it is available for a specific task and process. As soon as this process is finished, the container is also stopped. Thus, the container lifecycle is based on the process running in it. In order for the container to know which process to run, this must be specified via a command. Docker offers two possibilities for this with *CMD* and *ENTRYPOINT*, which are explained briefly in the following [Sim20].

- ***CMD***: This command can be used to specify the default executable action of the container. If *docker run* is called without a command argument, then the container will run with the action specified in the *CMD*. However, if an argument is specified, the command is overwritten and the action in the argument is executed. There can only be one *CMD* command, otherwise the last one is always taken. The command is not executed at build time of the image.

- ***ENTRYPOINT***: With this command it is also possible to specify an instruction that the container will execute at startup. However, this command cannot be overwritten even if an argument is passed to the *docker run* command.

Both commands can also be combined. The executable can then be specified by *ENTRYPOINT* and the default parameters in *CMD*. This procedure is also chosen in this example. *CMD* defines the default hostname and port of the JULEA-Server and in *ENTRYPOINT* the start script Listing 4.5 is specified.

```
1  #!/bin/bash
2  serverName=$1
3  portNumber=$2
4
5  . /julea/scripts/environment.sh
6
7  sed -i -e '$a. /julea/scripts/environment.sh' ~/.bashrc
8
9  julea-config --user \
10     --object-servers="$serverName:$portNumber"
11     --kv-servers="$serverName:$portNumber"
12     --db-servers="$serverName:$portNumber"
13 [...]
14
15 julea-server --host "$serverName" --port $portNumber
```

Listing 4.5: JULEA-Server entrypoint script

Listing 4.5 takes the passed arguments in line 2 and 3 and writes them into variables for better comprehensibility. So that, when working with a bash console inside the container later, the environment variables are set. These are entered into the *.bashrc* file, which is responsible for configuring the bash at startup (line 7). Afterwards the JULEA configuration can be set by setting the future server name and port. Finally, only the server must be started with the appropriate parameters. As long as this process is running, the container will also run. The server must not be started with the daemon parameter, otherwise the container will stop.
It would also be possible to define this whole configuration, which is executed in the script, in the *ENTRYPOINT* directly. This was also tried at first, but it made the command very confusing and complicated, because again the environment variables must be set.
Thus, the JULEA-Server image can be built and then the container can be started as shown in Listing 4.6. With *docker exec* one can work directly inside the container. When running this command, the environment variables should be set and then the container's bash console should be displayed.

```
1 cd ./container/docker/server
2 sudo docker build -t julea-server-image .
3 sudo docker run -it -d --name julea-server
    ↪ julea-server-image
4 sudo docker exec -ti julea-server /bin/bash
```

Listing 4.6: JULEA-Server build image and run it

If the default parameters should be overwritten, only the parameters at the end of the *docker run* command must be specified as seen in Listing 4.7.

```
1 sudo docker run -it -d --name julea-server
    ↪ julea-server-image SERVERNAME PORT
```

Listing 4.7: JULEA-Server run image with parameters

The same Dockerfile is created for the client as for the server. A startup script is also copied into the image and the default values for hostname and port of the server are set. It is important to note here that when starting the container, if the default values of the server container are overwritten, these must also be overwritten for the client.
The main difference is in the last command of the script in Listing 4.8. No server can be started for the client. Thus another command must be specified, which runs permanently as a process, so that the container is not stopped again immediately after starting. Here a little trick was done by permanently creating a process with *tail -f*, which actually has no meaning for further use. Whether this is the best approach should be reconsidered in the future.

```
1 [...]
2 tail -f /dev/null
```

Listing 4.8: JULEA-Client start command in Dockerfile

Afterwards, the image for the JULEA-Client can also be built and started.
Volumes are a good solution to compile local files within the container more dynamically. Thereby the directory on the local filesystem could be mounted in the container. The specification must happen over the *-v* flag by the start of the container in Listing 4.9.

```
1 sudo docker run -it -d -v
    ↪ /absolute/path/to/dir/build:/build --name julea-client
    ↪ julea-client-image
```

Listing 4.9: JULEA-Client start with volume

To enable a connection between client and server, it is necessary to set up a network.

For this purpose, such a network must first be created in Docker in Listing 4.10.

```
1 sudo docker network create juleaNetwork
```

Listing 4.10: Docker create network

For the server, when starting the container, the hostname, port, and network previously defined must be specified here in Listing 4.11.

```
1 sudo docker run -it -d --network-alias juleaServer -p
   ↪ 9876:9876/tcp --network juleaNetwork --name
   ↪ julea-server julea-server-image
```

Listing 4.11: Docker server container with network

To be able to communicate with the server, the network must also be specified by the client in Listing 4.12.

```
1 sudo docker run -it -d --network juleaNetwork --name
   ↪ julea-client julea-client-image
```

Listing 4.12: Docker client container with network

The next section introduces Docker Compose, which eliminates the need to create a network and include it for each container, making it easier to use.

4.1.1. Docker Compose

In this section, the server-client example is shown again using Docker Compose. The YAML-file Listing 4.13 is located in the root directory of the JULEA project.
Two services are created with the name *juleaServer* and *juleaClient*. The name assignment here is important, because the service name becomes the hostname at the same time. Thus, the service name must be specified for the client in the start script or Dockerfile, otherwise the client cannot reach the server.
The build contexts each reference the Dockerfile in the corresponding folder. Both an image name and a container name are defined for both services, because otherwise random names are assigned, which can lead to confusion.
Two ports are specified for the server. The number on the left refers to the host and on the right to the container. Thus, the host as well as the container can be reached via port 9876.
A volume is defined for the client, which maps the example directory of JULEA to the path */build* in the container.
It is not longer necessary to create a network manually. This is created automatically and named with *julea_default*.

```
services:
    juleaServer:
        build:
            context: ./container/docker/server
        ports:
            - "9876:9876"
        image: julea-server-image
        container_name: julea-server
    juleaClient:
        build:
            context: ./container/docker/client
        volumes:
            - "./example:/build"
        image: julea-client-image
        container_name: julea-client
```

Listing 4.13: Docker Compose for JULEA-Server and Client

To start both containers only one command is needed in Listing 4.14.

```
sudo docker-compose up -d
```

Listing 4.14: Start container with Docker Compose

4.2. Podman

In order to use Podman, no further preparation is necessary, except for installing Podman. Because Podman allows to use Dockerfiles and images that have already been built, using Podman is basically like using Docker. On top of that, many commands, for example to build an image or launch a container, are the same, which makes using Podman feel like using Docker. When taking advantage of the possibility to use the alias *docker=podman* [wha20], a difference is almost impossible to notice.

All examples explained in Section 4.1 can be recreated exactly the same way with Podman.
However, one difference was noticed. In order to test whether the test setup is correct and to check whether JULEA works properly, the example in the *examples* folder was always compiled and executed for all examples. With Docker, there were no warnings or errors that occurred. Whereas with Podman, a warning appeared at the end of the samples that something could not be executed properly in Listing 4.15. It seems that the message has no influence on the actual result, the saving of the data. The data is correctly created in the specified directory. The error probably refers to an event that

happens afterwards. However, this should be examined more closely later to see whether the error message has an influence on the result. A presumption was here that possibly the absence of the root privileges can have influence on something. However, the error did not occur later with Singularity, which may invalidate this theory.

```
1 root@2445390c9c15:/julea/example# ./hello-world
2 Object contains: Hello World! (13 bytes)
3 KV contains: Hello World! (13 bytes)
4 DB contains: Hello World! (12 bytes)
5 ==276==LeakSanitizer has encountered a fatal error.
6 ==276==HINT: For debugging, try setting environment
    ↪ variable LSAN_OPTIONS=verbosity=1:log_threads=1
7 ==276==HINT: LeakSanitizer does not work under ptrace
    ↪ (strace, gdb, etc)
```

Listing 4.15: JULEA example in Podman container with notification

In order to use the Docker Compose file in the root directory, an additional package must be installed, as this is not supported by Podman by default. For this purpose, a community project is offered that provides this functionality [Pod20]. With this project it is also possible to execute the *docker-compose.yml* file.

4.3. Singularity

This section shows how JULEA can be used with Singularity. To a large extent, this can build on what has already been explained in Section 4.1.

The server definition file in Listing 4.16 builds on *JULEA-Configured*. The syntax is a little different, however, this basically does exactly the same as the previous ones in Docker. Using *%files*, a file can be copied into the image. After that, one only needs to define a script that starts the desired process when the instance is started in the container.

```
1 Bootstrap: docker
2 From: julea-config:1.0
3
4 %files
5     singularity-entrypoint.sh /singularity-entrypoint.sh
6
7 %startscript
8     /bin/bash /singularity-entrypoint.sh
```

Listing 4.16: JULEA-Server description file with Singularity

One difference can be seen in the startup script. As written in Section 3.4.4, Singularity mounts the */tmp* directory into the image by default. If this directory is no longer mounted by the administrator or it is not desired to write to this path, this can be changed in the script. Listing 4.17 shows an example for the sqlite directory.

```
1 --db-path="/singularity-mnt/julea-$(id -u)/sqlite"
```

Listing 4.17: JULEA output path in Singularity

Care must be taken to ensure that the specified directory is either mounted by default or set by the *bind* flag, as seen in Listing 4.18.

```
1 cd ./container/singularity/server
2 singularity build --fakeroot julea-server.sif Singularity
3 singularity instance start --bind
   ↪ singularity-mnt/:/singularity-mnt julea-server.sif
   ↪ julea-server-instance
```

Listing 4.18: Singularity commands to start JULEA-Server

The shown commands describe the possibility to build a JULEA container with Singularity on a system without root privileges with the use of the *fakeroot* option.

With this described solution it is not yet possible to communicate with the server instance. To change this, Singularity's network virtualization must be used, which only works with root privileges [Inc20i]. The instance startup must be extended by the *–net* option Listing 4.19. With this option the container will join a new network which will be a bridge network by default.

```
1 sudo singularity instance start --net --bind
   ↪ singularity-mnt/:/singularity-mnt julea-server.sif
   ↪ julea-server-instance
```

Listing 4.19: Singularity start JULEA-Server commands with network

To communicate to the server it is necessary to find out the IP address of the server. With the help of the *exec* command, as Listing 4.20 shows, it is possible to execute a command directly in a running instance. Because the instance is started as root user it is running in the root context which leads to the fact that all commands must be executed as root. *hostname -I* outputs all IP addresses of the host. Singularity generates an address in the subnet 10.22.0.0/16 which is configured in the bridge settings under this path */usr/local/etc/singularity/network/00_bridge.conflist*.

```
1 sudo singularity exec instance://julea-server-instance
    ↪ hostname -I
```

Listing 4.20: Singularity JULEA-Server IP address

The Singularity client image proceeds the same way as the server image. Again, a script is created that uses the hostname and port of the server. In order to resolve the hostname of the server, an *etc.hosts* file must be created as shown in Listing 4.21. In this file the previously found IP address of the server is entered as well as the hostname that was entered in all scripts. Alternatively, one can also work directly with the IP address of the server, then the additional file is no longer necessary, but then the IP address must be entered everywhere instead of the hostname.

```
1 10.22.0.5    juleaserver
```

Listing 4.21: Singularity JULEA-Client etc.hosts

In order for the instance to be used, the *etc.hosts* file must be bound when the instance is started as shown in Listing 4.22. This instance must also use the *–net* option to join the network where the server is located. Then, the *shell* command can be used to work within the instance. For JULEA to be used, it is essential to run the environment variable script again. The solution of entering it in the *.bashrc* file, which was previously used with Docker, no longer works here, since the home directory is automatically mounted.

```
1 sudo singularity instance start --net --bind
    ↪ etc.hosts/:/etc/hosts --bind
    ↪ singularity-mnt/:/singularity-mnt julea-client.sif
    ↪ julea-client-instance
2 sudo singularity shell instance://julea-client-instance
```

Listing 4.22: Singularity JULEA-Client starts instance with etc.hosts bind

4.3.1. Singulularity Compose

There is a community project [Sin20] for Singularity, so that the management of the instances can be simplified.
However, it is not possible to use the existing *docker-compose* file, as the library does not support the naming convention of Docker. Because of that a new one must be created as seen in Listing 4.23.

```
version: "1.0"
instances:
    juleaserver:
        build:
            context: ./container/singularity/server
        ports:
            - 9876:9876
    juleaclient:
        build:
            context: ./container/singularity/client
        volumes:
            - ./container/singularity/client/
            singularity-mnt:/singularity-mnt
```

Listing 4.23: Singularity Compose YAML file

This file can now be used to start two instances using *singularity-compose up*. As written in the Singularity documentation, an *etc.hosts* file is also created in the current directory with the corresponding IP addresses and hostname in Listing 4.24.

```
# cat etc.hosts
10.22.0.3   juleaclient
10.22.0.2   juleaserver
127.0.0.1   localhost
```

Listing 4.24: etc.hosts generated by Singulartiy Compose

Now it is possible to switch to the desired instance using the *shell* command. To test if the client can reach the server, first switch to the client instance. To work with JULEA, the environment variables must be loaded first. After that the example can be compiled in */julea/examples*. However, when executing it, the following error occurs in Listing 4.25.

```
Singularity> ./hello-world
(process:437): JULEA-CRITICAL **: 15:22:09.010: Could not
  ↪ connect to juleaserver: Connection timed out
(process:437): JULEA-CRITICAL **: 15:22:09.011: Can not
  ↪ connect to juleaserver:9876 [1].
```

Listing 4.25: JULEA-Client network timeout in Singularity

There appears to be a name resolution error here. In the */etc/hosts* file, the hosts created in Listing 4.24 are mounted correctly.
Switching to the server instance and displaying the running processes, the JULEA process

was started and is running. Also, here the assignment of IP address to hostname was done correctly.
Further attempts to fix the error all failed. The problem is still present and the client-server example could not be made to work with Singularity Compose.

4.4. GitHub Actions

In this section, the structure of the GitHub Actions is explained in more detail, in order to provide current images. The *JULEA-Dependency* is used as an example to show the structure.

All files that define a workflow are placed in the path *./github/workflows/*.

At the beginning, it has to be define on which event the workflow should be triggered, see Listing 4.26. As already mentioned, this should only be triggered on each publish event for a new release. Furthermore, there is the possibility to define environment variables within the script. This form is used here for setting an image name.

```
1 on:
2     release:
3         types: [published]
4
5 env:
6     IMAGE_NAME: julea-ubuntu-deps-standard
```

Listing 4.26: GitHub Action for publishing JULEA Docker dependency image

Afterwards, the actual job can be defined in Listing 4.27. Here, only the most interesting parts are briefly explained.

```
1  jobs:
2      [...]
3      steps:
4          [...]
5          - name: Build image
6              run: docker build .
7              --file container/docker/github/Dockerfile_Ubuntu
8              --tag $IMAGE_NAME
9
10         - name: Log into registry
11             run: echo "${{ secrets.GIT_TOKEN }}" | docker
                   ↪ login ghcr.io -u ${{ github.actor }}
                   ↪ --password-stdin
```

```
12
13          - name: Push image
14              run: |
15              IMAGE_ID=ghcr.io/${{ github.actor }}/$IMAGE_NAME
16              [...]
17              docker push $IMAGE_ID:$VERSION
```

Listing 4.27: GitHub Action for publishing *JULEA-Dependency*

As explained at the beginning, the actual step is subdivided into actions. Three actions can be seen in the snippet. The first action *Build image* builds the actual Docker image. With the help of the *file* flag, the corresponding Dockerfile is referenced. In addition, the environment variable defined earlier is used in the *tag* flag.
In order for the image to be pushed to the registry later, it is necessary to log in to it beforehand. To use the GitHub API without passing a password, GitHub provides a functional alternative [Inc20g] with a Personal Access Token (PAT). With the PAT, it is also possible to define permissions so that the images can be managed later. Once the token has been created, it must be stored in the corresponding repository under Secrets and given a name. In the example *secrets.GIT_TOKEN* there is a token with the name *GIT_TOKEN* under Secrets. Then, *docker login* can be used to log in to the GitHub Container Registry (ghcr.io).
In the last action, the previously created image can be pushed to the registry. In this step, the image gets a unique ID. This consists of the registry URL, the name of the creator and the image name. This ID is later the link from which the image can be downloaded to the local registry using *docker pull*. In the last step of the action, the actual pushing of the image takes place.

After the workflow has been triggered and the job has been successfully executed, the image can be found in its own repository overview under packages. By default, the image is set to private, which means that the image can only be used by logged-in users with the appropriate rights. However, the goal is to make the image available to all users without restriction. Therefore, the owner of the repository can set in the settings of the package that the image should be made public. This means that the image can now be downloaded via the ID.
These steps are repeated for *JULEA-Configured*, which is built on top of the dependency image. The only difference in the workflow is that it is created for each push event and has a different ID.
It should be noted that during the development of the project, the feature is in a beta version. In the future, the procedures may still change. In addition, the procedure for an organization can differ in individual steps, because this was only played out for a private user.

4.5. Use Case Examples

In this section, two examples of how the previously created containers can be used are shown.

4.5.1. Visual Studio Code

Visual Studio Code (VSCode) is a source code editor developed by Microsoft that runs on Windows, macOS and Linux. Some languages like Javascript are supported by default. Through extensions the editor also offers support for languages like C, C++ or Java [Mic20].
With the help of extensions, it is not only possible to use other languages in the editor, but also tools such as containers. Through Remote Containers [Mic21], Docker containers can be used as a development environment, which makes it possible to further develop the application with the IDE within a container. The user can decide whether the extension should start a container or attach to an existing one.
The Docker Compose file created in Section 4.1.1 is used for this example. After running *docker-compose up*, the running containers can be displayed, see figure Figure 4.1.

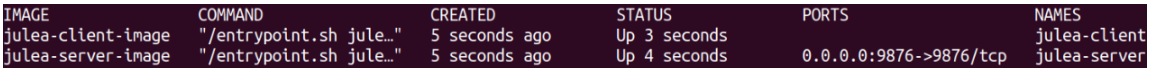

Figure 4.1.: Running Docker containers for VSCode

After the extension has been installed in VSCode, one of the created containers can be selected as shown in Figure 4.2.

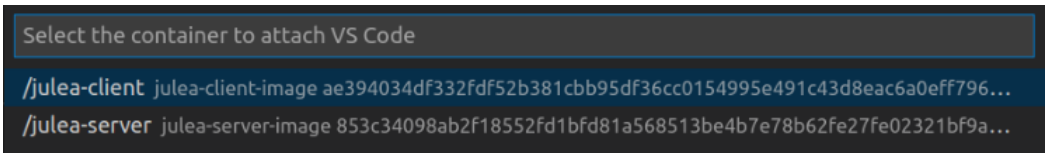

Figure 4.2.: Attach to running container in VSCode

Once one of the containers is selected, the terminal will indicate that the environment variables are being loaded. A folder within the image can now be selected in the menu and files can be modified with the help of the IDE. Modifications are written directly to the image.

4.5.2. Gitpod

With Gitpod [Git21], an open source platform, a development environment can be started automatically. For each desired task, a new and fresh development environment can be started, which is immediately available via the internet browsers.
To ensure that JULEA is configured when the workspace is started, a configuration file

for Gitpod can be used to specify that a Docker image or Dockerfile should be used. The image is then downloaded or created and mounted directly when the Gitpod instance is started. One way to start a new instance of Gitpod is by calling a URL that contains the desired GitHub repository, like *gitpod.io/#https://github.com/name/repo*.
After the workspace has finished loading, a usable development environment is ready to be used with JULEA. However, in initial testing, it was found that the scripts that run when Docker containers are started, are not executed properly here. This leads to the fact that the environment variables and the JULEA configuration must be set manually. It should also be checked again whether the JULEA server is started.
Here, only some first attempts were made to test Gitpod fundamentally. There is also the possibility to host a Gitpod instance yourself which could provide a better environment for the user.

5. Evaluation

In this section, the results and experiences are evaluated in more detail. It also looks at the advantages and disadvantages of the individual container engines used and what influence they had on the implementation. In addition, the benefits that resulted from the introduction of containers will be discussed.

5.1. Benchmark

In this section, a small benchmark test for the presented container engines is shown. For this, the benchmark script provided in the JULEA repository was used, which offers a wide range of different possibilities. It was limited here to two different parameters: */object/object/write* and */object/object/read*. The script creates objects, reads or writes and then deletes the objects again. The created images which are described Chapter 4 were used. The software and hardware used can be found at the link: `https://github.com/vittel1/julea/blob/master/test-setup.txt`.
For Docker and Podman, it is possible to specify a path in the JULEA configuration, which is available as a volume in the container. Therefore, two different tests are executed for each of the two engines. One writes to the volume and the other writes directly to the container. Since the Singularity container is read-only and the option would not exist to write directly into the container, only a test with the volume is executed. The execution of the script without container solution, i.e. on a native host system, serves as a comparison value.

The results from 10 runs were determined for each option and an average value is calculated from them. The results are shown in Figure 5.1. The left graph shows the results of the write speed. Here, the values from the native and Singularity runs are close to each other with a value of just under 100 MB/s. The two Docker tests show a difference of almost 10 MB/s between volume and writing directly to the container. If the volume is used as a path, an average speed of about 82 MB/s is achieved. The values for Podman are close to each other and are closer to the native and Singularity speed with 90 MB/s.
The graph on the right shows the results for the reading speed. Similar results are shown here for the native and Singularity measurement. Docker and Podman both show a higher speed overall compared to the write speed. Docker achieves a value of 85 MB/s for the volume measurement and Podman of 95 MB/s. Docker again shows a difference between container and volume path.
For Docker, it was already mentioned in Section 3.4.2 that volumes should be used instead of writing directly to the container for better performance. This result is reflected

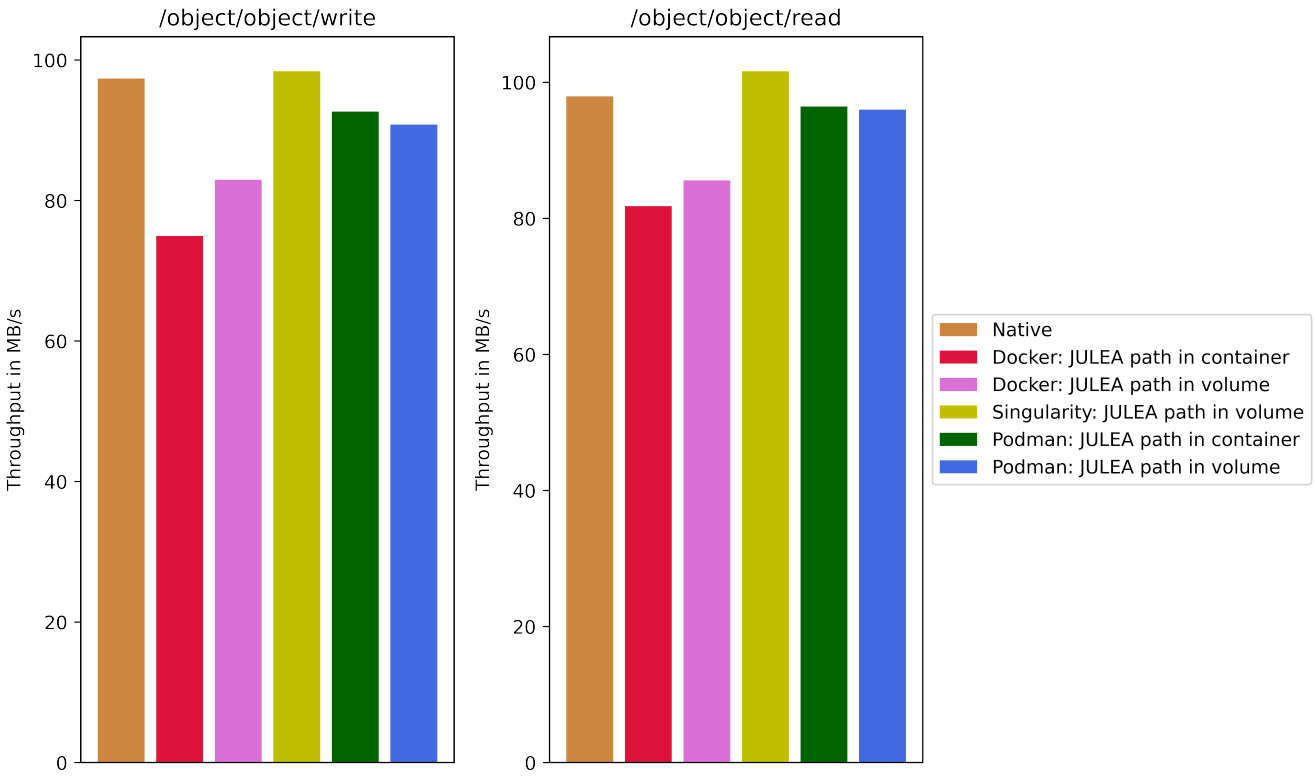

Figure 5.1.: JULEA benchmark with different container engines

here. Podman shows faster results overall compared to Docker. Furthermore, it does not seem to make a very big performance difference whether a volume is used or written directly to the container. Singularity performs best overall of all container engines and is always comparable with native execution. Overall, it can be said for Singularity and Podman that there is just a small performance overhead.
Arango et al. describe a similar result for disk I/O-Performance in their work for Docker and Singularity. Other categories such as memory or network are also tested here [ADS17].

5.2. Before/After Comparison

This section compares the use of JULEA before the introduction of the container solution and afterwards. Special attention is paid to the factors of time required, potential for errors, and usability. These factors relate to the installation and configuration of JULEA. Here, errors or possible error points that were a problem during the installation are described in particular.

Before

- **Error potential**: During the installation for the dependencies for JULEA, an

installation script is provided that automatically installs all necessary packages for JULEA. The problem here was that the necessary packages for the actual installation of the JULEA dependencies were often not yet available. For example, Spack requires that a Python interpreter is installed on the system. If this is not the case, the actual packages can not be installed at all. If Python was installed and the installation script was executed again, the next error occurred, e.g. that a package for unpacking the packages was missing. Again the installation was unsuccessful and the required package had to be installed first. This process could be repeated several times, depending on how many packages are missing on the system. It is also not immediately obvious which package is actually missing for each error. The described problem does not necessarily only occur when installing the dependencies, but can also occur when configuring if packages like *make* are not present.

- **Time requirement**: The time required can vary greatly. If, as described in the first point, a package is missing when installing the packages, this may not be noticed until halfway through or even at the end. Depending on the hardware and system, installing the dependencies in standard mode takes between 45 and 60 minutes. So, if the absence is noticed at the end of the script, the complete installation must be started from the beginning, which then takes the same time again.

- **Usability**: Meson and Ninja are used for configuration and compilation. In the JULEA documentation all necessary commands are given, but if errors should occur, the user must be familiar with the two applications in order to be able to correct the errors.

After

- **Error potential**: The sources of errors for the user have been greatly reduced. Since JULEA is now available preconfigured, it can be used directly. Errors can now occur in the use of the individual container engines, which are difficult to evaluate so far and will probably become apparent in the further course.
  The JULEA-Developer theoretically adds another error source. Because Docker images are now created, the developer also has to deal with the basics of Docker in case problems arise with it. For example, when creating the images, the environment variables have been a source of errors.
  In addition, several new GitHub Actions will need to be maintained and customized in the future.

- **Time requirement**: The biggest time effort for the JULEA-User should be downloading the image, which can vary depending on the internet connection. Since most Docker commands for working with the containers are shown as examples in the documentation, it should be possible to start working immediately.
  The time commitment here has now been moved to GitHub. Due to the fact that

an image is built for every release or push event. However, this is now no longer a time investment that the user has to spend, but time that GitHub needs to build an up-to-date image.
Building *JULEA-Dependency* with the standard mode takes about 60 minutes. However, since this is only built on release events, it should be rather rare to build a new image. The image by the push event takes between 5 and 10 minutes, which means that there is almost always an up-to-date JULEA image available.
However, if something is changed in the dependencies and a new image is built, there is also a possibility that the image cannot be built due to dependency errors. However, fixing the error and thus spending time is limited here to the JULEA-Developers or operators of the repository.

- **Usability**: For users who just want to use JULEA, the provided image with preconfigured JULEA is quick to use. The whole configuration and compilation process is omitted. However, a basic understanding of Docker is now required. This is simplified by providing a Docker Compose option but if the user want to work inside the container, a few more commands are necessary. For the JULEA-Developer, the only change is that the necessary packages no longer need to be installed, but can be started directly with the configuration and compilation.
  Another advantage in use is the provision of different JULEA versions. By creating an image for each new version, older JULEA versions can be used. In addition, there are images for Ubuntu and CentOS, for example, which makes it easier to switch to a new operating system.
  VSCode and the extension make it even easier for the end user to use JULEA, as shown in Section 4.5.1. Combining a development environment with containers reduces users' knowledge of specific container commands. Code developed on the host system no longer needs to be manually mounted and executed in a container. Since VSCode directly opens a terminal in the container, where for example the code can be compiled, it makes it seem as if the user is working directly on the local system.

5.3. Experiences with different Container engines

This section goes into more detail about each of the container engines and what experience has been gained while using them with JULEA.

- **Docker**: It is not without reason that Docker is very well known and widely used. There is a large amount of documentation and examples, which made it easier to implement the solution for JULEA. Docker Compose takes even more work out of the equation, which was especially beneficial for networking between client and server.
  On a local system where root privileges are available, there are also no restrictions. However, if Docker were to be deployed on an HPC system, users would have to be managed accordingly for Docker to be used at all.

To what extent the described security risk is a problem can unfortunately not be further assessed by this project.
Small implementation problems arose due to the layer structure, which caused repeated problems with the environment variables. However, this is not a problem of Docker, but perhaps the project to be implemented would have to be adapted to the architecture of Docker, so that the environment variables can be set here with the help of the *ENV* commands.

- **Podman**: Podman is a good alternative to Docker. Since it adopts Dockerfiles, Docker images and even the syntax to operate them, nothing stands in the way of using it. The only criticism here is that the Podman Compose project is not integrated into the Podman project, which means that further development and support is not guaranteed. Podman seems to rely on a Kubernetes solution here [wha20]. Probably the biggest advantage for JULEA's purpose is the absence of root permissions, allowing anyone on a system to use it. There does not seem to be any other issues or consequences here that limit its use either.

- **Singularity**: The implementation of Singularity was more difficult compared to Docker. Although existing Docker images can be used at the beginning, which makes it a little easier to get started, Singularity follows more of an integration approach instead of isolation, which was quickly noticed. In the examples, the JULEA configuration was always set to the output path */tmp*. With Singularity, this path is mounted by default from the host system into the container. As a result, files created by the container are immediately available on the local system. Furthermore, the JULEA configuration is written to the local home directory. With this architecture, the abstraction between host and container is lost. Whether this is an advantage or disadvantage should be questioned carefully. At the end, with Docker is possible to achieved the same goal through the volumes, that just the output is written on the host system, whereby it should not really be a disadvantage. There is also the option to no longer automatically mount the path. However, this option is mentioned in a separate documentation because a distinction is made between admins and users.
  Furthermore, one must be aware of Singularity's workflow. Building an image requires root privileges, unless one uses the *fakeroot* option. So the user should only use the finished image and not build one himself. The fact and the point that an image is read-only by default shows that Singularity has clearly put its benefit on reproducibility and portability. For the implementation of JULEA this seemed to be a disadvantage, which of course does not speak against Singularity in generals, but rather for the present use case.
  The implementation for the networking between client and server instance compared to Docker was significantly more complex. There were repeated problems with the name resolution, which led to the client not reaching the server or to errors. Although the problem was solved with the *etc.hosts* file, the way to get the IP address of the server and create a file with hostname and IP address seems too

complicated. This problem dragged through the implementation with the help of Singularity Compose and prevented an optimal use. The benefit that Singularity can be operated without root rights is unfortunately no longer possible with the network option, which means that the advantage is completely lost and the upside over Docker is fading.

Table 5.1 summarizes all described points once again.

	Docker	**Podman**	**Singularity**
Abstraction Level	High	High	Medium
No privileged or trusted daemons	No	Yes	Yes
Additional network configuration	No	No	Yes
Support Docker images	-	Yes	Yes
Compose-Feature	Integrated	Extra	Extra
I/O-Performance	Medium	Good	Good
Usability	Easy	Easy	Medium

Table 5.1.: Overview of all evaluation factors

For Singularity, the point *No privileged or trusted daemons* refers to the use without a network, which does not need root rights. The point *Additional network configuration* should clarify that an extra *etc.hosts* file with the appropriate entries was needed to establish a connection between the individual instances. Docker and Podman only needed a network here, which can be specified as a flag.

In summary, it can be said that Podman performed best of the tested container engines. Although Podman differs only slightly from Docker, the rootless feature is very interesting. In addition, it shows better performance compared to Docker, as shown in Section 5.1. Singularity shows some weaknesses in the usability for this thesis. It was not as easy to use as the other two engines and also requires manual settings of the network configuration in order to let two containers communicate with each other.

6. Related Work

In this chapter some related work is shown that deal with reproducibility on the one hand and container solutions on the other hand.

6.1. Singularity and MPI Applications on HPC Clouds

Zhang et al. [ZLP17] address the question of whether Singularity containers are ready to run with MPI applications in HPC clouds. MPI has become mostly the standard for parallel computing over the last few years. To date, many MPI applications still run natively on HPC systems. Due to the lower overhead compared to hypervisior-based virtualization, container-based virtualization offers a good opportunity for HPC cloud systems. However, initial testing showed that MPI applications in Docker have a large overhead compared to the native application. This, among other reasons, is why Singularity is getting more attention in the HPC space.
The results show that Singulartiy achieves near-native performance on memory accesses. Furthermore, only a minimal overhead is observed when using MPI-based HPC applications. This could make Singularity a promising option for future HPC projects. Singularity could also be an option for JULEA in terms of performance, because it communicates between different nodes using MPI.

6.2. Reproducible scientific workflows

Popper [JAA+17] is a convention for generating articles that can be used to make it easier for researchers to reproduce the experiments. In collaboration with PopperCI, a continuous integration (CI) service, it can be ensured that on the one hand the convention is followed and on the other hand experiments can be validated. Through the DevOps approach, only simple and few commands are required for users to re-execute the artifacts of the experiments. Popper uses container engines such as Docker or Singularity to pack software packages.
Since Popper is only a convention, it is of limited use for this thesis. There is still a risk of forgetting commands or configurations in the definition file, making the project impossible to reproduce.
There is also a need to agree on this convention with other developers so that it is applied consistently. Workflows for existing projects may also have to be adapted for this purpose.

6.3. Performance in HPC Applications

In order to run applications in the HPC field or in cloud environments, virtualization is being used more and more often, which can improve the provisioning of resources and the use of computing and storage space. Either virtual machines (VM) or container solutions like Docker are used. Chung et al. [CQHNT16] compare the performance of VMs and Docker by well-known HPC tools.
The paper points out that both virtual machines and Docker containers can have advantages and disadvantages. The decisive factor is the usage objective and the function of the application that is to run on the system. For example, a VM offers greater isolation from the host system, while a container offers a reduction in overhead. Especially sharing the OS kernel with Docker can be used as a feature to use an application efficiently. Chung et al. concluded that Docker is preferable to a virtual machine for data-intensive applications.
This finding can also be helpful for JULEA, showing that using containers is advantageous compared to a VM. Chung et al. also reiterated that containers bring advantages in scalability and portability, which is also important for JULEA.

6.4. Docker security

Bui [Bui15] analyzed the security aspects of Docker. In doing so, emphasis is placed on two points in particular. The first is Docker's internal security, which looks at Docker's isolation, for example. The second point is the interaction with the security features of the Linux kernel, which largely looks at how Docker supports them.
The analysis shows that Docker containers are quite secure. Mechanisms like namespaces or cgroups create a large level of isolation. In addition, various kernel security features are supported. The only problem presented in this paper is Docker's network model, which is vulnerable to ARP spoofing and MAC flooding. It is again made clear that a container running with privilege rights is effectively the same as if it were running on the host directly. In addition, the direct communication with the host kernel means that there is a greater potential for attack than with a virtual machine. This problem can be solved by running a container inside a VM. To further increase security, it is suggested to run containers as non-privileged user.
So in the future, if Docker is used as the container engine for JULEA, there will be no problems in terms of security. An interesting thought here is that a container could run inside a VM to increase security which could be an approach to consider in the future.

7. Summary, Conclusion, Future Work

In this chapter, the thesis is summarized and a small outlook is given on what could be further improved in the future. Also, it is figured out which questions have remained open and on which topics further attention could be paid.

7.1. Summary

This thesis showed an approach how to implement a user-space storage system using containers. JULEA was used as an example.
First, some basics were laid, which included Docker, Podman and Singularity on the one hand, and on the other hand, there was an overview of what goals a container can address. This included a brief explanation of the current approach or workflow for installing JULEA and an approach to see what a future solution with containers might look like.
In the implementation some concrete examples were shown for how JULEA can be used in a container. It was also discussed what problems were encountered and how that affected the implementation and solution idea.
In a short evalution, the experiences that were noticed during the implementation with the different container engines were shared. It was also compared how the container solution differs from the existing solution.

7.2. Conclusion

Through the thesis and especially the implementation presented in Chapter 4 it was shown that a user-space storage framework can be optimized relatively easily and quickly to a container solution.
Especially for a user who wants to use the storage framework, the process has been simplified significantly. Potential sources of error, e.g. during installation, and the time required for installation are significantly reduced by images. The hierarchical structure of the images makes it possible to switch to a different container operating system with the same configuration without major adjustments.
In addition, the solution presented laid a foundation for addressing topics such as reproducibility in greater detail later on.

7.3. Future work

So far, the container solution presented is just a start, which can be built upon on later. On the one hand, it must be evaluated which container engine will be used in the future. Three were presented, each of which has advantages and disadvantages and behaves differently.
One point that would especially help to improve the current container solution is the customization of the environment variables. If it would be possible to load these already in the image, a source of error for the later user would be eliminated and the use would be still more optimal.
So far, only a small performance test for the I/O performance of an object has been performed. JULEA offers further possibilities here, which should also be tested. Furthermore, this should not be limited to disk performance, but other values such as memory or network should also be taken into consideration.

Bibliography

[ADS17] Carlos Arango, Rémy Dernat, and John Sanabria. Performance evaluation of container-based virtualization for high performance computing environments. *CoRR*, abs/1709.10140, 2017.

[Ber14] David Bernstein. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Comput.*, 1(3):81–84, 2014.

[Boe15] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Oper. Syst. Rev.*, 49(1):71–79, 2015.

[Bui15] Thanh Bui. Analysis of docker security. *CoRR*, abs/1501.02967, 2015.

[Cha18] Doug Chamberlain. Containers vs. Virtual Machines (VMs): What's the Difference?, 2018. `https://blog.netapp.com/blogs/containers-vs-vms/`, last accessed 2021-01-22.

[CMF+16] Antonio Celesti, Davide Mulfari, Maria Fazio, Massimo Villari, and Antonio Puliafito. Exploring container virtualization in iot clouds. In *2016 IEEE International Conference on Smart Computing, SMARTCOMP 2016, St Louis, MO, USA, May 18-20, 2016*, pages 1–6. IEEE Computer Society, 2016.

[CQHNT16] Minh Thanh Chung, Nguyen Quang-Hung, Manh-Thin Nguyen, and Nam Thoai. Using docker in high performance computing applications. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57. IEEE, 2016.

[CSW+17] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. In Jesús M. González-Barahona, Abram Hindle, and Lin Tan, editors, *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 323–333. IEEE Computer Society, 2017.

[Dat18] Datadog. Docker adoption, 2018. `https://www.datadoghq.com/docker-adoption/`, last accessed 2021-01-03.

[FS12] Juliana Freire and Cláudio T. Silva. Making computations and publications reproducible with vistrails. *Comput. Sci. Eng.*, 14(4):18–25, 2012.

[Git21] Gitpod. Introduction to Gitpod, 2021. https://www.gitpod.io/docs/, last accessed 2021-01-22.

[GWR20] Holger Gantikow, Steffen Walter, and Christoph Reich. Rootless containers with podman for HPC. In Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing - ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21-25, 2020, Revised Selected Papers*, volume 12321 of *Lecture Notes in Computer Science*, pages 343–354. Springer, 2020.

[Inc20a] ACM Inc. Artifact Review and Badging – Version 1.0 (not current), 2020. https://www.acm.org/publications/policies/artifact-review-badging, last accessed 2021-01-26.

[Inc20b] Docker Inc. About storage drivers, 2020. https://docs.docker.com/storage/storagedriver/, last accessed 2020-12-03.

[Inc20c] Docker Inc. Best practices for writing Dockerfiles, 2020. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, last accessed 2020-12-18.

[Inc20d] Docker Inc. Docker development best practices, 2020. https://docs.docker.com/develop/dev-best-practices/, last accessed 2020-12-18.

[Inc20e] Docker Inc. Dockerfile reference, 2020. https://docs.docker.com/engine/reference/builder/, last accessed 2020-12-23.

[Inc20f] Docker Inc. Overview of Docker Compose, 2020. https://docs.docker.com/compose/, last accessed 2020-12-03.

[Inc20g] GitHub Inc. Creating a personal access token, 2020. https://docs.github.com/en/free-pro-team@latest/github/authenticating-to-github/creating-a-personal-access-token, last accessed 2020-12-29.

[Inc20h] GitHub Inc. Introduction to GitHub Actions, 2020. https://docs.github.com/en/free-pro-team@latest/actions/learn-github-actions/introduction-to-github-actions, last accessed 2020-12-09.

[Inc20i] Sylabs Inc. User Guide, 2020. https://sylabs.io/guides/3.7/user-guide/, last accessed 2021-01-25.

[Inc21] ACM Inc. Software, 2021. https://dl.acm.org/artifacts/software, last accessed 2021-01-27.

[JAA+17] Ivo Jimenez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Jay F. Lofstead, Carlos Maltzahn, Kathryn Mohror, and Robert Ricci. Popperci: Automated reproducibility validation. In *2017 IEEE Conference on Computer Communications Workshops, INFOCOM Workshops, Atlanta, GA, USA, May 1-4, 2017*, pages 450–455. IEEE, 2017.

[JUL20] JULEA, 2020. https://github.com/julea-io/julea, last accessed 2020-12-11.

[K+15] Gregory M Kurtzer et al. Singularity. *Retrieved July*, 8:2019, 2015.

[KD21] Michael Kuhn Kira Duwe. Deliverable D1: Report: Coupled Storage System for Efficient Management of Self-Describing Data Formats (CoSEMoS). March 2021. https://parcio.ovgu.de/parcio_media/Research/CoSEMoS/Deliverable_D1_Report-p-166.pdf.

[KSB17] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.

[Kuh17] Michael Kuhn. JULEA: A flexible storage framework for HPC. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, volume 10524 of *Lecture Notes in Computer Science*, pages 712–723. Springer, 2017.

[MCP21] Xiaoping Duan Martyn Corden and Barbara Perz. Floating-Point Reproducibility in Intel® Software Tools, 2021. https://techdecoded.intel.io/resources/floating-point-reproducibility-in-intel-software-tools/, last accessed 2021-01-26.

[Mer14] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[Mic20] Microsoft. Getting started, 2020. https://code.visualstudio.com/docs, last accessed 2021-01-22.

[Mic21] Microsoft. Visual Studio Code Remote - Containers, 2021. https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-containers, last accessed 2021-01-22.

[Nga20] Kayla Ngan. Introducing GitHub Container Registry, 2020. https://github.blog/2020-09-01-introducing-github-container-registry/, last accessed 2021-01-02.

[Pod20] Podman Compose, 2020. https://github.com/containers/podman-compose, last accessed 2020-12-27.

[RBA17] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[Sim20] Sofija Simic. Docker CMD Vs Entrypoint Commands: What's The Difference?, 2020. `https://phoenixnap.com/kb/docker-cmd-vs-entrypoint`, last accessed 2020-12-23.

[Sin18] The Singularity Usage Workflow, 2018. `https://sylabs.io/guides/2.6/user-guide/introduction.html`, last accessed 2021-01-22.

[Sin20] Singularity Compose, 2020. `https://singularityhub.github.io/singularity-compose/`, last accessed 2020-12-28.

[Var20] Pallavi Varanasi. What is Container In Cloud Computing, 2020. `https://www.cloudcodes.com/blog/container-in-cloud-computing.html`, last accessed 2021-01-26.

[wha20] What is Podman?, 2020. `http://docs.podman.io/en/latest/index.html`, last accessed 2021-01-20.

[ZLP17] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. Is singularity-based container technology ready for running MPI applications on HPC clouds? In Ashiq Anjum, Alan Sill, Geoffrey C. Fox, and Yong Chen, editors, *Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, pages 151–160. ACM, 2017.

[ZRT+18] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 174–185. ACM, 2018.

Appendices

A. Benchmark

A.1. Data

Write: Native

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/write:	1.021 seconds	(24496/s)	(100.3 MB/s)	[1.024 seconds]
/object/object/write:	1.016 seconds	(23619/s)	(96.7 MB/s)	[1.019 seconds]
/object/object/write:	1.021 seconds	(24479/s)	(100.3 MB/s)	[1.024 seconds]
/object/object/write:	1.020 seconds	(23540/s)	(96.4 MB/s)	[1.023 seconds]
/object/object/write:	1.016 seconds	(24594/s)	(100.7 MB/s)	[1.020 seconds]
/object/object/write:	1.031 seconds	(22309/s)	(91.4 MB/s)	[1.034 seconds]
/object/object/write:	1.031 seconds	(23287/s)	(95.4 MB/s)	[1.034 seconds]
/object/object/write:	1.005 seconds	(23887/s)	(97.8 MB/s)	[1.008 seconds]
/object/object/write:	1.002 seconds	(22956/s)	(94.0 MB/s)	[1.040 seconds]
/object/object/write:	1.018 seconds	(24564/s)	(100.6 MB/s)	[1.021 seconds]

Write: Docker path in container

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/write:	1.027 seconds	(19483/s)	(79.8 MB/s)	[1.032 seconds]
/object/object/write:	1.057 seconds	(17983/s)	(73.7 MB/s)	[1.064 seconds]
/object/object/write:	1.045 seconds	(19142/s)	(78.4 MB/s)	[1.050 seconds]
/object/object/write:	1.022 seconds	(13704/s)	(56.1 MB/s)	[1.026 seconds]
/object/object/write:	1.010 seconds	(19807/s)	(81.1 MB/s)	[1.014 seconds]
/object/object/write:	1.012 seconds	(18774/s)	(76.9 MB/s)	[1.017 seconds]
/object/object/write:	1.039 seconds	(19253/s)	(78.9 MB/s)	[1.044 seconds]
/object/object/write:	1.052 seconds	(19011/s)	(77.9 MB/s)	[1.058 seconds]
/object/object/write:	1.010 seconds	(16827/s)	(68.9 MB/s)	[1.015 seconds]
/object/object/write:	1.000 seconds	(18998/s)	(77.8 MB/s)	[1.005 seconds]

Write: Docker path in volume

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/write:	1.001 seconds	(19986/s)	(81.9 MB/s)	[1.010 seconds]
/object/object/write:	1.002 seconds	(19956/s)	(81.7 MB/s)	[1.009 seconds]
/object/object/write:	1.038 seconds	(21198/s)	(86.8 MB/s)	[1.043 seconds]
/object/object/write:	1.013 seconds	(20727/s)	(84.9 MB/s)	[1.018 seconds]
/object/object/write:	1.022 seconds	(17618/s)	(72.2 MB/s)	[1.027 seconds]
/object/object/write:	1.011 seconds	(20763/s)	(85.0 MB/s)	[1.016 seconds]
/object/object/write:	1.038 seconds	(20229/s)	(82.9 MB/s)	[1.043 seconds]
/object/object/write:	1.047 seconds	(21018/s)	(86.1 MB/s)	[1.051 seconds]
/object/object/write:	1.026 seconds	(20458/s)	(83.8 MB/s)	[1.031 seconds]
/object/object/write:	1.020 seconds	(20590/s)	(84.3 MB/s)	[1.025 seconds]

Write: Singularity

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/write:	1.032 seconds	(23260/s)	(95.3 MB/s)	[1.035 seconds]
/object/object/write:	1.013 seconds	(23700/s)	(97.1 MB/s)	[1.016 seconds]
/object/object/write:	1.022 seconds	(23474/s)	(96.1 MB/s)	[1.025 seconds]
/object/object/write:	1.034 seconds	(25146/s)	(103.0 MB/s)	[1.037 seconds]
/object/object/write:	1.016 seconds	(24614/s)	(100.8 MB/s)	[1.019 seconds]
/object/object/write:	1.022 seconds	(22504/s)	(92.2 MB/s)	[1.025 seconds]
/object/object/write:	1.032 seconds	(24216/s)	(99.2 MB/s)	[1.036 seconds]
/object/object/write:	1.039 seconds	(24073/s)	(98.6 MB/s)	[1.042 seconds]
/object/object/write:	1.008 seconds	(24812/s)	(101.6 MB/s)	[1.011 seconds]
/object/object/write:	1.022 seconds	(24466/s)	(100.2 MB/s)	[1.025 seconds]

Write: Podman path in container

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/write:	1.024 seconds	(23441/s)	(96.0 MB/s)	[1.032 seconds]
/object/object/write:	1.026 seconds	(23391/s)	(95.8 MB/s)	[1.030 seconds]
/object/object/write:	1.016 seconds	(22639/s)	(92.7 MB/s)	[1.019 seconds]
/object/object/write:	1.006 seconds	(22869/s)	(93.7 MB/s)	[1.010 seconds]
/object/object/write:	1.017 seconds	(22616/s)	(92.6 MB/s)	[1.022 seconds]
/object/object/write:	1.039 seconds	(23101/s)	(94.6 MB/s)	[1.043 seconds]
/object/object/write:	1.037 seconds	(23152/s)	(94.8 MB/s)	[1.041 seconds]
/object/object/write:	1.026 seconds	(21436/s)	(87.8 MB/s)	[1.030 seconds]
/object/object/write:	1.034 seconds	(23203/s)	(95.0 MB/s)	[1.040 seconds]
/object/object/write:	1.029 seconds	(20402/s)	(83.6 MB/s)	[1.034 seconds]

Write: Podman path in volume

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/write:	1.026 seconds	(23382/s)	(95.8 MB/s)	[1.031 seconds]
/object/object/write:	1.004 seconds	(22911/s)	(93.8 MB/s)	[1.007 seconds]
/object/object/write:	1.038 seconds	(22153/s)	(90.7 MB/s)	[1.043 seconds]
/object/object/write:	1.028 seconds	(21391/s)	(87.6 MB/s)	[1.052 seconds]
/object/object/write:	1.040 seconds	(19230/s)	(78.8 MB/s)	[1.044 seconds]
/object/object/write:	1.022 seconds	(22499/s)	(92.2 MB/s)	[1.028 seconds]
/object/object/write:	1.030 seconds	(21366/s)	(87.5 MB/s)	[1.035 seconds]
/object/object/write:	1.008 seconds	(22825/s)	(93.5 MB/s)	[1.013 seconds]
/object/object/write:	1.008 seconds	(22824/s)	(93.5 MB/s)	[1.012 seconds]
/object/object/write:	1.039 seconds	(23094/s)	(94.6 MB/s)	[1.044 seconds]

Read: Native

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/write:	1.009 seconds	(24784/s)	(101.5 MB/s)	[1.012 seconds]
/object/object/write:	1.032 seconds	(24223/s)	(99.2 MB/s)	[1.035 seconds]
/object/object/write:	1.026 seconds	(24378/s)	(99.9 MB/s)	[1.029 seconds]
/object/object/write:	1.012 seconds	(24703/s)	(101.2 MB/s)	[1.015 seconds]
/object/object/write:	1.011 seconds	(22749/s)	(93.2 MB/s)	[1.014 seconds]
/object/object/write:	1.029 seconds	(22344/s)	(91.5 MB/s)	[1.032 seconds]
/object/object/write:	1.030 seconds	(24278/s)	(99.4 MB/s)	[1.033 seconds]
/object/object/write:	1.022 seconds	(24460/s)	(100.2 MB/s)	[1.025 seconds]
/object/object/write:	1.003 seconds	(22936/s)	(93.9 MB/s)	[1.006 seconds]
/object/object/write:	1.031 seconds	(24258/s)	(99.4 MB/s)	[1.034 seconds]

Read: Docker path in container

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/read:	1.026 seconds	(18517/s)	(75.8 MB/s)	[1.042 seconds]
/object/object/read:	1.013 seconds	(19734/s)	(80.8 MB/s)	[1.027 seconds]
/object/object/read:	1.034 seconds	(20305/s)	(83.2 MB/s)	[1.046 seconds]
/object/object/read:	1.011 seconds	(19786/s)	(81.0 MB/s)	[1.061 seconds]
/object/object/read:	1.047 seconds	(20057/s)	(82.2 MB/s)	[1.064 seconds]
/object/object/read:	1.045 seconds	(20100/s)	(82.3 MB/s)	[1.058 seconds]
/object/object/read:	1.036 seconds	(19303/s)	(79.1 MB/s)	[1.053 seconds]
/object/object/read:	1.009 seconds	(20813/s)	(85.2 MB/s)	[1.024 seconds]
/object/object/read:	1.035 seconds	(20286/s)	(83.1 MB/s)	[1.050 seconds]
/object/object/read:	1.009 seconds	(20805/s)	(85.2 MB/s)	[1.025 seconds]

Read: Docker path in volume

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/read:	1.043 seconds	(21093/s)	(86.4 MB/s)	[1.058 seconds]
/object/object/read:	1.038 seconds	(21197/s)	(86.8 MB/s)	[1.057 seconds]
/object/object/read:	1.041 seconds	(21133/s)	(86.6 MB/s)	[1.057 seconds]
/object/object/read:	1.028 seconds	(20427/s)	(83.7 MB/s)	[1.043 seconds]
/object/object/read:	1.002 seconds	(20967/s)	(85.9 MB/s)	[1.018 seconds]
/object/object/read:	1.040 seconds	(21147/s)	(86.6 MB/s)	[1.055 seconds]
/object/object/read:	1.025 seconds	(21463/s)	(87.9 MB/s)	[1.037 seconds]
/object/object/read:	1.046 seconds	(19122/s)	(78.3 MB/s)	[1.060 seconds]
/object/object/read:	1.005 seconds	(20901/s)	(85.6 MB/s)	[1.021 seconds]
/object/object/read:	1.024 seconds	(21476/s)	(88.0 MB/s)	[1.073 seconds]

Read: Singularity

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/read:	1.013 seconds	(23701/s)	(97.1 MB/s)	[1.026 seconds]
/object/object/read:	1.032 seconds	(25206/s)	(103.2 MB/s)	[1.044 seconds]
/object/object/read:	1.002 seconds	(24938/s)	(102.1 MB/s)	[1.014 seconds]
/object/object/read:	1.031 seconds	(25226/s)	(103.3 MB/s)	[1.044 seconds]
/object/object/read:	1.031 seconds	(25211/s)	(103.3 MB/s)	[1.042 seconds]
/object/object/read:	1.007 seconds	(23832/s)	(97.6 MB/s)	[1.018 seconds]
/object/object/read:	1.009 seconds	(25765/s)	(105.5 MB/s)	[1.056 seconds]
/object/object/read:	1.015 seconds	(24635/s)	(100.9 MB/s)	[1.026 seconds]
/object/object/read:	1.031 seconds	(25209/s)	(103.3 MB/s)	[1.044 seconds]
/object/object/read:	1.024 seconds	(24407/s)	(100.0 MB/s)	[1.035 seconds]

Read: Podman path in container

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/read:	1.011 seconds	(21753/s)	(89.1 MB/s)	[1.026 seconds]
/object/object/read:	1.013 seconds	(23692/s)	(97.0 MB/s)	[1.030 seconds]
/object/object/read:	1.014 seconds	(23679/s)	(97.0 MB/s)	[1.028 seconds]
/object/object/read:	1.001 seconds	(23975/s)	(98.2 MB/s)	[1.013 seconds]
/object/object/read:	1.031 seconds	(23275/s)	(95.3 MB/s)	[1.046 seconds]
/object/object/read:	1.032 seconds	(24231/s)	(99.3 MB/s)	[1.046 seconds]
/object/object/read:	1.034 seconds	(24180/s)	(99.0 MB/s)	[1.049 seconds]
/object/object/read:	1.024 seconds	(24410/s)	(100.0 MB/s)	[1.038 seconds]
/object/object/read:	1.016 seconds	(22634/s)	(92.7 MB/s)	[1.028 seconds]
/object/object/read:	1.016 seconds	(23616/s)	(96.7 MB/s)	[1.065 seconds]

Read: Podman path in volume

Name	Duration	(Operations/s)	(Throughput/s)	[Total Duration]
/object/object/read:	1.018 seconds	(23573/s)	(96.6 MB/s)	[1.032 seconds]
/object/object/read:	1.001 seconds	(23986/s)	(98.2 MB/s)	[1.014 seconds]
/object/object/read:	1.017 seconds	(23604/s)	(96.7 MB/s)	[1.031 seconds]
/object/object/read:	1.018 seconds	(23575/s)	(96.6 MB/s)	[1.031 seconds]
/object/object/read:	1.005 seconds	(22889/s)	(93.8 MB/s)	[1.016 seconds]
/object/object/read:	1.016 seconds	(23625/s)	(96.8 MB/s)	[1.029 seconds]
/object/object/read:	1.001 seconds	(21971/s)	(90.0 MB/s)	[1.020 seconds]
/object/object/read:	1.008 seconds	(23813/s)	(97.5 MB/s)	[1.019 seconds]
/object/object/read:	1.027 seconds	(23379/s)	(95.8 MB/s)	[1.040 seconds]
/object/object/read:	1.006 seconds	(23862/s)	(97.7 MB/s)	[1.055 seconds]

List of Figures

List of Listings

List of Tables

B. Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Bachelor of Science Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

C. Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift