Frankfurt University of Applied Sciences

Master's Thesis

Containerized multi-level deployment for a distributed adaptive microservice application

Author:
Tim Wißmann

Supervisors:
Prof. Dr. Peter Thoma
Prof. Dr. Eicke Godehardt

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the course

Allgemeine Informatik Master

March 17th, 2023

Declaration of Authorship

I, Tim Wißmann, declare that this thesis titled, ’Containerized multi-level deployment for a distributed adaptive microservice application’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ________________________________

Date: ________________________________

“Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism.”

Dave Barry

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Abstract

Faculty 2 - Computer Science and Engineering
Allgemeine Informatik Master

Master of Science

Containerized multi-level deployment for a distributed adaptive microservice application

by Tim WIßMANN

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Abbreviations

LAH **L**ist **A**bbreviations **H**ere

Chapter 1

Introduction

This thesis is about the deployment of a microservice application.

1.1 Scope

1.2 Intended audience

1.3 Outline

Chapter 2

Background and related work

2.1 Application under study

OT is a open-source simulation platform developed by the university of Applied Sciences in Frankfurt, Germany. It covers features like computer aided design, 3D modeling, meshing, and flow and physics simulation (like FIT-TD and PHREEC). The projects can be administered by a user and group management (see Figure 2.1). Furthermore, all changes on a project are version-controlled. The application is designed in a way, that only a local thin-client needs to run on the users computer. After entering the login credentials (see Figure 2.2), the client securely connects to a centralized service platform where the computation is made. The results and even the UI information is sent back to the client application. This has the benefit, that also weak computers can run the application.

Figure 2.3 shows the application itself with a loaded project and a simple geometric model.

The development team consists of a small core team and several student groups during the semester.

2.2 Baseline architecture

The current system design consists of multiple levels. It is a multi-process application based on the programming languages C++ and Rust. The source code is mainly aligned to be built on Microsoft Microsoft® Windows® (Windows). A port to Unix based systems is currently in

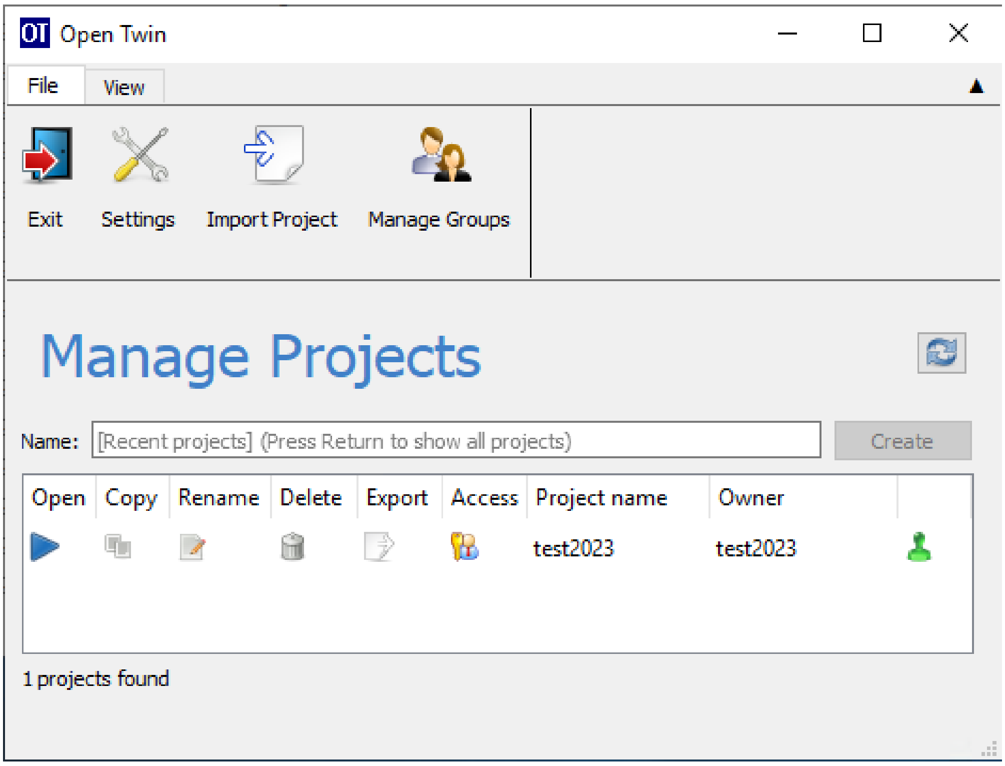

FIGURE 2.1: The OT project overview.

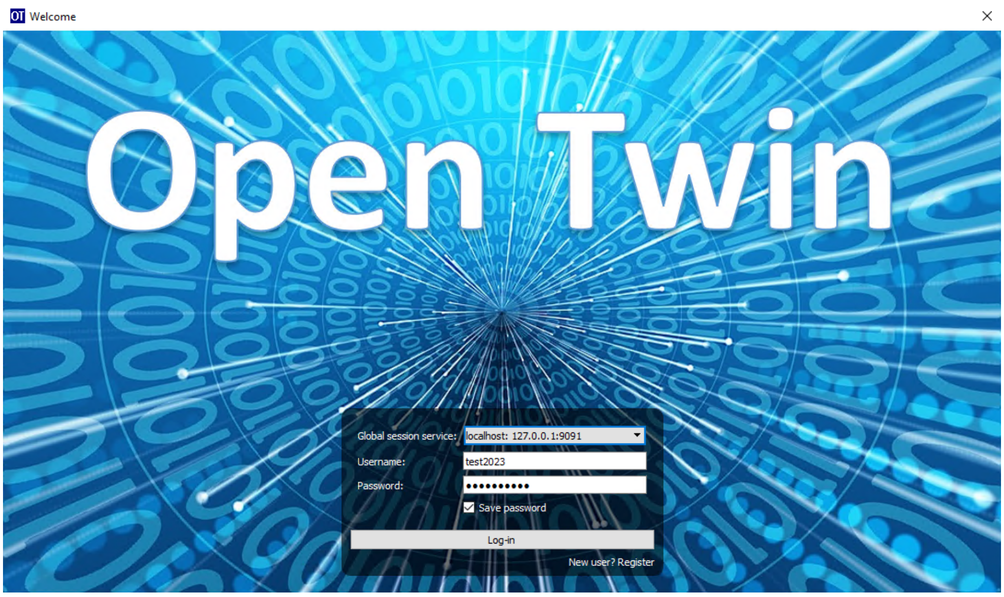

FIGURE 2.2: The OT login screen.

work. Therefore parts of the code base are aligned for multiple system architectures already, but the application is not yet able to be compiled for Linux.

Each microservice of the application is included dynamically and linked as a Dynamic Link Library (DLL) file. For starting the microservice environment, a central executable (”open_twin.exe”) is started with the corresponding arguments for the services (like binding address,

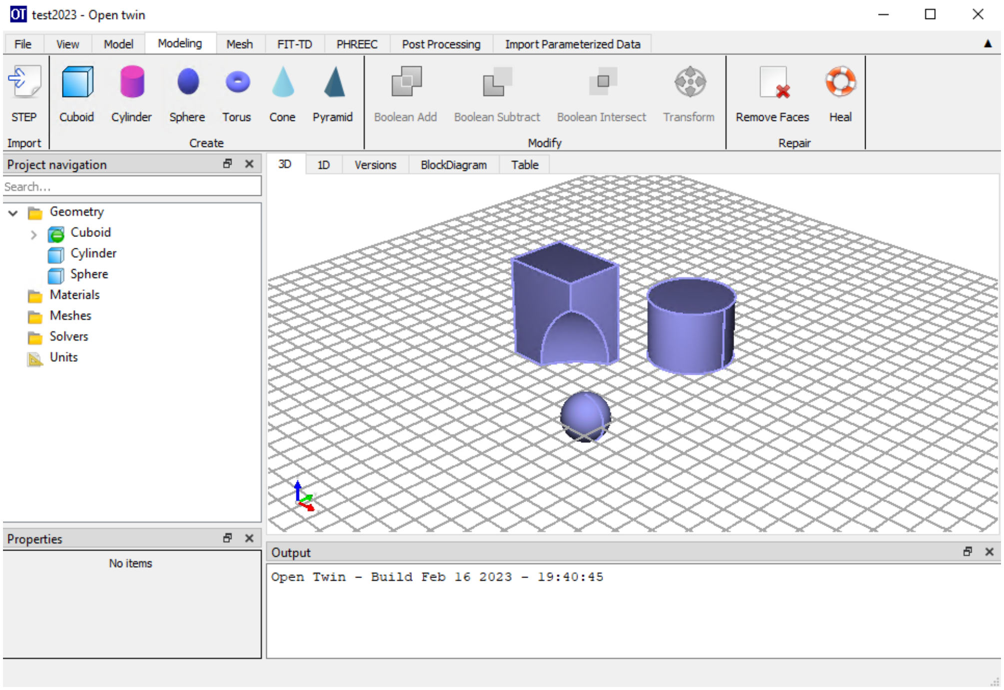

FIGURE 2.3: A opened project inside OT with a few created geometric models and subtracted computation.

port numbers, encrypted passwords) (see Listing 2.1) and the path to the DLL file itself. The UI front end, which is started by the user directly, is compiled in its own executable ("uiFrontend.exe"). ?? shows the main services and its corresponding parameters.

```
open_twin.exe GlobalSessionService.dll \
  "0" "127.0.0.1:8091" "tls@127.0.0.1:27017" "127.0.0.1:8092"
```

LISTING 2.1: Command line of Open Twin Service start

For conveniently running the services with all their necessary arguments, batch files were provided that read environment variables and convert them into runtime arguments for the service executable. Therefore, if the services are started locally, the user runs a batch file that sets up the environment for the network binding details, path to certificates and encrypted database credentials.

The system consists of the following micro services that are permanently accessible: Global Session Service (GSS), Authorization Service (AUTH) and the database. The database is running on MongoDB[1]. Another Service is the Local Session Service (LSS) that spawns the so called compute services. Those are services for running the actual computation after opening

[1]MongoDB: https://www.mongodb.com/

a project that can dynamically spawn and exit. A list of compute services and their corresponding tasks can be found in Table 2.1. Each service runs in its own operating system process.

Name	Task
CartesianMeshService	If demanded, it converts a continuous geometry into a discrete Cartesian mesh.
FITTDService	If demanded, it runs a solver algorithm for fitting and simulation.
KrigingService	If demanded, it runs a kriging interpolation of curves on geometry.
LoggerService	A background service, that accepts logging messages from other services.
ModelingService	Performs calculations for the creation, modeling and arithmetic combination of geometric data.
PHREECService	If demanded, it runs flow simulation with the help of PHREEC.
TetMeshService	If demanded, it meshes a form with an tetrahedral mesh.
VisualizationService	Runs the graphical calculations for displaying the resulting geometric data on the UI.
UiService	Creates and renders the UI elements and sends them towards the UI front end.

Table 2.1: List of compute services and their corresponding tasks.

As shown in Figure 2.4, the services can be separated by their network space. Not all services require a public available network address. While GSS, AUTH and database are globally accessible via a fixed network address, the LSS can theoretically run on a dedicated host and is only communicated to other parties after it has registered itself to the GSS. The services, spawned by LSS do not require a public address space either. All communication between the UI front end and the compute services is achieved via a relay service and a web socket communication channel.

The whole process of the LSS registration and connection of the UI front end to the compute services is depicted in Figure 2.5. Once started, the user can login. In order to connect to the database, the following steps are performed:

1. The UI front end requests further service information from the publicly available GSS. The address for this service is provided by the user. The GSS responses with Uniform Resource Locators (URLs) to the database and the AUTH.

2. The UI front end connects to the AUTH using the authentication information provided by the user.

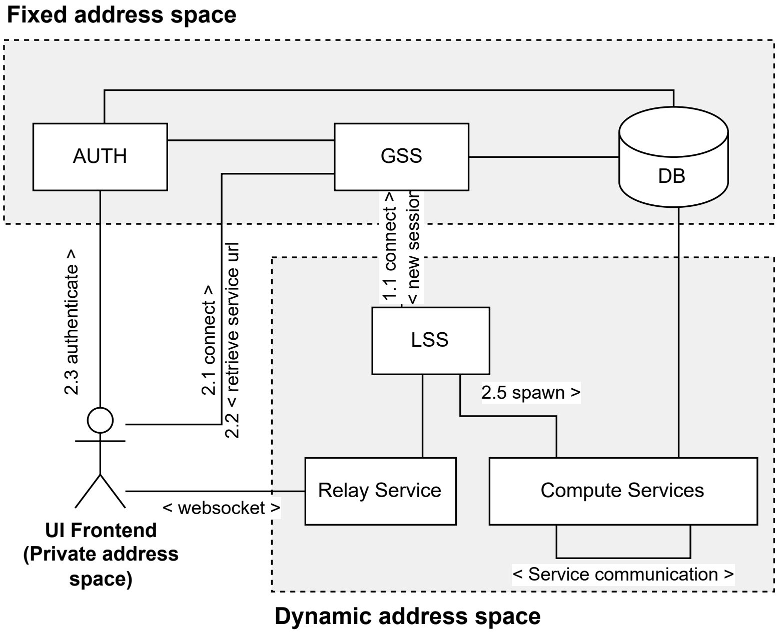

FIGURE 2.4: Communication overview and service organization for OT main services. In 1.1 the LSS registers at GSS. As soon as the UI front end connects to the GSS (2.1), service information is exchanged (2.2) and the user is authenticated (2.3). As a consequence, the GSS creates a new session and tells the LSS to spawn new compute services. From now on the UI front end communicates directly with the Compute services via the Relay Service over a web socket connection.

3. If the AUTH replies with a positive authentication, the UI front end connects to the database and lists the projects.

4. Once a project is opened or created, the UI front end requests a new session from the GSS. The GSS replies with the connection URLs from the LSS. The LSS has been registered to the GSS during its initialization.

5. The UI front end then connects to the LSS and requests a new session. As a result, the LSS spawns new application service processes and replies with the respective service URLs.

6. From now on, the UI front end communicates with the application services via the Relay service over a web socket.

The traffic between services is encrypted using mutual Transfer Layer Security (mTLS) technology. While regular Transfer Layer Security (TLS) ensures the authenticity of the server by using Certificates and the chain of trust, it does not verify the identity of the client. This is the benefit of mTLS. In mTLS, both sides, client and server has to verify their identity by providing a certificate inherited from a common root authority.

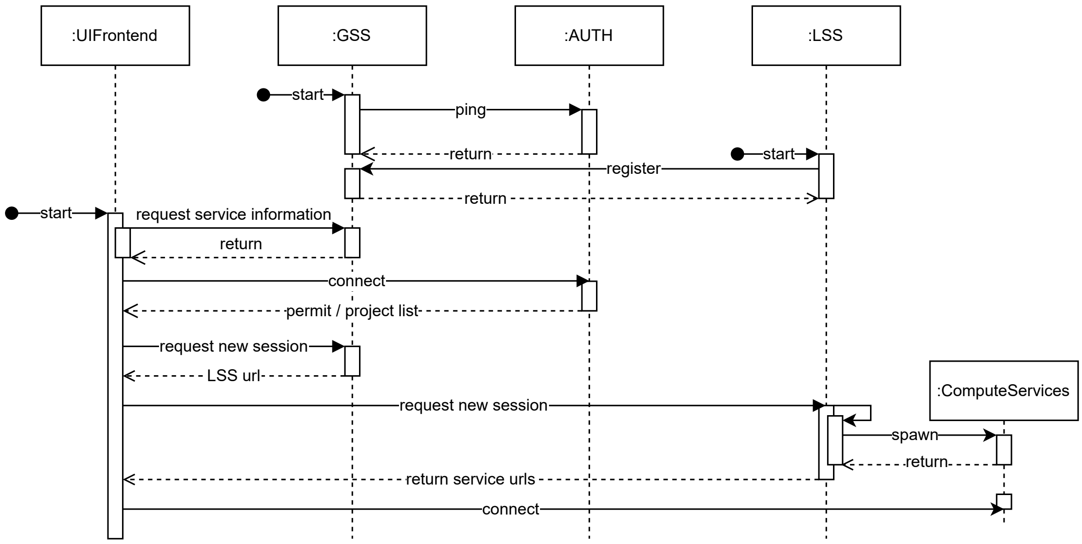

FIGURE 2.5: Service initialization of OT processes. In the beginning, the main services GSS, AUTH and an optional LSS are initialized. While the GSS checks the reachability of AUTH, the LSS registers itself at the GSS. After starting the UI front end, the service information is requested from a GSS and the user is authenticated. After success, the UI front end connects to the LSS and requests a new session. As consequence, the LSS spawns the compute services and connects them to the UI front end via a Relay Service. (Ping messages are omitted.)

2.3 Problem statement

Even though, the application is clearly based on a microservice architecture and it is able to run on a distributed system, it is not designed for a automated cluster yet. It consists of multiple processes where many of them have to run on the same system and need a full working operating system as baseline. Containerization of the system has never been tested and needs to be introduced. First, the cluster engine needs to be set up for Windows compute nodes to allow Windows containers to run inside the cluster. Additionally, it needs to have full network capabilities as well as inter connectivity between the several services. Next, the application needs to run inside containers. Therefore, container images must be created and provided for the cluster engine. Additionally, the automatic extension of services requires communication between the cluster orchestration management and the applications running on the nodes. A feature that needs to be introduced later on.

Regarding logging, while the front end application does, the microservices currently do not produce log files. Instead, only a few sub processes write the information on its standard output stream. In some cases, the error information given by exceptions is dropped. Furthermore, proper exit codes in error cases are not returned. That is, if the application exits there is currently no way to detect if the process terminated normally or crashed as part of an error.

2.4 Limitations

Due to the limited amount of time, not all code changes are applied. On the one hand, this involves the adaption for automatic extension of services. On the other hand, it implies the changes required to make the application more fault-tolerant. The changes that would be necessary, would be too extensive. Therefore, they are only made to the main processes.

As a first case study, the application is not fully containerized. Since the network connectivity is known to cause troubles in Windows container networks, there is more investigation required later on. As part of this study, only the main services are containerized and the cluster is set up to investigate the behavior in cluster environments. The actual distribution of an full functional cluster network can be part of further studies later on.

2.5 Related Work

Chapter 3

System design

Various applications for realizing the architecture have been compared. In the following sections the different options that were taken into account are presented.

3.1 Orchestration engine

Orchestration engines aggregate the processes and tools that are used to distribute services across multiple machines. Further, multiple replications are provided to maintain reliability. In addition, some solutions offer load balancing of incoming requests and network interconnection. What all of these engines have in common is that a group of virtual machines or containers, known as ”nodes”, are managed from a central spot. An administrator directs what application is run on the cluster. Based on the application's metadata, the orchestration engine then decides where to run the application by selecting a node inside that cluster.

The engine of choice was K8s because of its rich feature set. Also studies showed that K8s outperforms Docker Swarms when it comes to performance. For example, Marathe et. al. [3] compared a simple web server service deployed on a Docker Swarm cluster with a K8s cluster. The results showed better performance for K8s in terms of memory consumption and CPU usage. Another study of Kang et. al. [4] compared the performance of Docker Swarm and K8s in a limited computing environment on Raspberry Pi boards. They also concluded that K8s outperforms Docker Swarm if used with a high amount (=30) of service containers on 3 Pi boards [4]. Since they focused on container distribution and management methods this might get handy in the use case scenario under study.

3.1.1 Hyper-V Replication

Microsoft Windows supports a replication mechanism for virtual machines hosted by Hyper-V. The existing virtual machines are mirrored to secondary virtual machine host servers which highers scalability and reliability. The replications are replicated to a secondary Hyper-V host server, enabling process continuity and recovery on outages. Although there are benefits, like scalability and recovery, Hyper-V is mainly designed for virtual machines. Therefore, the cluster management solution is not applicable on this use case.

3.1.2 Docker Swarm

"Docker Swarm" is a cluster and orchestration engine for the container service "Docker". The offered extension mode has more features compared to the Hyper-V replication and is specialized for containers. For example, Load Balancing, increased fault tolerance and automatic service discovery. A highlighted feature among Docker Swarm is the decentralized design. That means, manager and application service can both run on any node within the cluster. Since it comes with Docker, no additional installation is required if Docker is already installed on the system. However, since it is bound to the Docker Application Programming Interface (API), using this orchestration technology involves the risk of inflexibility later on ("vendor lock-in").

3.1.3 Kubernetes

Kubernetes (K8s) is a orchestration engine similar to "Docker Swarm". Load balancing, auto-scaling and automatic service discovery are also offered. However, K8s additionally comes with the ability to rollback to a previous version in a product lifecycle and has built-in support for auto-scaling. However, K8s has more sophisticated configuration options which makes it harder to configure in the beginning.

3.2 Kubernetes

Since K8s is the chosen orchestration engine, the following sections are taking a deeper look inside its architecture.

3.2.1 Entities

There are many entities for objects inside the cluster. For description of those entities the configuration language YAML is used. Some of the most widely used entities are described in the following paragraphs.

Deployment Deployments are used to define declarative states for Pods. This allows to maintain consecutive versions of the pod and upgrade them during runtime.

Pod A pod represents a set of running containers on a node. Each pod has additional information stored, such as Health state, the cluster internal network Internet Protocol (IP) address or the amoun of replications.

Daemon set These ensure that multiple (or all) nodes run a certain pod[5]. Common use cases are tasks for all nodes or running the network overlay pod.

User This entity describes a user that can access the K8s cluster and API services. Users can be part of a group and permission roles.

Node A node represents a physical machine inside the cluster. Nodes can run multiple pods.

3.2.2 Services

K8s comes with a set of core services (see Figure 3.1) that ensure the life span of scheduled containers, and the application services that offer the actual application.

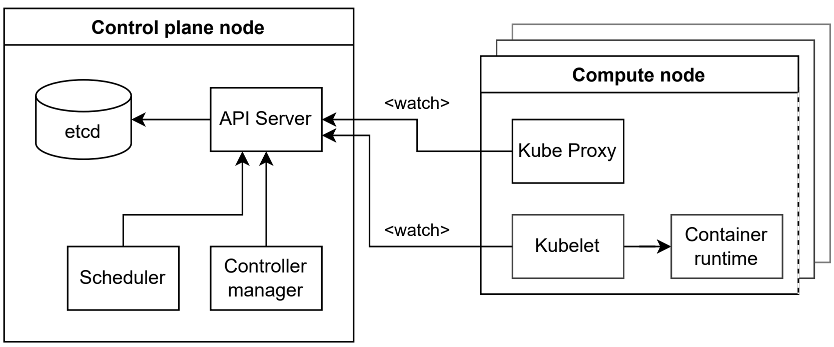

FIGURE 3.1: Core and compute services for Kubernetes[1]

In the following paragraphs, the crucial services are described in detail. Since every service is a pod, they can have multiple replicas. Only the core service have to run on a dedicated Linux node the so called ”control plane node”. The other services can run on nodes for executing the applications and perform computations (”compute node”).

etcd The etcd[1] database server is a key-value store designed for distributed systems[1]. That means it could run with multiple replications and would still be able to keep a persistent storage synchronized across multiple instances. It contains the applied configuration of several cluster entities (e.g. User configurations, deployments, pod configurations).

API server This is a RESTful web server that serves the Kubernetes API via Hypertext transfer protocol (HTTP)[6]. It is the central joint between the services and establishes communication between users, external components and other core services. It makes the objects stored in etcd accessible over an Open API specification[1, 7] and allows observing changes on the entities. The Command line interface (CLI) tools ”kubectl” and ”kubeadm” both interact with the API server.

Kubelet Kubelet is the service on the operating system level that maintains the pod life cycle and ensures the runtime of a container inside a pod. Furthermore, it manages the registration of the node to the control plane and reports its health and pod status to the API server.

Kube Proxy The Kube-Proxy runs as a separate pod on every compute node. It maintains the connectivity between the services and pods[1]. For a given IP address and port combination it assures the connection to the corresponding pod. If multiple pods can offer a service, the proxy also acts as a load balancer[1].

Scheduler The scheduler is responsible for distributing services on the cluster and determining which node to choose during runtime. It reads conditions for scheduling (e.g. hardware resources, operating system, labels) from the API server and decides which node matches the configuration[1].

[1] etcd: https://etcd.io/

Controller manager While the API-Server is responsible for storing data in etcd and announcing changes to the clients, the Controller manager and its parts try to achieve a described target state[1]. The controller manager consists of several controllers for replications, daemon sets, deployments, volumes, and so on.

3.2.3 Cluster networking

Cluster networking is achieved using two components: The network plugin and the Container Network Interface (CNI). Pods receive their own IP address and can communicate with other pods. However, this is not a functionality which is achieved by Kubernetes directly. By using a CNI the automated generation of network addresses and their inclusion is achieved when new containers are create or destroyed. It is crucial that pods share the same subnet across all the nodes in a cluster and Network Address Translation (NAT) is avoided[1].

Network plugins do implement the CNI. They usually come with a manifest for a daemon set that introduces a network agent on all nodes inside the cluster to support the network communication. For setting up the network interface, namespace and its IP address, a dedicated container image is used. This is called the ”pause container” image.

Even though, the team behind K8s do not recommend any specific network plugin, there are only a few common network plugins widely used. For this case, Flannel is used as network plugin, since it is the described plugin used in the documentations for setting up K8s with Windows containers[8, 9]. Hence, support for this CNI plugin in relation with Windows containers is larger.

Flannel Flannel[2] was originally developed as part from Fedora CoreOS[3][10]. It works with various backends for transferring packets in the internal network. Two possible backends are virtual extensible Local Area Network (”vxlan”) and host gateway (”host-gw”). While ”host-gw” needs an existing infrastructure and performs routing on the layer 3 network level, VXLAN is more flexible and could also be used in cloud environments[11]. VXLAN is an overlay protocol and encapsulates layer 2 Ethernet frames within datagrams[10]. It is similar to regular VLAN, but offers more than 4,096 network identifiers[10]. Thus, VXLAN is a good choice for highly scalable systems.

[2]Flannel: https://github.com/flannel-io/flannel
[3]CoreOS: https://getfedora.org/en/coreos

Calico Compared to Flannel, Calico[4] is stated to be more performative, flexible and powerful[10, 12]. Calico comes with a sophisticated access control system[12] and more configuration options. However, its advanced configuration makes it hard to maintain long-term.

3.3 Container environment

The ecosystem around containerization defines terminology that needs to be looked at before going into details for K8s. First of all, the Container Runtime Interface (CRI) defines the interface between K8s and container runtime. Most of the container runtimes follow the design principles defined by the Open Container Initiative (OCI)[5] for describing images and containers. The actual container runtime runs the isolation layer between the physical host machine and the K8s cluster by using containerization of processes. This is what can be selected when working with K8s.

While K8s used to support Docker as their standard container runtime, they announced it to be deprecated in 2020, and finally removed the support in February 2022[13, 14]. The teams behind K8s decided to drop the hard coded support for Docker and offer ContainerD instead. However, the specification for ContainerD's "Containerfile" has only minor differences compared to Docker's "Dockerfile". Thus, ContainerD files are fully compatible to docker files.

Some of the container runtimes offered by K8s are not available for Windows hosts. For example, Linux containers (LXC)[6] use process groups, control groups (cgroups) and name spaces on the operating system level. The CRI from the Open Container Initiative (CRI-O)[7] is another alternative offered for K8s on Linux systems. Since those are not available in Windows, they are not further considered.

At the current time being, container networking with ContainerD is not well-established on Windows [15–18] even though the docker runtime is already removed in current versions of K8s [13]. However, these are the only two working container backends for Windows containers. Therefore ContainerD as container backend was chosen.

[4]Tigera's Calico: https://www.tigera.io/project-calico/
[5]OCI: https://opencontainers.org/
[6]LXC: https://linuxcontainers.org/
[7]CRI-O: https://cri-o.io/

3.3.1 ContainerD

Containerd is a native version of a container runtime. Newer versions of Docker on Linux, are running Containerd under the hood for process isolation. On Windows ContainerD uses slim host process isolation. The process isolation with ContainerD consists of multiple abstraction layers (shown in Figure 3.2). The ContainerD back end contacts the containerd-shim which is maintaining an abstraction layer for communication for the underlying layers (depending on Linux and Windows). Below that, Windows offers a custom fork of the CLI *runc*, so called *runhcs*[2]. With *runc* new containers can be created by running a simple command[2]. The layer for *runhcs* connects to the Host Compute Service (HCS) which is another abstraction layer of Windows for providing a stable API to the low level functionality of the operating system[19].

Containerd does not come with any mechanisms for networking. Instead, this is in responsibility of the HCS. In current versions of K8s, ContainerD is the only available CRI for Windows containers.

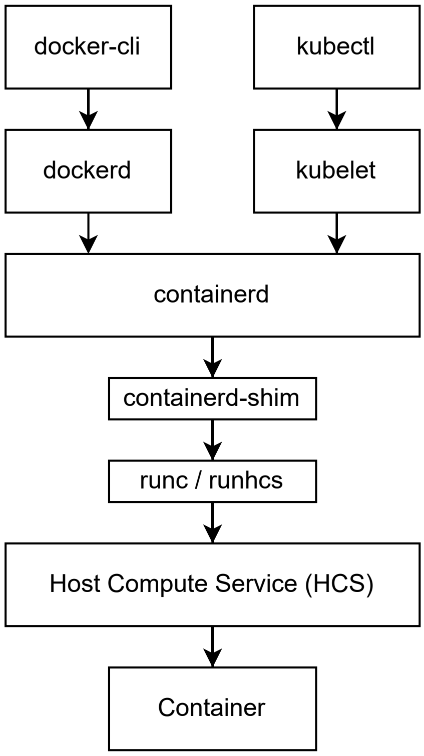

FIGURE 3.2: Abstraction layers for ContainerD on Windows. The image shows the technology stack from the Docker and K8s command line to the Container layer.[2]

3.3.2 Docker Container Runtime Interface

The Docker CRI (so called ”Docker shim”) is using the internal mechanisms from Docker to run containers. Older versions in Linux were using control group isolation.

For Windows, there are two different modes available. The first option is using the process isolation mode offered by ContainerD. It can be enabled by switching to ”Windows Containers”. It is the default mode for Windows Server systems. However, older versions of Docker and Docker on Windows 10 and above has the opposite behavior[20]. On client versions of Windows, a dedicated hypervisor-isolated virtualization is the default option[20]. This means, during the installation of Docker on Windows, the underlying Windows container host creates a separate Hyper-V Virtual Machine (VM) to run container images. However, this is not a regular Hyper-V VM. Instead, it is a purpose-built VM, often referred to as utility VM or UVM that can't be managed directly and is fully controlled by the Windows container runtime[20]. For networking, the internal mechanisms from Docker are used. All running containers are deployed inside the dedicated Hyper-V VM. Therefore, this is a mixture of process isolation and full isolation using virtualization.

However, the hypervisor approach still has the disadvantage of using large resources for containers, even though they are running in one virtual machine. In addition, containers running in hypervisor isolation take longer to start up than those running in process isolation[20].

3.4 Container Image

The container image consists of a base part and a part for custom configuration. Both are further explained in the following sections.

Container images are described, using the Containerfile[8] format. There are container images for each process of the system architecture, each of them having their own Containerfile definition with different command line arguments and environment variables.

[8] Containerfile: https://www.mankier.com/5/Containerfile

3.4.1 Base image

The container images to use for running the OpenTwin processes need to run a Windows base image. Beside the full Windows images, Microsoft® offers the more common images "Windows Server Core" and "Windows Nanoserver"[21]. They significantly differ in the download size, their on-disk footprint and the features supported[21]. As Microsoft states, "Nanoserver was built to provide just enough API surface to run apps that have a dependency on .NET core or other modern open source frameworks. PowerShell, Windows Management Instrumentation, and the Windows servicing stack are absent from the Nanoserver image"[21].

The Server Core image is not the smallest base image. However, for the current use case it is the smallest image with full functional support for the required technologies.

3.4.2 Custom image

On top of the base image, customizations and the actual application are applied. The binary files are included in the container image and added during build. The common CRIs only forward the output of processes with process id 1 to the host machine. Furthermore, this is also the only process the CRI is waiting for, to keep the container alive. Thus, instead of using the provided batch files to start the application services, the OpenTwin process is called directly with the appropriate command line arguments as command for the container. Therefore, the environment variables needs to be set up as part of the container file. The root certificate (certificate authority) is passed as file mount into the container later on.

3.5 Target architecture

The application needs to be distributed on multiple systems. Kubernetes supports application rollout only as container images. To be able to distribute the services on a cluster management tool a containerization of the application is necessary.

Chapter 4

Implementation

4.1 Containerization of Services

Container images are provided for running the application inside the CRI. Definition of the container manifest is done in the "Containerfile"[1] format is used. - Build has to be done inside the container. / All binary files are part of the container image.

4.2 Cluster Setup

The following section describes the setup of different machines in the cluster, so called nodes. While the master node refers to the K8s Control-Plane node which is responsible for distribution of the workers, the worker nodes are the actual machines that are executing the applications. During development the cluster was set up on virtual machines completely, due to the lack of physical hardware.

4.2.1 Creating the master node

For setting up the master node on Linux a system based on Debian Bullseye 11.5 has been used. After installing and setting up the operating system, the swap mechanism needs to be permanently turned off. This is done by editing the file system table (fstab) in file `/etc/fstab` respectively by commenting out the swap partitions and masking the systemd swap units.

[1]https://www.mankier.com/5/Containerfile

After installing the pre-requisite packages, a containerd config file needs to be created. For this, the command from Listing 4.1 is applied.

```
sudo sysctl net.bridge.bridge-nf-call-iptables=1
echo 1 > /proc/sys/net/ipv4/ip_forward
sudo containerd config default | sudo tee /etc/containerd/config.toml &>/dev/null
```

LISTING 4.1: Bash command for setting up containerd config

Afterwards the systemd cgroup is added to the runtime options of containerd and the its service is restarted. After setting up the prerequisites, the cluster can be initialized by running the command line tool as shown in Listing 4.2 with the appropriate configuration as parameter.

```
sudo kubeadm init --config config.yaml
```

LISTING 4.2: Bash command for setting up the cluster

4.2.1.1 Installing a Container Network Interface

After successfully running the initialization, the cluster overlay network `flannel` needs to be setup. This is required for working with Windows worker nodes. To setup `flannel` the respective pod description can be directly downloaded from the vendor[2]. In the configuration the VNI (4096) and port (4789) for Flannel on Windows were set. Afterwards the configuration has been applied on the cluster. After linking kubectl to the local control plane node, the successful setup of the cluster can be checked with the `kubectl` command.

4.2.1.2 Adding the proxy daemonsets

For using Windows worker nodes in a Kubernetes cluster, additional configmaps and daemonsets need to be applied on the cluster. Those are used for setting up a proxy for `flannel`.

```
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest \
  /download/kube-proxy.yml | sed 's/VERSION/v1.25.3/g' | kubectl apply -f -
kubectl apply -f https://github.com/kubernetes-sigs/sig-windows-tools/releases \
  /latest/download/flannel-overlay.yml
```

[2]https://raw.githubusercontent.com/flannel-io/flannel/master/Documentation/kube-flannel.yml

4.2.2 Creating the worker node

On the Windows worker node, the prerequisites were installed first. While installing the prerequisite `crictl` it was added to the PATH environment variable. After the setup, the node preparation scripts of the Kubernetes Special Interest Group (SIG) were retrieved. Before running the scripts, the CNI version needs to be aligned to the same version written on the master node. Therefore the appearances of v0.2.0 have been replaced with v0.3.0 respectively. Afterwards the installation script was executed. It sets up the NAT configuration on the worker node and registers containerd as a service. For having a valid image at a later point in time, the value sandbox_image in cri in containerd's configuration file (config.toml) needs to be replaced with a newer version.

After sucessfully setting up the NAT and installing containerd as a service, the node will be finally prepared to host tasks. For this, another powershell script "PrepareNode"[3] from the Kubernetes SIG was run. After running the script the resulting "StartKubelet" file needs to be changed to drop invalid arguments.

Furthermore, the following lines were added to the Kubelet configuration:

```
enforceNodeAllocatable: []
cgroupsPerQOS: false
enableDebuggingHandlers: true
```

Since the configuration changes needs to be served only to Windows machines (config value are valid for Windows only) we (!!WE!!) need to manually change the configuration on the nodes locally. This

After successful run of the preparation script the node was ready to join the cluster.

4.3 Automatic setup

[3]https://github.com/kubernetes-sigs/sig-windows-tools/releases/download/v0.1.5/PrepareNode.ps1

Chapter 5

Results

The development of the K8s cluster and the containerization of OT hides several pitfalls. These are discussed in this chapter.

5.1 Containerization

During the containerization of the application, the issues that require consideration are discussed in this section.

5.1.1 Container manifest

Even though the format of the container manifests ”Containerfile” is compatible to the proprieteary ”Dockerfile” format from Docker, the CRIs do not follow the specification everywhere[22] This is an issue while writing a Containerfile for the ContainerD CRI. Especially in cases where line breaks in the Containerfile might be necessary to shorten long lines and increase readability. ContainerD is treating line breaks paths in string notation different compared to paths in JSON array notation and is not following the specification[23].

```
ENTRYPOINT open_twin.exe \
Service.dll
```

LISTING 5.1: Containerfile entrypoint specification across multiple lines in text format.

Listing 5.1 shows an example of the problem. While the entry point in Docker is interpreted as *open_twin.exe Service.dll*, the interpreter in ContainerD only reads the first line as entry point and ignores the line break character ”äs in *open_twin.exe*. To overcome this flaw, the *ENTRYPOINT* definition has to be written in JSON notation. If defined as in Listing 5.2 the interpreter

```
ENTRYPOINT ["open_twin.exe",
"Service.dll"]
```

LISTING 5.2: Containerfile entrypoint specification across multiple lines in JSON format.

5.1.2 Windows Base Image

While it is normal to have a requirement for the same operating system kernel when running container images, it is a uncommon requirement to have the exact same build version.

- Windows base image has to have same build number than host computer (not the same in linux containers)

- OpenTwin needs to be compiled on the system it is running. Developers need to fix this issue.

5.2 Discovered issues and pitfalls

- K8s Documentation redirects to a different page when searching for tutorial for adding windows nodes - Error messages of Windows hcs shim are not explanatory

- Windows needs to have certain Update installed to run flannel container

- Scripts and docs are maintained by a relatively small community (SIG windows tools)

- Cluster networking throws weird error messages when performed inside hyper-v vm (”Directory not found”)

- Docker support was removed, containerd is not fully supported yet (without bugs) in Windows

Chapter 6

Discussion

6.1 Analysis

Chapter 7

Conclusion and future work

7.1 Future work

- Linux port and cluster based on linux - Automated image building using Packer.io (for multiple platforms) - Ranger for streamlining kubernetes deployment - Images verkleinern - nur die Dateien ins Image bundlen, die für den entsprechenden Service notwendig sind.

7.2 Conclusion

Appendix A

Appendix Title Here

Write your Appendix content here.

OT OpenTwin

Windows Microsoft® Windows®

cgroup control group

SIG Special Interest Group

DLL Dynamic Link LibraryDynamic Link Libraries

URL Uniform Resource Locator

UI User Interface

TLS Transfer Layer Security

mTLS mutual Transfer Layer Security

GSS Global Session Service

AUTH Authorization Service

LSS Local Session Service

HTTP Hypertext transfer protocol

API Application Programming Interface

CLI Command line interface

K8s Kubernetes

IP Internet Protocol

NAT Network Address Translation

CNI Container Network Interface

CRI Container Runtime Interface

VM Virtual Machine

HCS Host Compute Service

Bibliography

[1] Marko Lukša. *Kubernetes in action: Anwendungen in Kubernetes-Clustern bereitstellen und verwalten*. Hanser eLibrary. Hanser, München, 2018. ISBN 9783446456020. doi: 10.3139/9783446456020. URL https://www.hanser-elibrary.com/doi/book/10.3139/9783446456020.

[2] Scooley. Windows container platform: Microsoft learn, 2022. URL https://learn.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/containerd.

[3] Nikhil Marathe, Ankita Gandhi, and Jaimeel M. Shah. Docker swarm and kubernetes in cloud computing environment. In *Proceedings of the International Conference on Trends in Electronics and Informatics (ICOEI 2019)*, pages 179–184, Piscataway, NJ, 2019. IEEE. ISBN 978-1-5386-9439-8. doi: 10.1109/ICOEI.2019.8862654.

[4] Byungseok Kang, Jaeyeop Jeong, and Hyunseung Choo. Docker swarm and kubernetes containers for smart home gateway. *IT Professional*, 23(4):75–80, 2021. ISSN 1520-9202. doi: 10.1109/MITP.2020.3034116.

[5] Kubernetes. Daemonset: Kubernetes documentation, 2022-08-31. URL https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

[6] Kubernetes. The kubernetes api: Documentation, 2022-10-24. URL https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

[7] OpenAPI Initiative. Version 3.1.0 of the openapi-specification: Github - oai/openapi-specification, 2023-02-10. URL https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md.

[8] GitHub - Kubernetes SIG Windows Tools. Guide for adding windows node: Documentation, 2023-02-13. URL https://github.com/kubernetes-sigs/sig-windows-tools/blob/727707fa7d83d401956b467a5ce41700cce7d9a3/guides/guide-for-adding-windows-node.md.

[9] Kubernetes. Adding windows nodes: Kubernetes - documentation (archived / v1.23), 2022-04-19. URL https://v1-23.docs.kubernetes.io/docs/tasks/administer-cluster/kubeadm/adding-windows-nodes/.

[10] Suse Rancher Community. Comparing kubernetes cni providers: Flannel, calico, canal, and weave | suse communities, 2023-02-12. URL https://www.suse.com/c/rancher_blog/comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/

[11] GitHub - Flannel.io. Flannel backends documentation, 2023-02-12. URL https://github.com/flannel-io/flannel/blob/master/Documentation/backends.md.

[12] Tigera. Project calico | tigera, 2023-02-10. URL https://www.tigera.io/project-calico/.

[13] Kubernetes. Don't panic: Kubernetes and docker, 2020. URL https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/.

[14] Kubernetes. Kubernetes is moving on from dockershim: Commitments and next steps, 2022. URL https://kubernetes.io/blog/2022/01/07/kubernetes-is-moving-on-from-dockershim/.

[15] GitHub. Guide for adding windows node: Rbac config not found: Issue #261 - kubernetes-sigs/sig-windows-tools, 2023-02-02. URL https://github.com/kubernetes-sigs/sig-windows-tools/issues/261.

[16] GitHub. Windows node with containerd can't run flannel and kubeproxy daemonsets: Issue #128 - kubernetes-sigs/sig-windows-tools, 2023-02-02. URL https://github.com/kubernetes-sigs/sig-windows-tools/issues/128.

[17] amaltinsky. Windows containers issue: Windows server 2022 container on windows 10 (10.0.19044) host., 2022. URL https://github.com/microsoft/Windows-Containers/issues/258.

[18] GitHub. kube-flannel for linux fails to (re)start after rbac for kube-flannel for windows is applied: Issue #277 - kubernetes-sigs/sig-windows-tools, 2023-02-02. URL https://github.com/kubernetes-sigs/sig-windows-tools/issues/277.

[19] Microsoft. Introducing the host compute service (hcs): Blog post, archived technet article, 2017. URL https://techcommunity.microsoft.com/t5/containers/introducing-the-host-compute-service-hcs/ba-p/382332.

[20] Vinicius Ramos Apolinario. *Windows Containers for IT Pros: Transitioning Existing Applications to Containers for on-Premises, Cloud, or Hybrid*. Apress L. P, Berkeley, CA, 2021. ISBN 9781484266861. URL https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=6508410.

[21] Mattbriggs - Microsoft. Windows container base images: Microsoft documentation, 2023-02-14. URL https://learn.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/container-base-images.

[22] failed to create containerd task: hcsshim::createcomputesystem kube-proxy: The directory name is invalid. URL https://stackoverflow.com/questions/74799620/kubernetes-windows-worker-node-addition-failed-to-create-container

[23] ManKier. Containerfile format: automate the steps of creating a container image, 2023-02-22. URL https://www.mankier.com/5/Containerfile.