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

"Obstacles don't have to stop you. If you run into a wall, don't turn around and give up. Figure out how to climb it, go through it, or work around it."

Michael Jordan

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

This thesis explores the feasibility of deploying a Microsoft® Windows® (Windows) based microservice application using Kubernetes (K8s). It is showcased on the basis of OpenTwin (OT), which is a application for computer aided design and physics simulation. OT is a distributed application where the computations can be performed server-side, while the operation by the user stays client-side. It turns out, that deploying the application is challenging. Some changes on the application itself have to be done, especially in respect of the mutual authentication that comes to use. Further, finding information for setting up Windows based K8s clusters is not as easy as it seems. A problem was the ongoing switch from Hyper-V isolation to process isolation, because the new technology is not yet supported throughout the application landscape of K8s. Thus, the thesis provides value for a rather unexplored field, because the usage of Windows containers for deployment is not well explored yet.

Acknowledgements

First of all, I wish to thank my supervisors. Prof. Dr. Peter Thoma provided outstanding support for writing the thesis. He guided my work and helped to keep the motivation high. Prof. Dr. Eicke Godehardt taught me the fundamentals of Cloud Computing and Software Engineering that were required to have a basic understanding of the topic.

I am grateful to my friends and fellow students who proofread the thesis. I would especially like to thank Johannes Krafft and Lucas Sambale, who both gave me qualitative thorough feedback.

Lastly, I wish to thank my supportive family. They have been with me throughout my studies and have given me the support I needed. Without their love and guidance I would not be where I am today.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

The containerization of applications became more and more popular in the recent decade. Running applications in an isolated environment makes them more easy to manage and easier to deploy. The lightweight method of isolating processes is the basis for deploying applications on a cluster environment. Deployed applications on distributed systems can nowadays profit from more reliability and availability. Additionally, the services can gain a higher performance due to horizontal scaling mechanisms. This means, instead of increasing performance by upgrading the hardware, the services are accessible on multiple servers. By having a microservice architecture, complex systems are well prepared for a distributed system. The several services of the application are responsible for just their own task and thus can be scaled horizontally.

This thesis explores the feasibility of deploying a Microsoft® Windows® (Windows) based microservice application on a cluster and gives an overview about the faced challenges. The feasibility is showcased on the basis of OpenTwin (OT). OT is a distributed application for computer aided design and physics simulation. The application is distributed, thus computations can be done server-side, while the operation by the user stays client-side. Accordingly, the orchestration has to take the multi-level architecture into account. This means that some services should only accessible in a private network space and do not require a public Internet Protocol (IP) address, while others have to be publicly available. Further, the communication has to be achieved bidirectional. Considering that clients are often protected by a firewall and located behind Network Address Translation (NAT), this is another challenge to face. Additionally, the architecture of OT is adaptive by automatically spawning new processes where they are needed. Hence the application is performing an implicit load balancing based on the system resources. Because security requirements have to be met, the network communication of OT is encrypted with mutual authentication. Using the two-sided encryption ensures authenticity on both sides, the client and the server. The deployment has to take this into account, by distributing and creating certificates for each service and sharing them across the cluster.

The majority of deployments is still performed on Linux, even though applications orchestration for Windows is gaining more and more demand. Recently Microsoft introduced with "Windows Containers" its own container technology[11]. Since the common orchestration Kubernetes (K8s) dropped the support for Docker containers, the requested container technology on Windows shifted towards Windows Containers[12]. OT is based on Windows, and thus the orchestration and containerization has to be aligned towards Windows Containers. Hence this is a fundamental task of this thesis.

Because of the recent change in isolation techniques, the previous research in this field is sparse. The documentations and guides are mostly based on the containerization using Docker and do not take the problems of the native container technology into account. Thus, the thesis aims to provide a valuable resource for research groups and developers who want to deploy applications using the modern Windows Container technology.

Another aspect is the introduction of a cluster using Windows as compute nodes. Finding an orchestration engine with full support for Windows is difficult. As shown in this thesis, K8s as chosen orchestration engine also has struggles adding Windows nodes into the cluster.

The next chapter (chapter 2) of the thesis provides a brief introduction into OT and its architecture. Furthermore, it introduces its encryption protocol and gives an outline about the overall topic. In chapter 3, the major design decisions are explained. It dives into the used container engine and the chosen cluster management software and their alternatives. The design decisions are implemented in chapter 4. This is where the application is adapted by the required changes. The application is containerized, and the cluster is set up. Moreover, the probed options for designing the cluster are presented and the deployment of an application is shown. The findings are discussed in chapter 5. A deeper look is taken in challenges that occurred during the implementation and the underlying reasons. The thesis ends with chapter 6 and gives an overview about the conclusion and future work.

Chapter 2

Background and problem statement

2.1 Application under study

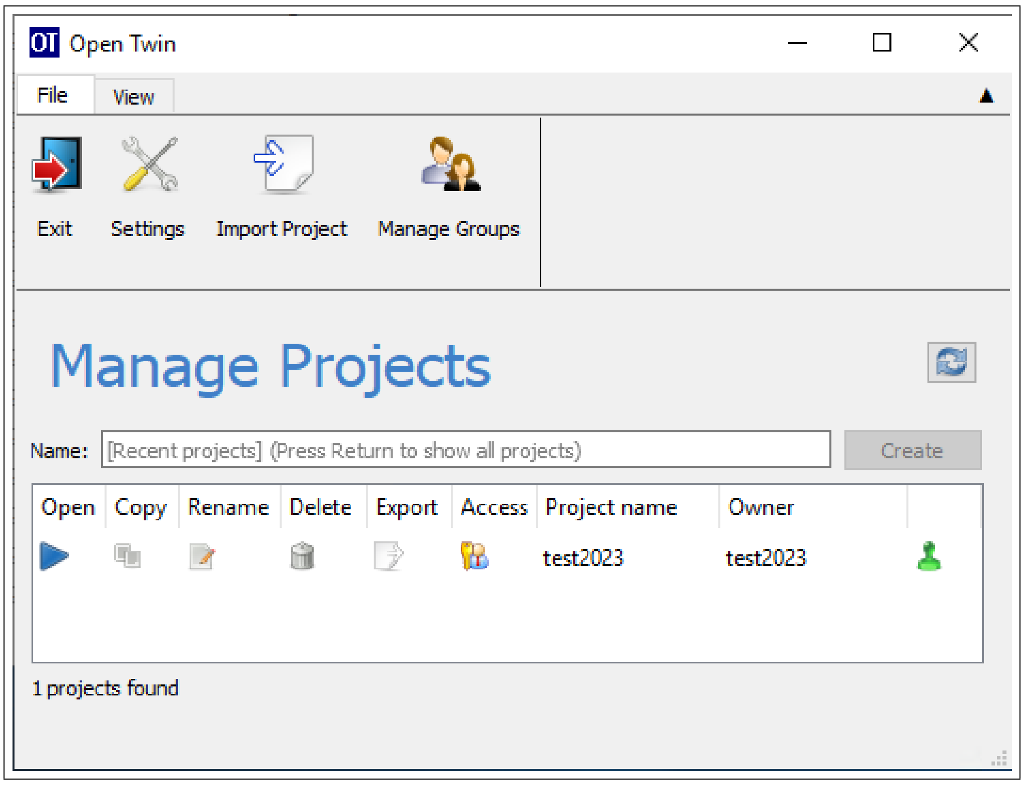
OpenTwin (OT) is an open-source simulation platform developed by the university of Applied Sciences in Frankfurt, Germany. It covers features like computer aided design and meshing and is also a physics simulation (having solvers for Finite Integration and PHREEC). The projects can be administered by 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 User Interface (UI) information is sent back to the client application. This has the benefit, that also underpowered computers can run the application. Further details about the network protocol are explained later in this section.

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



Manage Projects

Name: [Recent projects] (Press Return to show all projects) Create

Open	Copy	Rename	Delete	Export	Access	Project name	Owner	
						test2023	test2023	

1 projects found

FIGURE 2.1: The OT project overview.

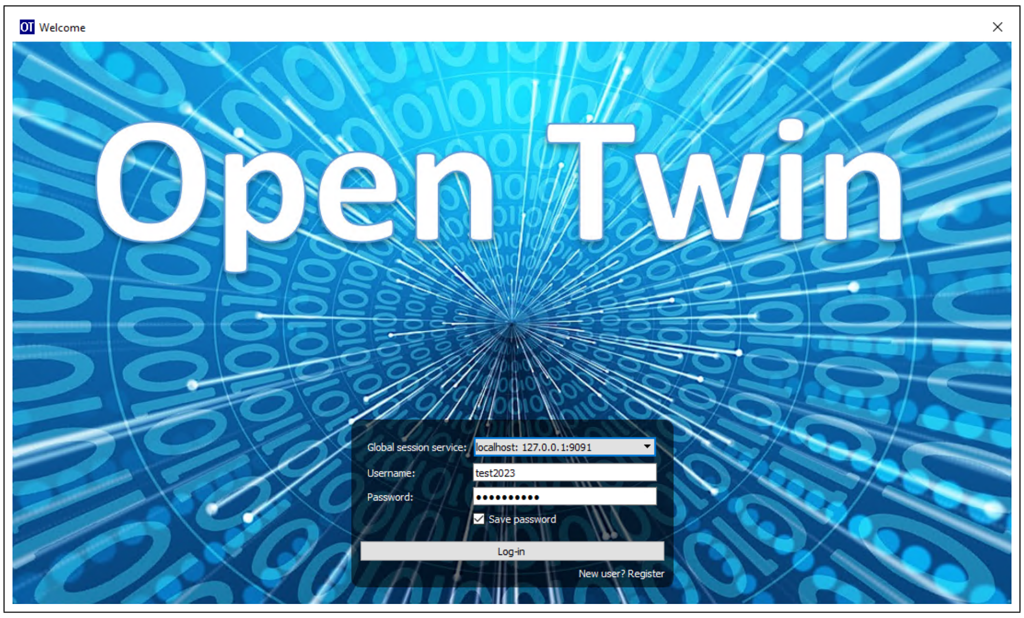
on Microsoft Windows. A port to Unix based systems is currently in work. Therefore parts of the code base are aligned for multiple system architectures already, but the application is not yet able to be compiled for Linux.

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 the server's binding address, port numbers, and 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").

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

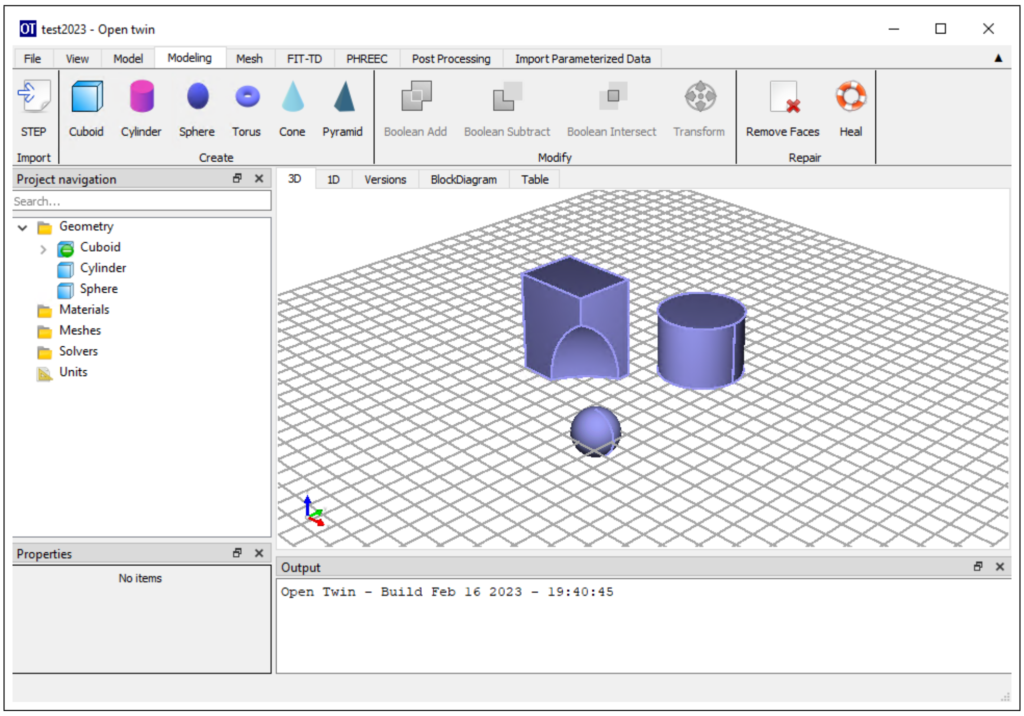
LISTING 2.1: Command line of Open Twin Service start

For conveniently running the services with all their necessary arguments, batch files are 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



Global session service: localhost: 127.0.0.1:9091
Username: test2023
Password: ••••••••••
☑ Save password

FIGURE 2.2: The OT login screen.



```
Open Twin - Build Feb 16 2023 - 19:40:45
```

FIGURE 2.3: An opened project inside OT with a few created geometric models and a boolean operation.

the environment for the network binding details, path to certificates and encrypted database credentials.

The system consists of the following microservices 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 a project that can dynamically spawn and exit. A partial list of compute services and their corresponding tasks can be found in Table 2.1. Each service runs in its own Operating System (OS) 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 transle electromagnetic simulation based on the finite integration technique (FIT).
KrigingService	If demanded, it runs a kriging interpolation of result data.
LoggerService	A background service, that accepts logging messages from other services.
ModelingService	Performs calculations for the creation, modeling and boolean combination of geometric data.
PHREECService	If demanded, it runs simulation based on PHREEC.
TetMeshService	If demanded, it meshes a form with an tetrahedral mesh.
VisualizationService	Runs the graphical calculations for displaying the geometric and data based results on the UI.

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

The Visualization Service requires features from version 2.0 of the graphics library OpenGL[2]. Systems where no graphics card is installed or systems that run a server OS do not have OpenGL 2.0 available. For these environments the DLL file “opengl32sw.dll” in the installation directory can be renamed to “opengl32.dll”. This enables software rendering and allows system to run OT that do not meet this requirement.

As shown in Figure 2.4, the services can be separated by their network space on multiple levels. 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 WebSocket[13] communication channel.

[1]MongoDB: https://www.mongodb.com/
[2]OpenGL: https://opengl.org/

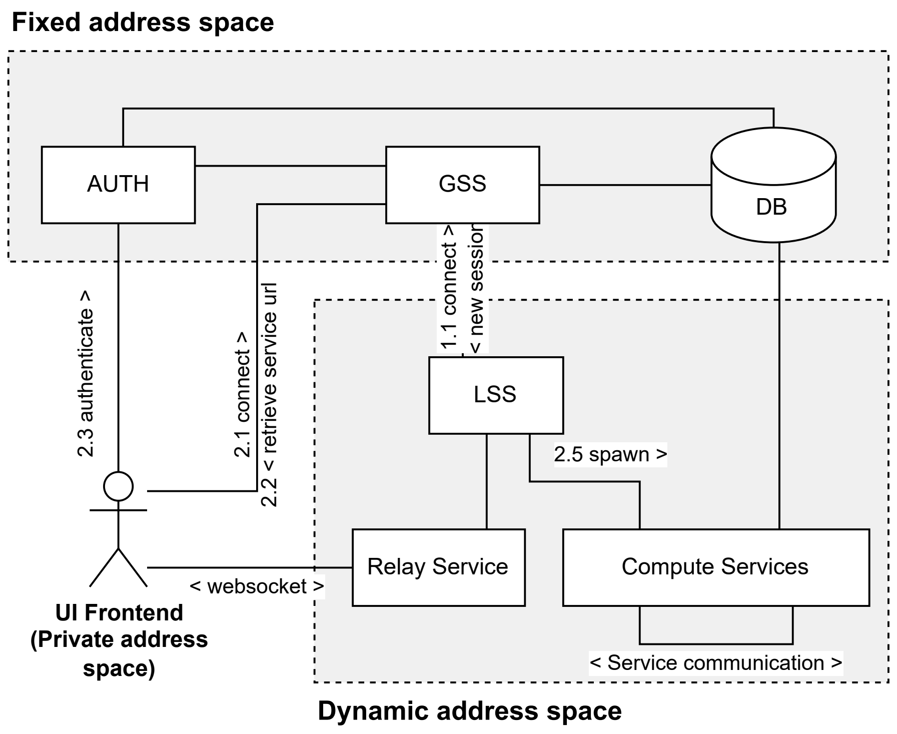


FIGURE 2.4: Communication overview and service organization for OT main services. In 1.1 the LSS registers at GSS. As soon as the UI front end connects to the GSS (2.1), service information is exchanged (2.2) and the user is authenticated (2.3). As a consequence, the GSS creates a new session and tells 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 WebSocket connection.

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 responds 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.

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 of 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 WebSocket.

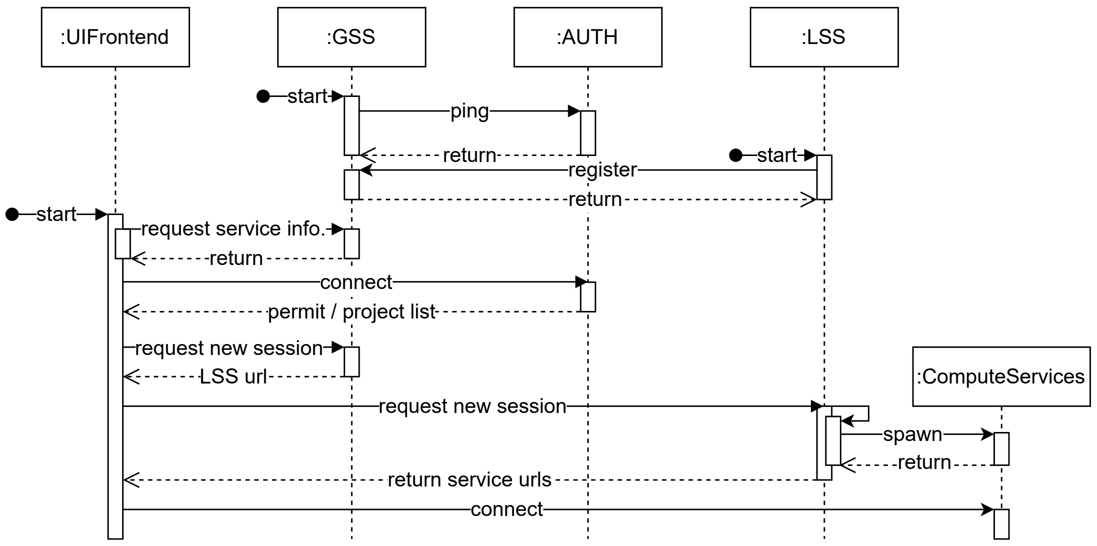


Figure 2.5: Service initialization of OT processes. In the beginning, the main services GSS, AUTH and an optional LSS are initialized. While the GSS checks the reachability of AUTH, the LSS registers itself at the GSS. After starting the UI front end, the service information is requested from a GSS and the user is authenticated. Afterwards, the project list for the authenticated user is displayed. After opening a project, 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 Network traffic encryption

Each service of OT offers two different channels for secure communication between services. One channel supports traditional one-way Transfer Layer Security (TLS), while the other uses bidirectional mutual Transfer Layer Security (mTLS). The one-way TLS channel is mostly used for checking the general health state of a service, while the two-way mTLS is used for relevant application based communication. In this section both encryption methods are briefly presented.

2.3.1 Transport Layer Security

TLS is an cryptography extension mainly designed for providing security over Hypertext transfer protocol (HTTP). The main goals of cryptography are confidentiality, integrity and authenticity between two communicating parties. This means, the communication on a network is kept secret between the two endpoints (Confidentiality), messages are not subject of manipulation (Integrity), and message exchanges are only allowed between authorized and trusted individuals (Authenticity). TLS ensures the three traits by using certificates.

A simplified handshake of the TLS protocol is depicted in Figure 2.6. After sending the certificate from server to the client, the client validates the certificate based on the chain of trust. This means, it checks if the certificate was issued by a trusted root authority (so called Certificate Authority (CA)). Only if the authenticity of the server is ensured, the encryption key is exchanged in order to start an encrypted communication.

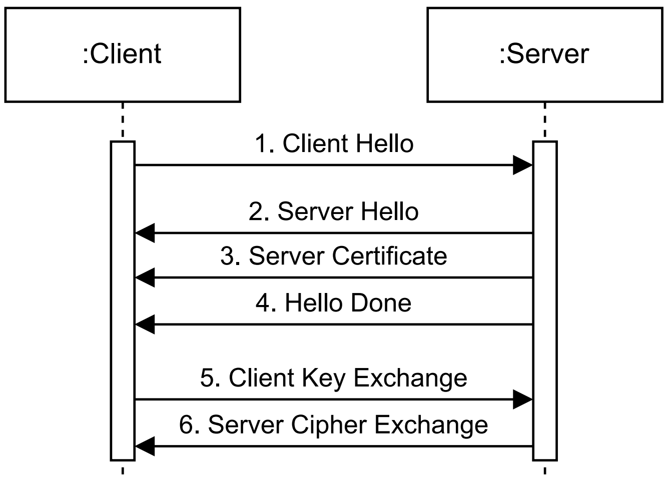


FIGURE 2.6: Simplified visualization of the TLS handshake[1]. After initialization of the handshake (1,2), the server sends its certificate (3) and finishes with a message “Hello Done” (4). The client then validates the certificate and compares it against the chain of trust. Afterwards, the key exchange starts (5,6) to ensure an encrypted communication.

2.3.2 Mutual authentication

The mutual authentication is adding another step to the one-way authentication. The application of it is used in mTLS as extension of the classic TLS protocol[1, 14]. Instead of just sending the server’s certificate to the client, the client also sends a certificate to the server. Therefore, not only the authenticity of the server is ensured, but also of the client.

As can be seen in Figure 2.7, compared to the TLS handshake, the mTLS handshake involves additional messages 5 and 6 for sending and validating the client's certificate. Unlike with the server certificate, the client certificate is not validated against a publicly available root authority[1, 2]. Instead, the server acts as root authority, creates the client certificate and ships it with the application[2].

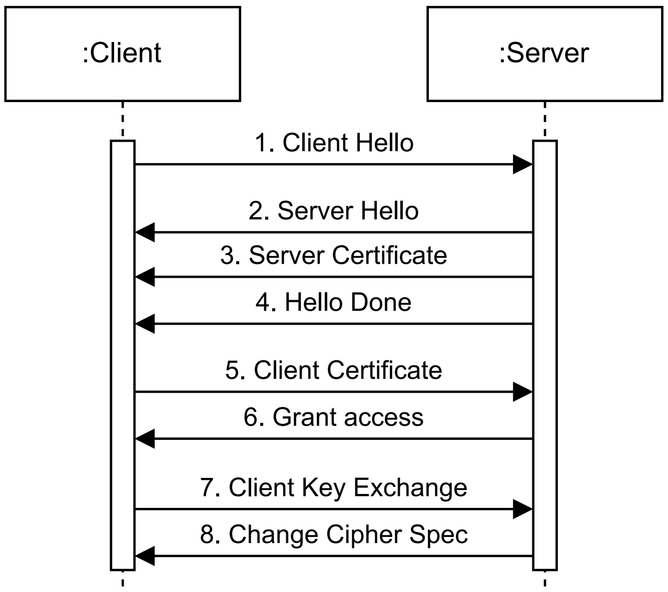


FIGURE 2.7: Simplified visualization of the mTLS handshake[1, 2]. After initialization of the handshake (1,2), the server sends its certificate (3) and finishes with a message "Hello Done" (4). The client then validates the certificate and compares it against the locally stored root authority. Afterwards, the client sends its certificate to the server (5). The server validates the client certificate against its local root authority and grants access to the service (6). Afterwards, the key exchange starts (7,8) to ensure an encrypted communication.

2.3.3 Certificate creation

For creation of certificates in the application landscape of OT, CloudFlare's public key infrastructure toolkit "cfssl"[3] is used. For generating certificates with the toolkit, it is fed with a JSON file with subject information for the Certiticate Request (CSR).

It contains information about the issuer, as well as the name and cryptography algorithm of the certificate. Additionally, the client and server certificate that derive from the root CA contain information for whitelisted hosts in their CSR JSON file. If a host is not mentioned in the resulting certificate, requests from the corresponding host are rejected. Listing 2.2 shows such a configuration with accepted host names in the form of a CSR.

[3]cfssl: https://cfssl.org/ or https://github.com/cloudflare/cfssl

```
{
  "CN": "OpenTwin",
  "hosts": ["localhost", "127.0.0.1"],
  "key": {
    "algo": "rsa",
    "size": 4096},
  "names": [{
    "O": "Frankfurt University of Applied Sciences"
  }]
}
```

LISTING 2.2: Example of meta data in form of CSR configuration. “CN” is the certificate name. “hosts” describes the accepted hostnames, “key” describes information about the cryptography algorithm, “names” contains meta data of the organization

2.4 Containers and Virtualization

In general, there are two different options for isolating applications in cloud environments. These are Virtualization and Containerization. Both are briefly presented in this section.

2.4.1 Virtualization

Virtualization (or Virtual Machines (VMs)) allows the software simulation of physical hardware to an OSs. The real hardware is faked to the OS kernel and the VM can, under optimal conditions, not distinguish between whether it is running on real hardware or if it runs in a simulation. Because VMs just run as a software on a physical machine (the so called “host computer”), it provides the possibility for operating multiple computers on one physical machine. The fundamental architecture of the virtualization technology is shown in Figure 2.8.

Besides isolating the OS kernel and all of its processes, the network layer also has to be isolated. This is usually achieved by three different approaches:

- **Network address translation** One approach is to convert the network packets between the VM and the host computer based on NAT. This means, the host computer provides a network interface which is connected to the VM. Incoming and outgoing traffic to the VM is translated using the additional virtual network interface.

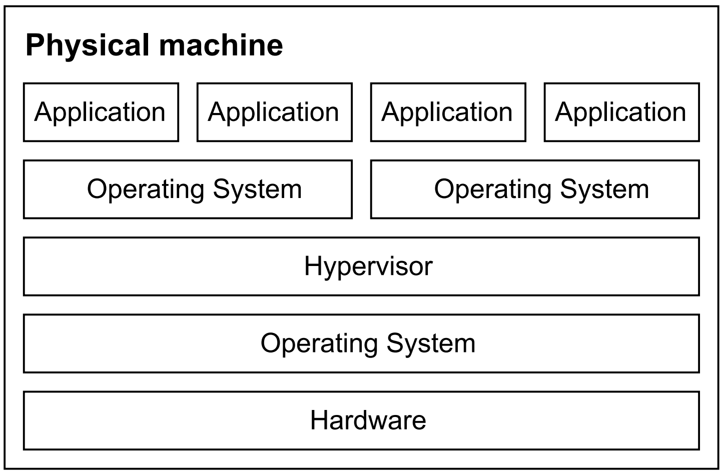


FIGURE 2.8: Brief presentation of the virtualization technology[3, 4]. Multiple OSs run on the same physical machine simultaneously. The hardware is mocked by the hypervisor layer.

- **Bridged network** In this approach the traffic between VM and host computer is "bridged". This means, the VM gets its own IP address assigned and is seen and acts as its own member.
- **Host network** The host network approach hides the VM from the outer network and only allows communication between the VM and the host machine.

Microsoft offers with Hyper-V its own virtualization software since Windows 8[15].

2.4.2 Container

As shown in Figure 2.9, Container have a more lightweight concept of isolation compared to virtualization. Instead of providing an isolation layer for both, the OS kernel and the running processes, only the processes are isolated ("process isolation"). Compared to virtualization, this increases performance of the isolated processes and reduces the startup time.

Containers share the OS kernel with the host system. Thus, OS kernel and processes running inside the container have to match. However, this does allow libraries to be shared within containers where necessary[4]. The process isolation is performed by encapsulating the file system, the users, the kernel processes and the network interfaces of the host system towards the container. Each container receives its own set of those encapsulated core objects. For the isolation of the network the same strategies as for virtualization apply.

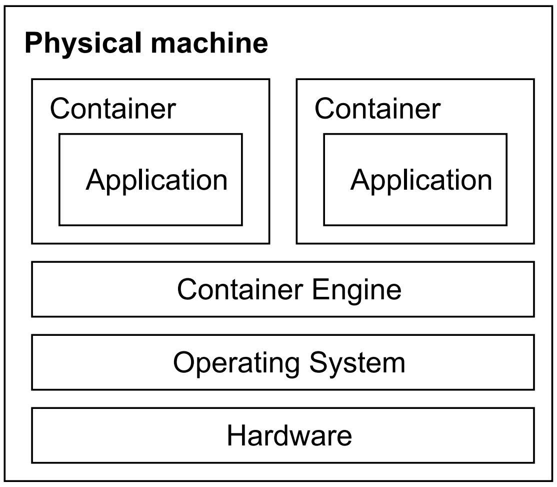


FIGURE 2.9: Brief presentation of the containerization technology[3, 4]. Each isolated process runs inside a container, the OS kernel is shared.

Because of their lightweight nature, containers usually only encapsulate a single application process. For encapsulating an application, one has to build a "container image". For the description of an image, the Containerfile format[4] is mostly used. The preparation of an image as "infrastructure-as-code" allows to build a reproducible environment on different machines. An example of a Containerfile is given in Listing 2.3.

```
1  FROM mcr.microsoft.com/windows/server:ltsc2022
2  COPY file.txt C:\out.txt
3  CMD type C:\out.txt
```

LISTING 2.3: Example of a container file. A file (file.txt) is copied to a Windows Server container (to C:\out.txt), and this file is printed during the container run using the command.

Container images can derive from other images that can derive from images again, and so on. Thus, an image definition consist of a general part as baseline (the "base image") and an application specific part. On top of this chain is an image for containerizing the OS processes. From which base image a container image derives is defined by the "FROM" statement (line 1). The value after "FROM" defines the address to the base image, whereas the value "ltsc2022" is called "image tag" and usually used for defining a concrete version of the base image. In this case, this is "Windows Server 2022".

Other statements can be given, for example, for copying files into the container file system (line 2), running commands for preparing the image or defining meta data about the image. The statement "CMD" (line 3) defines the command that is automatically run and monitored,

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

once the container image has started. If this command exits, the container is also automatically terminated.

If images are not available on a host system, they are "pulled" from a "container registry". It is a server that hosts the images, and is available either public or private.

2.5 Problem statement

Even though the application is clearly based on a microservice architecture and is able to run on a distributed system, it is not designed for an automated cluster yet.

Containerization of the system has never been tested and needs to be introduced. This is why one focus lies on the containerization of the application. Due to its complexity with mTLS encrypted communication and its sophisticated behavior with spawning new processes on demand, the containerization on Windows based system is challenging. Moreover, virtualization on Windows has shifted more and more towards containerization in recent years. Nevertheless, true process isolation is still a rather rarely used technology under Windows. This circumstance makes research and development of Windows containers quite complex and therefore containerization of the application must be investigated first.

As prerequisite, the application needs to be adapted. Error handling and logging is not present in the current application. Even though the front end application does write logs, the microservices currently do not produce log files. Instead, only a few sub processes write the information on their 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. Therefore, for simplifying development of the container image, logging must be introduced and proper error handling should be optimized.

The second focus lies on the setup of the cluster with regards to Windows nodes. First, the cluster engine needs to be set up. Since clustering with Windows containers is uncommon, preliminary studies where Windows nodes are part of a cluster are rare. Some cluster designs should be probed to provide a qualitative statement about which configuration is feasible. After adding the Windows node to the cluster, the network interface has to be set up to provide connectivity between the services and nodes. The created containerized application must be provided to the cluster engine so that it can be rolled out. Thus, automatic deployment of the developed images in the cluster engine has to be investigated. The functional connectivity of the services has to be ensured by checking the connectivity with regards to the mTLS encryption.

2.6 Limitations

The time frame for performing the task is limited. This is why not all code changes are applied. This involves the adaption for automatic extension of services, but also 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.

This thesis aims to provide a proof of concept for shifting an existing application to the cluster. Since the previous research in the field of host-process container applications is rather limited, the focus is on the investigation of using Windows nodes as part of the cluster. This is why the application is not fully containerized. The main services are containerized and the cluster is set up to investigate the behavior in cluster environments. Rather than providing a complete step-by-step guide, it highlights the challenges of developing deployment automation for Windows.

Chapter 3

System design

Various applications for realizing the architecture have been compared. These are mainly the orchestration engine and the container runtime. Additionally, the network driver is chosen. In the following sections the different options that were taken into account are presented and explained in detail.

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 for overloaded systems. Mirroring on multiple hosts can ensure fail-safety in the case one host goes down. 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.

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. Therefore, by replicating to a secondary Hyper-V host server, enabling process continuity and recovery on outages is ensured. Although there are benefits, like scalability and recovery, Hyper-V is mainly designed for virtual machines. Therefore, the cluster management solution is not applicable for 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 an orchestration engine similar to “Docker Swarm”. Load balancing, autoscaling and automatic service discovery are also offered. However, K8s additionally comes with the ability to rollback to a previous version in a product life cycle and has built-in support for auto-scaling. However, K8s has more sophisticated configuration options which makes it more complex to configure in the beginning. Because K8s clusters are usually hosted on premise, the risk of a vendor lock-in is mostly eliminated.

The engine of choice was K8s because of its rich feature set. Studies also showed that K8s outperforms Docker Swarms when it comes to performance. For example, Marathe et. al. [16] 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. [17] 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 [17]. Since they focused on container distribution and management methods this might get handy in the use case scenario investigated here.

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

K8s offers many entities for management objects inside the cluster. For description of those entities the configuration language YAML[1] is used. Some of the most widely used entities are described in the following paragraphs.

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 IP address or the amount of replications.

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

Daemon set These ensure that multiple (or all) nodes run a certain pod[18]. Common use cases are tasks for all nodes or running the network overlay pod.

Configuration Map A configuration map stores non-confidential data as key-value pair[19]. Data stored in a configuration map can be mounted as volume, environment variable or command line argument to make applications more portable[19].

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.

[1]YAML: https://yaml.org/

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 compute services that offer the actual application.

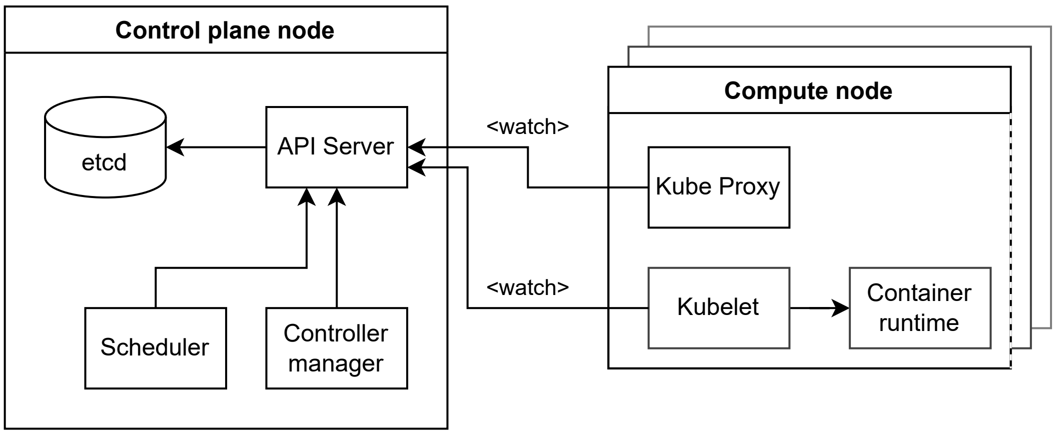


FIGURE 3.1: Core and compute services for Kubernetes[5]

In the following paragraphs, the crucial services are described in detail. The so called "control plane node" or "master node" is the node where core services of K8s are located. The underlying OS must be Linux, because Windows is not supported for the control plane. The other services can run on nodes (with any OS) for executing the applications and perform computations ("compute node" or "worker node"). Since every service is a pod, they can have multiple replicas.

The services are controlled by the Command line interface (CLI) tools "kubectl" and "kubeadm". While "kubectl" controls the deployment of services and application on the cluster, "kubeadm" is for setting up the cluster initially. Furthermore, the tool "crictl" can be installed separately for the inspection of the local Container Runtime Interface (CRI) on a node. It allows insights into the containers and images, but also into locally running pods. Additionally, pods and containers can be managed using this tool.

etcd The etcd[2] database server is a key-value store designed for distributed systems[5]. 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).

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

API server This is a RESTful web server that serves the K8s API via HTTP[20]. 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 through an Open API specification[5, 21] and allows observing changes on the entities. The CLI tools "kubectl" and "kubeadm" both interact with the API server.

Kubelet Kubelet is the service on the OS 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[5]. 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[5].

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, OS, labels) from the API server and decides which node matches the configuration[5].

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[5]. The controller manager consists of several controllers for replications, daemon sets, deployments, volumes, and so on.

3.2.3 Pod life cycle

Similar to the underlying application container, Pods in K8s have a ephemeral lifetime[8]. After creation on the cluster, a unique identifier is assigned before a pod gets scheduled to an available node[8]. The pod is kept alive until its termination or deletion[8]. For distinguishing different kind of states of a pod life cycle, K8s defines the pod states as described in Table 3.1.

A terminated pod automatically gets restarted based on a configured restart policy. As the K8s documentation states, "the kubelet restarts them [the containers of a pod] with an exponential

State	**Description**
Pending	The pod has been set up, the container and pod is currently initialized.
Running	The pod is bound to a node, the container is running.
Succeeded	The container terminated with a zero exit code.
Failed	The container terminated with a non-zero exit code or was terminated by the system.
Unknown	The pod state could not be obtained.

TABLE 3.1: List of K8s states during pod life cycle[8].

back-off delay (10s, 20s, 40s, ...), that is capped at five minutes"[8]. Furthermore, it is explained that the back-off time gets reset, once a container keeps running for 10 minutes[8].

3.2.4 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 K8s directly. By using a CNI the automated generation of network addresses and their inclusion is achieved when new containers are created or destroyed. It is crucial that pods share the same subnet across all the nodes in a cluster and NAT is avoided[5].

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.

Flannel Flannel[3] was originally developed as part from Fedora CoreOS[4][22]. 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[23]. As shown in Figure 3.2, VxLAN is an overlay protocol and encapsulates layer 2 Ethernet frames within datagrams[22]. On top of the physical (underlay) network, the VxLAN network provides their own IP subnet.

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

It is similar to regular VLAN and offers more than 4,096 network identifiers[22]. Thus, VxLAN is a good choice for highly scalable systems.

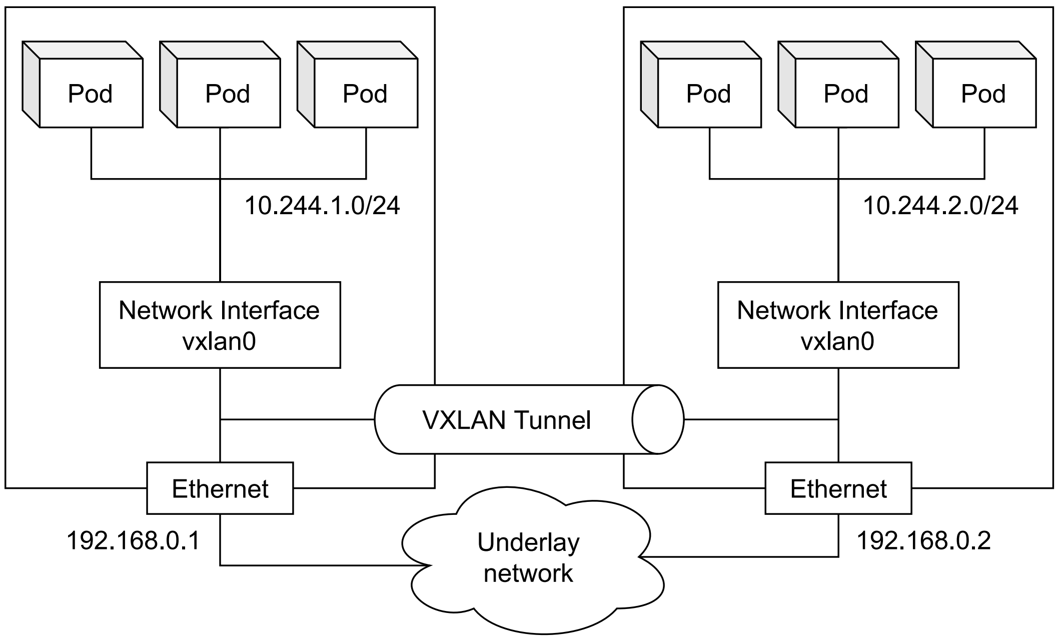


FIGURE 3.2: Flannel overlay network architecture for K8s[6]. The pods have their own IP address range and traffic gets routed through a VxLAN tunnel.

Although the team behind K8s does not recommend any specific network plugin, there are only a few common network plugins widely used. The amount of available CNI plugins is even more reduced if the support for Windows nodes is taken into account.

Calico Compared to Flannel, Calico[5] is stated to be more performative, flexible and powerful[22, 24]. Calico comes with a sophisticated access control system[24] and more configuration options. However, its advanced configuration makes it more complex to maintain long-term.

For this use case, Flannel is used as network plugin, since it is the more prominently used plugin used in the documentations for setting up K8s with Windows containers[10, 25]. Hence, support for this CNI plugin in relation with Windows containers is assumed to be larger than with Calico.

[5]Tigera's Calico: https://www.tigera.io/project-calico/

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 defines the isolation layer between the physical host machine and the K8s cluster. 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)[6] for describing images and containers. The container runtimes that are supported by K8s are presented in this section.

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[12, 26]. 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)[7] use process groups, control groups (cgroups) and name spaces on the OS level. The CRI from the Open Container Initiative (CRI-O)[8] is another alternative offered for K8s on Linux systems. Since those are not available in Windows, they are not further considered.

3.3.1 Docker Container Runtime Interface

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

For Windows, there are two different isolation modes available. The first option is using the process isolation (so called "Host 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 have the opposite behavior[3]. On client versions of Windows, a dedicated hypervisor-isolated virtualization (so called "Hyper-V isolation") is the default option[3]. This means, during the installation of Docker on Windows, the underlying Windows container host creates a separate

[6]OCI: https://opencontainers.org/
[7]LXC: https://linuxcontainers.org/
[8]CRI-O: https://cri-o.io/

Hyper-V VM to run container images. However, this is not a regular Hyper-V VM[3]. 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[3]. 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[3].

3.3.2 ContainerD

ContainerD is a native version of a container runtime. Newer versions of Docker on Linux are running ContainerD as a basis for process isolation. On Windows, ContainerD uses slim host process isolation. The process isolation with ContainerD consists of multiple abstraction layers (shown in Figure 3.3). Its 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*[7]. Using *runc*, new containers can be created by running a simple command[7]. 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 OS[27]. ContainerD does not come with any mechanisms for networking. Instead, this is in responsibility of the HCS.

The developers of K8s marked the Docker CRI as deprecated in version v1.20[12]. Since version v1.23 of K8s, Docker was fully removed which lead to ContainerD being the only available CRI for Windows containers.

There are two CLI tools that work with ContainerD. The tool "ctr" is the low level CLI tool that is shipped with ContainerD. "nerdctl" is a tool that is similar to the Docker CLI and supports the same set of commands and options as Docker. Compared to "ctr", however, "nerdctl" is more feature-rich.

Currently, container networking with ContainerD is not well-established on Windows [28–31] even though the docker runtime is already removed in current versions of K8s [12]. However, these are the only two working container backends for Windows containers. Therefore ContainerD as container backend was chosen.

3.4 Container Image

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. The container image and its base image is further explained in this section.

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”[32]. They significantly differ in the download size, their on-disk footprint and the features supported[32]. As Microsoft states, “Nanoserver was built to provide just enough API surface to run apps that have a dependency on .NET core or

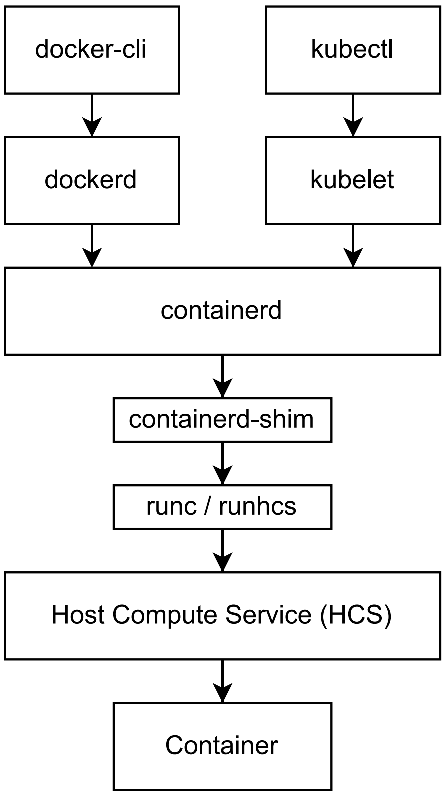


FIGURE 3.3: Abstraction layers for ContainerD on Windows. The image shows the technology stack from the Docker and K8s command line to the Container layer.[7]

other modern open source frameworks. PowerShell, Windows Management Instrumentation, and the Windows servicing stack are absent from the Nanoserver image"[32].

The design of containerization of OT envisages the usage of "Server Core" as base image. Even though the "Server Core" image is not the smallest base image, it provides full functionality for the required technologies for the current use case.

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 is distributed on multiple hosts. K8s supports application rollouts only as container images. To be able to distribute the services on a cluster management tool a containerization of the application is necessary. Each service must be operated on the cluster to profit from the features which K8s offers. For example, it should be possible to offer multiple replications of services - with the compute services included. This is the only way to ensure that the application can be delivered reliably. However, for testing the feasibility it is sufficient to deploy only the main services as a first step. This involves GSS, LSS and AUTH. Subsequently, the LSS spawns the compute services as sub processes inside the same container. In order to communicate with the UI front end, the connection between the external network and the relay service has to be ensured.

The MongoDB server is not required to run on Windows. Because setting it up on Linux is easier and better supported, a dedicated Linux node for running the MongoDB server is preferred. Furthermore, automation for a MongoDB container image on Windows is currently unsupported, because of an design limitation in Windows containers[33]. The OT services, however, have to run on Windows and are therefore deployed on a Windows node.

The encrypted communication between services has to be ensured. As a consequence, the certificates need to be accessible for each service. The certificate files shall be published on the container file system using file mounts. This increases flexibility, because files can be easily swapped out and generated for each service outside the container. Certificate files then have to be created during pod creation. However, as a first step, the certificates are generated as part of the image build process to keep the automation simpler.

Chapter 4

Implementation

This chapter provides an overview about the required changes of OT and describes the containerization of the microservices. This involves the creation of container images and special handling of the certificates. Further, it shows the steps that were performed during the cluster setup.

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.

4.1.1 Code changes on core application

For containerizing the OT application, several changes were applied to its code base. This improves error handling and error tracing of the application and therefore simplifies development of the cluster. In the following sections, the several code changes are described in detail.

Introduction of exit codes

The microservice DLL files have return codes for different error cases. This is, for instance a code 200 in the AUTH for connection issues to the database. As with process exit codes, a zero

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

return code indicates a successful termination. Even though the Rust based main executable “open_twin.exe” retrieves the return code, it did not convert these codes into proper process exit codes. Exit-Codes are a crucial part of the K8s’s life cycle management as described in subsection 3.2.3. Therefore, the return codes need to be converted into process exit codes. Otherwise, K8s is unable to distinct application failures from regular process termination and does not restart the pods accordingly.

```rust
85 let _result = initialize(siteid_c_str.as_ptr(), service_c_str.as_ptr(), db_c_str.as_ptr(),
     dir_c_str.as_ptr());
```

Listing 4.1: Former code snippet from main executable for calling the microservice library code in Rust (*/Microservices/OpenTwin/src/main.rs*)

The surrounding lines of code in the main executable are shown in Listing 4.1[2]. It calls the initialization method of the microservice DLL, passes all its parameters and retrieves its return code.

```rust
86 if _result != 0 {
87   eprintln!("Library/Service initialization ({}:init()) failed with exit code {}",
       lib_path, _result);
88   std::process::exit(_result);
89 }
```

Listing 4.2: Code changes in Rust main executable for additional treatment of exit codes (*/Microservices/OpenTwin/src/main.rs*)

A new condition for non-zero exit codes was added as shown in Listing 4.2. This forces the main executable to terminate the process on any service failure, instead of proceeding and failing afterwards. Thus, errors are more comprehensible. The variable “`_result`” contains the returned exit code of the library DLL, whereas “`lib_path`” contains the path to the library DLL. As can be seen, it is used to describe the affected service name in an error message (line 87).

Debug verbosity in launcher

By default, Rust programs show a window for console output, no matter if it was built in release mode or with debug parameters. However, the main executable uses conditional compilation to set configuration attributes about the window subsystem.

[2] The code listings in this thesis have the corresponding file mentioned as part of the caption. Furthermore, their line numbers are aligned to the corresponding file.

```
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]
```

Listing 4.3: Conditional compilation for disabling console output in non-debug builds (*/Microservices/OpenTwin/src/main.rs*)

Rust provides a compilation option “`debug_assertions`” that is set to “true” for compilations without code optimization[34]. Therefore it is set to “true” if the application runs in debug mode. As shown in Listing 4.3 with conditional compilation, this build option is checked and console output is only shown for debug builds. If the windows subsystem is set to “`windows`”, no console windows are rendered after starting the application.

Even if output is shown on the console window for debug builds only, the output messages are not able to be read conveniently in cases where the application crashes. This is because the console windows close immediately after termination of the program. This impedes debugging and diagnosis of regular application errors outside of a containerized environment. To avoid this behavior, the launcher batch files were adapted. For this, a new parameter was introduced to the batch file to enable running the batch file in verbose mode.

```
18  IF "%~1"=="/V" (
19    REM OT is opening console windows in debug build, so we want to pause them at the end
20    SET pause_prefix=cmd.exe /S /C "
21    SET pause_suffix=" ^& pause
22    ECHO ON
23  )
```

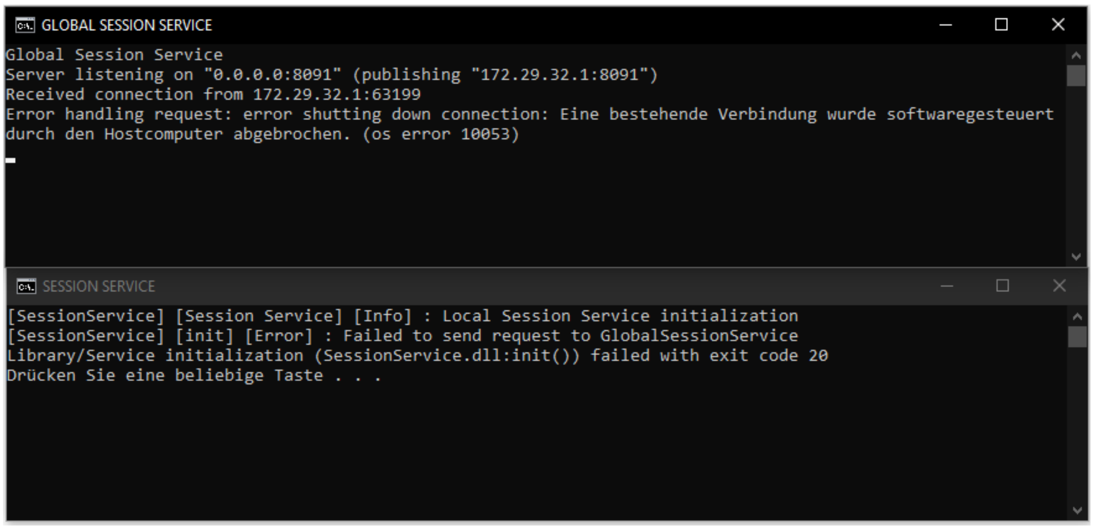
Listing 4.4: Additional command argument for preventing close of window after termination (*/Microservices/Launcher/OpenTwin_session.bat*)

As first step, a new command line argument has been introduced. If “/V” is appended to the start of the launcher batch file it will run normally and stops after proceeding its action. As can be seen in Listing 4.4, with “/V” appended, two new variables “`pause_prefix`” and “`pause_suffix`” are initialized (line 20-21). Furthermore, command output is enabled to debug the launcher file itself (line 22).

```
START "AUTHORIZATION SERVICE" %pause_prefix%open_twin.exe AuthorisationService.dll
  "%OPEN_TWIN_SERVICES_ADDRESS%:%OPEN_TWIN_AUTH_PORT%" "%OPEN_TWIN_MONGODB_ADDRESS%"
  "%OPEN_TWIN_MONGODB_PWD%"%pause_suffix%
```

Listing 4.5: Additional command extension for preventing close of window after termination (*/Microservices/Launcher/OpenTwin_session.bat*)

The newly defined variables "`pause_prefix`" and "`pause_suffix`" are then appended to the command of starting a service. This is shown in Listing 4.5. It ensures that a service process is started in a new window and the command is followed by the Windows "pause" command to stop and wait for user interaction. As shown in Figure 4.1, the user is now able to read error messages if application crashes occur.



```
Global Session Service
Server listening on "0.0.0.0:8091" (publishing "172.29.32.1:8091")
Received connection from 172.29.32.1:63199
Error handling request: error shutting down connection: Eine bestehende Verbindung wurde softwaregesteuert
durch den Hostcomputer abgebrochen. (os error 10053)
```

```
[SessionService] [Session Service] [Info] : Local Session Service initialization
[SessionService] [init] [Error] : Failed to send request to GlobalSessionService
Library/Service initialization (SessionService.dll:init()) failed with exit code 20
Drücken Sie eine beliebige Taste . . .
```

FIGURE 4.1: Application errors in verbosity mode, shown with GSS and LSS. The user is now able to read the errors, since the processes are started in new windows and wait for user interaction.

Enhanced logging and error tracing

For improving the error tracing, the overall log amount has been increased. This involves enabling the central logging functions inside the library "OpenTwinCommunication". Currently, there is an ongoing work to replace the logging from standard output by forwarding the log lines to a central logging service. The library offers C++ macros for sending log messages to this logging service. However, the logging service is currently not yet fully implemented. This is why the logging mechanism did not use logging to standard output. Instead, calls to the respective macro functions for logs were just ignored. This was changed and logging to standard output has been introduced.

```cpp
if (((int)_severity) < ((int)m_logLevel)) {
  return;
}
```

LISTING 4.6: Comparison of the set log level with the log severity of the message. (*/Libraries/OpenTwinCommunication/src/ServiceLogger.cpp*)

Log messages also have information stored about their severity that can be converted from "enum" values to their numeric representation. With respect of the high amount of printed log messages, they are now filtered by their severity. This filter mechanism is shown in Listing 4.6. If the converted log severity is less than the defined log level, the log message is ignored.

Additionally, more calls to the logging mechanism, including caller information, have been added. For instance, the AUTH now shows error messages for caught exceptions and errors during database initialization. Also, the general error tracing has been improved. In the main services GSS, LSS and AUTH, exception handling has been added, where it was not present before. Furthermore, the formatting of exception messages was improved and more information has been added.

Certificate changes

Since OT is using mTLS for communication between services, the CSR file needs to contain the host specification for the outgoing IP address and host name. Thus, in OT a batch script substitutes placeholders in a template CSR file. The respective section of the CSR template is shown in Listing 4.7.

```
"hosts": ["$HOSTNAME$", "$IP_ADDRESS$", "localhost", "127.0.0.1"],
```

Listing 4.7: Variables defined in the CSR are substituted by their respective values. The static host definition for "localhost" and "127.0.0.1" have been added. (*/Deployment/Certificates/server-csr_template.json*)

However, this process does not work if performed for a containerized application. If the replacement takes place during the creation of the container, the final host name or IP address does not yet exist. If, on the other hand, the automatic replacement is done later, it cannot be carried out from outside the container, as the script automatically replaces only the local host name. To solve this problem, the CSR template was extended by fixed values for "localhost" and "127.0.0.1". This already covers the majority of use cases. Additionally, it is recommended to not generate the certificate as part of the container image build process. Instead it should be inserted via a file mount.

Listening on all interfaces

Container images have a certain network interface for communicating to external hosts. Processes inside the container have to bind to this network interface. However, the IP address of this network interface is unpredictable during compile time. The processes that run inside a container therefore have to bind to all available network interfaces to be accessible to the outer network which is performed by binding on the address “0.0.0.0”. Moreover, the services exchange service information with other services as described in section 2.2. Because the binding address “0.0.0.0” is invalid and not accessible from the public, it must differ from this published address in a containerized environment.

Instead of binding to all network interfaces, the main executable in OT was only able to bind to a given IP address (based on the published service address) only. Also, it was not possible to configure a different binding address than the one published to other services.

```rust
let listener = net::TcpListener::bind(&service_url).await?;
println!("Starting server at {:?}", service_url);
```

Listing 4.8: Listener binding before the applied changes (*Microservices/OpenTwin/src/main.rs*)

Listing 4.8 shows the affected lines of code in the main executable. The passed argument for the service URL is forwarded to the server listener class. Afterwards, the address is printed on the console. The service URL, here passed as variable, consists of the service address and the port of the service.

To not affect the outer interface, the changes work with the data already provided. This means, the passed arguments to the main executable are not altered.

```rust
let service_port = Url::parse(&format!("https://{}", service_url))
.expect(&format!("Invalid service url. Unable to parse service url: {}", service_url))
.port();
if service_port.is_none() {
        panic!("Invalid service url. Service url is lacking port defintion: {}",
    service_url);
}
let binding_address = format!("0.0.0.0:{}", service_port.unwrap().to_string());
let listener = net::TcpListener::bind(&binding_address).await?;
println!("Server listening on {:?} (publishing {:?})", binding_address, service_url);
```

Listing 4.9: Listener binding after the applied changes. The service url is parsed, based on its port. Binding is done on all interfaces. (*Microservices/OpenTwin/src/main.rs*)

The binding address is separated from the published address, since the address in the argument still gets passed to the microservice DLL. Afterwards, the service URL is processed as shown in Listing 4.9. First, the given service URL is parsed and the port number is extracted (lines 150-152). If no port is found or the parsing failed, the application fails and shows an error (lines 153-155). Afterwards, the port number is concatenated with the binding address "0.0.0.0" and therefore the server binds to all addresses (lines 156-157). The last line shows the new output containing the port number and the address published to other services.

4.1.2 Container definition

There were three container images prepared for containerization of OT. These cover the main services for GSS, LSS and AUTH. The structure is the same for each of the container files. The first part of the container file is shown in Listing 4.10.

```
ARG BASE_IMAGE_TAG=ltsc2022
FROM mcr.microsoft.com/windows/servercore:$BASE_IMAGE_TAG
```

LISTING 4.10: Containerfile for the LSS. Description of the base image and variable tagging using a build argument. (*Distribution/Container/session.Containerfile*)

The first line introduces a build argument for defining the target image tag from the command line without altering the container file. It is used afterwards to pass the image tag to the base container image. The subsequent lines define labels for the resulting image.

```
ENTRYPOINT ["cmd", "/C"]
CMD ["open_twin.exe", "SessionService.dll", "0", \
"%OPEN_TWIN_LSS_SERVICE_ADDRESS%:%OPEN_TWIN_LSS_PORT%", \
"%OPEN_TWIN_GSS_SERVICE_ADDRESS%:%OPEN_TWIN_GSS_PORT%", \
"%OPEN_TWIN_AUTH_PORT%"]
EXPOSE 8093
WORKDIR C:/app/Deployment/Certificates
COPY ./ ../
RUN createCertificate.bat && certutil -addstore root %OPEN_TWIN_CA_CERT%
WORKDIR C:/app/Deployment
RUN C:\app\Deployment\VC_Redist\VC_redist.x64.exe /install /quiet && \
    move opengl32sw.dll opengl32.dll
```

LISTING 4.11: Containerfile for the GSS. Description of the command line and certificate creation. (*Distribution/Container/session.Containerfile*)

Listing 4.11 shows the lower section of the container file for containerizing the LSS. The container files differ in the runtime specification. They run the same entry point, but have a different process running as command. Besides, the exposed port depends on the service inside the container. After defining the command and copying the files into the container image (lines 22-28), the certificates are built as part of the image file system (line 29). Since the compute services are dependent on C++ libraries that are not present in the base image, the Microsoft Visual C++ Redistributable must be installed as well (line 31). The second command of this line performs the renaming of the OpenGL software rendering DLL. This is necessary, because OpenGL version 2.0 is not available inside the container.

4.2 Cluster Setup

The following section describes the setup of different nodes in the cluster. 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. The used version of K8s is "v1.26". The version of ContainerD is "v1.6.8", on both sides, the worker nodes and the master node.

4.2.1 Setting up the master node

For setting up the master node on Linux, a system based on Debian Bullseye 11.5 was used. Basis for the performed steps is the K8s documentation for setting up a container runtime[35] and the online tutorial of Basappa[9].

4.2.1.1 Installing prerequisites and ContainerD

K8s utilizes resources as good as possible. Thus, the memory on each node is used to nearly 100%. Additionally, OSs have a built-in mechanism for freeing the memory and storing unused data from the random access memory to the hard drive. This mechanism is called "Swapping". However, this is an unwanted behavior, since letting K8s read its data from the relatively slow hard drive would decrease the performance of the cluster.

Hence, 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 as shown in Listing 4.12.

```
$ sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
$ systemctl mask dev-sda3.swap
```

LISTING 4.12: Bash commands for disabling Swap mechanism and masking.

The container runtime requires certain kernel features to be enabled and need to facilitate VxLAN and overlay filesystems. On Debian 11, the required kernel modules for virtual networking facilities (“`br_netfilter`”) and overlay filesystems (“`overlay`”) are not enabled by default. Thus, they have to be manually enabled. To permanently enable them and keep them enabled across reboots, the setting is stored in a file in the directory for loaded modules as shown in Listing 4.13.

```
$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF
```

LISTING 4.13: Bash commands for enabling required kernel modules[9]. (*/etc/modules-load.d/k8s.conf*)

For functional networking on the master node, K8s also requires internal network packets being forwarded to the pods. Additionally, the tracking table for tracing network packets and their assigned connection is increased. This is a prerequisite for the applied NAT which is used in container networking. New files with respective settings are created in the directory for kernel settings as shown in Listing 4.14.

```
$ cat <<EOF | tee /etc/sysctl.d/k8s.conf
  net.bridge.bridge-nf-call-iptables  = 1
  net.bridge.bridge-nf-call-ip6tables = 1
  net.ipv4.ip_forward                 = 1
  net.netfilter.nf_conntrack_max      = 524288
EOF
$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

LISTING 4.14: Bash commands for enabling IP forwarding and network filtering.

After installing ContainerD with all its prerequisites from the package registry, a configuration file (“`config.toml`”) is created. Subsequently, the SystemD cgroup driver is enabled in the runtime options of ContainerD and its service is restarted. cgroups are a feature of the Linux kernel for managing and restricting groups of processes[36]. One provider of the cgroup is the Linux system and service manager “SystemD”. The container runtime of K8s can utilize cgroups to perform process isolation. For enabling the SystemD cgroup driver, the commands from Listing 4.15 are applied.

```
$ containerd config default |
  sed 's/\[plugins.\"io.containerd.grpc.v1.cri\".containerd.runtimes.runc.options\]/&\n
    SystemdCgroup = true/' | tee /etc/containerd/config.toml >/dev/null
$ service containerd restart
```

LISTING 4.15: Bash command for setting up containerd config

To finally start the cluster, the packages for the Kubelet service, “kubeadm” and “kubectl” need to be installed. The cluster can then be initialized by running the command line tool as shown in Listing 4.16. Since the configuration parameters can also be passed as YAML file, this is the preferred method of passing arguments and allows version control of the required configuration.

```
$ kubeadm init --config kubeadm_config.yaml
```

LISTING 4.16: Bash command for setting up the cluster

Since Flannel is used as network overlay, the IP address range provided for pods must be “10.244.0.0/16”[10]. In recent versions of K8s, ContainerD is not yet the standard CRI. This is why a value is provided that sets the CRI socket address for newly registered nodes to the one of ContainerD. Furthermore, the cgroup driver for the Kubelet service on worker nodes is set to SystemD, because the cgroup driver on Linux worker nodes should be aligned to the master node. The information passed as configuration is presented in Listing 4.17.

```
kind: ClusterConfiguration
apiVersion: kubeadm.k8s.io/v1beta3
kubernetesVersion: v1.25.3
networking:
  podSubnet: "10.244.0.0/16" # pod-network-cidr
---
kind: InitConfiguration
apiVersion: kubeadm.k8s.io/v1beta3
nodeRegistration:
  criSocket: unix:///var/run/containerd/containerd.sock
```

```
---
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
cgroupDriver: systemd
```

LISTING 4.17: YAML configuration for providing configuration for the cluster and newly registered ndoes. (*Distribution/Controlplane/kubeadm_config.yaml*)

4.2.1.2 Applying a Container Network Interface

After successfully initializing the cluster, the overlay network for Flannel must be set up. For this, the respective entity description can be directly downloaded from the vendor[3]. Networking has to blend with Flannel on Windows and communication should be achieved on the same VxLAN identifier throughout the cluster. This is why the Virtual Network Identifier (VNI) (4096) and port (4789) for Flannel on Windows must be set as part of the configuration. For this, manual editing of the description is required. The respective values are added to the configuration map section where the network configuration file is described (the section "net-conf.json" of "kube-flannel.yml") as shown in Listing 4.18. This file is automatically created on newly registered nodes. The changed entity description is applied on the cluster respectively.

```
net-conf.json: | {
  "Network": "10.244.0.0/16",
  "Backend": {
    "Type": "vxlan",
    "VNI" : 4096,
    "Port": 4789
}}
```

LISTING 4.18: Fixup for Flannel manifest. Here the values "`VNI`" and "`Port`" were added. (*kube-flannel.yml*)

4.2.1.3 Adding the proxy daemon sets

The usage of Windows worker nodes in a K8s cluster requires additional configuration mappings and daemon sets to be applied on the cluster. Those are used for setting up a proxy for

[3]Flannel entity description: https://raw.githubusercontent.com/flannel-io/flannel/master/Documentation/kube-flannel.yml

Flannel. The Special Interest Group (SIG) "Windows Tools" of K8s provides the additional objects[10]. Before applying them, a replacement of the K8s version and the Flannel version is performed. This procedure is achieved by running the two commands shown in Listing 4.19.

```
$ curl -L https://raw.githubusercontent.com/kubernetes-sigs/sig-windows-tools/master \
  /hostprocess/flannel/kube-proxy/kube-proxy.yml |
  sed 's/KUBE_PROXY_VERSION/v1.25.3/g' |
  kubectl apply -f -
$ curl -L https://raw.githubusercontent.com/kubernetes-sigs/sig-windows-tools/master \
  /hostprocess/flannel/flanneld/flannel-overlay.yml |
  sed 's/FLANNEL_VERSION/v0.17.0/g' |
  kubectl apply -f -
```

LISTING 4.19: Bash command for adding the flannel overlay configuration[10]

4.2.2 Setting up the worker node

The initialization of the worker node is mainly oriented on the guide for adding Windows nodes that is provided from the SIG "Windows Tools" on GitHub[10]. Although there is an original guide from K8s[25], it was not used as basis because it is outdated.

First, the node preparation scripts of the K8s SIG "Windows Tools" are retrieved and executed as shown in Listing 4.20. The first script retrieved (*Install-Containerd.ps1*) installs ContainerD and its prerequisites on Windows. Installation of the prerequisites involves the Windows features "Hyper-V", "Hyper-V Tools", "Hyper-V for PowerShell" and "Containers". Once the features are installed and the system has successfully rebooted, the script has to be restarted. It then proceeds by downloading the binary of ContainerD and adding its location to the "PATH" environment variable. ContainerD's configuration file is changed to align the CNI file locations with K8s. The script ends by registering ContainerD as a service.

The second script (*PrepareNode.ps1*) prepares the node for joining the cluster. It first downloads the Kubelet executable and creates a script file that is called by the Kubelet service. The created script file contains runs the Kubelet in consideration of version dependent command arguments. Afterwards, the Non-Sucking Service Manager (NSSM)[4] is downloaded and the Kubelet script is registered as a service. Finally, the *PrepareNode* script also sets up firewall rules for Kubelet.

[4]NSSM: https://nssm.cc/

Furthermore the command line tools “kubectl”, “kubeadm”, “crictl” and “nerdctl” are installed. While installing “crictl”, it is added to the “PATH” environment variable.

```
> curl.exe -LO https://raw.githubusercontent.com/kubernetes-sigs/sig-windows-tools/master/
    kubeadm/scripts/Install-Containerd.ps1
> .\Install-Containerd.ps1
> curl.exe -LO https://raw.githubusercontent.com/kubernetes-sigs/sig-windows-tools/master/
    kubeadm/scripts/PrepareNode.ps1
> .\PrepareNode.ps1 -KubernetesVersion v1.25.3
```

Listing 4.20: PowerShell commands for retrieval of node preparation scripts[10]

After successfully running the preparation script the node should be ready to join the cluster. For joining the cluster a token is generated on the master node as shown in Listing 4.21.

```
$ kubeadm token create --print-join-command
```

Listing 4.21: Command for creating a join token and printing the join command.

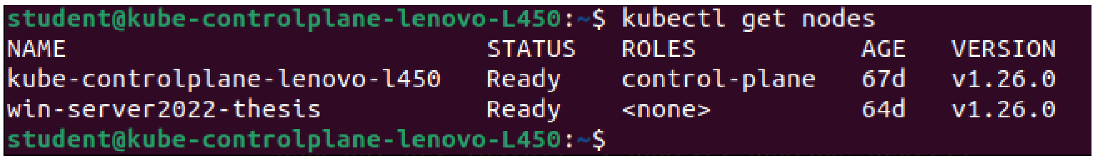
The token is copied to the prepared node and used as part of a “join” command. Listing 4.22 shows the command for joining the cluster, whereas the token is substituted by “TOKEN”.

```
> kubeadm.exe join 1.2.3.4:6443 --token TOKEN --discovery-token-ca-cert-hash sha256:HASH \
  --cri-socket "npipe:////./pipe/containerd-containerd"
```

Listing 4.22: Command for joining new nodes on Windows

Besides containing the token and the master node’s IP address, the command also has the hash value of the CA of the master node. Both are contained in the generated “join” command that is created during the token generation. For K8s versions “v1.25” and below the command also requires the explicit declaration of ContainerD as container runtime. Thus, the additional option “`cri-socket`” is defined.

The successful join of the node is verified by running “`kubectl get nodes`” on the master node. This prints the newly added node. The output is depicted in Figure 4.2.



```
student@kube-controlplane-lenovo-L450:~$ kubectl get nodes
NAME                            STATUS   ROLES           AGE   VERSION
kube-controlplane-lenovo-l450   Ready    control-plane   67d   v1.26.0
win-server2022-thesis           Ready    <none>          64d   v1.26.0
student@kube-controlplane-lenovo-L450:~$
```

Figure 4.2: Output of “`kubectl get nodes`” command after adding a Windows node to the cluster. The nodes are ready to run pods.

4.3 Automatic setup

Even though the scripts provided from the SIG “Windows Tools” cover most of the preparation steps, some manual steps are still required as described in the respective guide[10]. To automate the preparation process where possible, and to also provide tools for debugging error cases, further steps are required. Thus, a custom script (*Distribution/Container/setup-node.ps1*) was created to provide this aid.

The script automates the process of rebooting and rerunning the preparation after the required Windows features are installed, and provides instructions for steps that cannot be automated. It also checks the prerequisites before performing any action. This means, it is checked whether the OS is “Windows Server” and if a certain minimum version of Windows is used. The check for prerequisites is shown in Listing 4.23.

```
if (-not ((Get-ComputerInfo).WindowsProductName | Select-String "Server")) {
  throw "Prerequisites not met. Windows Server is required as operating system."
}
if (-not (Get-Hotfix -ErrorAction Ignore KB4489899)) {
  if (-not ([System.Environment]::OSVersion.Version -gt [System.Version]"10.0.17763.0")) {
    throw "Prerequisites not met. You either need KB4489899 installed, or a Windows
    Version higher than 10.0.17763"
  }
}
```

Listing 4.23: Powershell commands in the automated setup script. Checks for prerequisites.

Besides checking for system requirements, the custom script also installs tools for debugging. This includes the Windows version of “kubeadm” and “kubectl”, as well as the command line tools for debugging containers on Windows, namely “crictl” and “nerdctl”. Furthermore, the custom script also excludes the ContainerD process from the Windows Defender Virusscanning as shown in Listing 4.24. This increases the pull and general runtime performance of containers. The installation of the command line tools also involves setting up PowerShell auto completion and include to the “PATH” environment variable.

```
Add-MpPreference -ExclusionProcess "$Env:ProgramFiles\containerd\containerd.exe"
```

Listing 4.24: Powershell command in the automated setup script. Exclusion of all actions performed by containerd.exe from Windows Defender.

4.3.1 Cluster design

Adding worker nodes to the existing cluster was investigated in three distinctive scenarios. Starting point of the investigation was the most complex approach (Figure 4.3). The complexity of the system was subsequently reduced in order to break down errors that were related to networking, the encrypted communication and the host-process isolation. Those scenarios are described here more in detail.

The first scenario, as seen in Figure 4.3, is the most complex one. The two physical computers shown in the figure represent the machines where the K8s nodes are running. Here, “Physical computer 1 (PC1)” is a local Linux machine in the role of the control plane, whereas “Physical computer 2 (PC2)” is a high-performance computer for running the worker. Because PC2 is located in the university and therefore access restricted, the organizationally simplest approach was to set up a Hyper-V powered VM on PC2. The VM then had unrestricted access. The K8s compute node is set up in this VM. Both physical machines are connected to the Internet and not within the same network. Instead of accessing PC2 directly via a public IP address, both computers have to be connected to a Virtual Private Network (VPN). VPN is a technology, where network traffic is encrypted to enable access to an internal network from the outside.

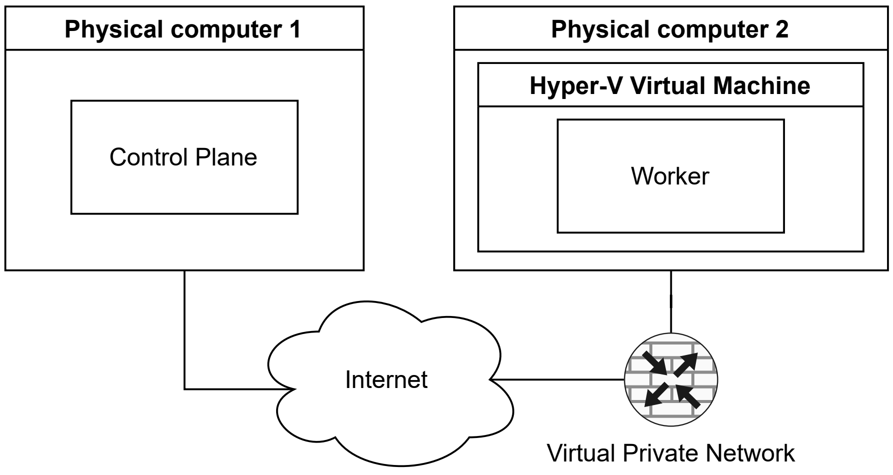


FIGURE 4.3: Cluster design scenario where worker node is running on a dedicated machine inside the protected University network. Additionally, the worker node is hosted inside a Hyper-V VM to have another isolation layer and full admin privileges on the worker node.

The problematic point in the first scenario is the additional abstraction layer between the Hyper-V VM for administering the worker node and the physical machine. It involves additional network interfaces that translate the addresses from within the VM to the external network. Misconfiguration in one of those layers can hinder the start of host-process containers that try to bind on these network interfaces.

The second scenario is shown in Figure 4.4. Here, the worker is outside the VM and runs "bare metal". The physical computers are still only accessible within the VPN and not located in the same network. Since the VPN is still necessary for the connection between PC1 and

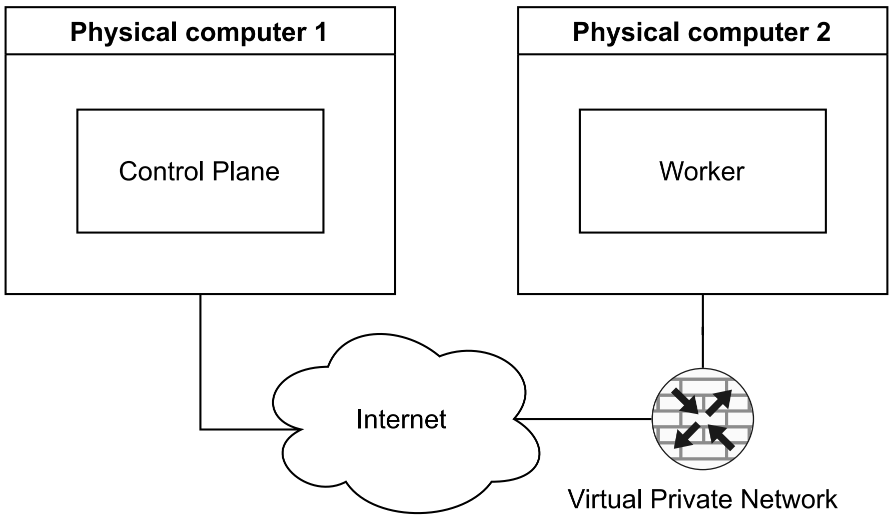


FIGURE 4.4: Cluster design scenario where worker node is running on a dedicated machine inside the protected University network.

PC2, this caused problems in the connection between Control Plane and worker node. While implementing the second scenario, IP addresses could not be assigned by K8s. After starting the containers, the HCS threw the error "IP address is either invalid or not part of any configured subnet(s)". This was justified by the VPN network interface, since it has to support a large amount of subnets and the acquired IP addresses were already in use.

This leads to the third scenario, shown in Figure 4.5, which is the most basic. Here, the two physical computers are connected directly in the same network. This setup is preferred, since this is the anticipated working scenario. Here, the containers were able to start and got an IP address assigned.

4.3.2 Deployment of the application

As a first prototype, only the LSS of the application is deployed. Deployment of the application is done by applying the manifest as shown in Listing 4.25. The manifest defines a pod using the container image of the LSS (lines 2-4). Since the images are not yet provided via an image

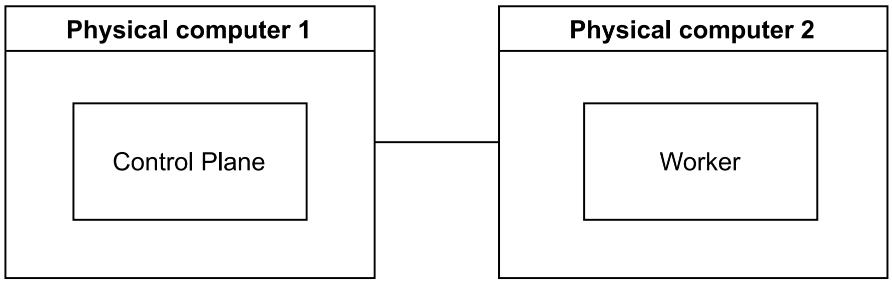


FIGURE 4.5: Cluster design scenario where master and worker node are both running on a dedicated physical machine each.

registry, the images are built and stored locally. Thus, the K8s is directed to never pull the image and retrieve it locally instead. The passed environment variables (lines 5-13) are defined by the section "env", whereas the variable "OPEN_TWIN_LSS_SERVICE_ADDRESS" is set with the IP address of the pod. The other variables are set with the static address of the database and the IP address of the GSS respectively. The external port is defined as 8093, which is equivalent to the port of the LSS. The node selector property (lines 16-17) defines a requirement for the node scheduler to select only Windows nodes. The manifest is applied on the cluster using "kubectl apply".

```yaml
 1  containers:
 2  - name: opentwin-session
 3    image: local.dev/opentwin-session:latest
 4    imagePullPolicy: Never
 5    env:
 6      - name: OPEN_TWIN_LSS_SERVICE_ADDRESS
 7        valueFrom:
 8          fieldRef:
 9            fieldPath: status.podIP
10      - name: OPEN_TWIN_MONGODB_ADDRESS
11        value: 1.2.3.4:27017
12      - name: OPEN_TWIN_GSS_SERVICE_ADDRESS
13        value: 5.6.7.8
14    ports:
15    - containerPort: 8093
16  nodeSelector:
17    kubernetes.io/os: windows
```

LISTING 4.25: Partial section of configuration for cluster deployment of the LSS (*Distribution/Kubernetes/open_twin.yaml*)

Chapter 5

Results

Several problems were discovered during the development of the K8s cluster and the containerization of OT. The results are discussed in this chapter.

5.1 Containerization

Issues occurred during the containerization of the application. This involves pitfalls with the development of the container images and the special cases with the new Windows host-process container technology. The required considerations are discussed in this section.

5.1.1 Container manifest

Even though the format of the container manifests ”Containerfile” is compatible to the proprietary ”Dockerfile” format from Docker, the CRIs do not follow the specification of the everywhere[37]. This was an issue while writing a Containerfile for the ContainerD CRI. Especially in cases where line breaks in the Containerfile were 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[37].

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

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

Listing 5.1[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 "\". Therefore, it results in just "`open_twin.exe`" as interpreted entry point. This causes troubles during build and execution of a container image. If the container image is built in Docker, it does not run in ContainerD. However, the image cannot be built in ContainerD, because the required extension "build-kit" is not available on Windows. To work around this, and to be able to run an image that was built with Docker in ContainerD, the *ENTRYPOINT* definition has to be written in JSON notation. If defined as in Listing 5.2[1], the interpreter of ContainerD successfully runs the image.

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

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

5.1.2 Windows Base Image

Because of the light-weight isolation layer of containers, the OS of container images must match those on the host machine. Compared to virtualization this light-weight approach reduces the isolation on the one hand, but improves performance on the other. It is a requirement to have the same OS kernel when running container images on Linux. However, on Windows host-process isolation it is a requirement to have the exact same build version for the base image as for the host machine. This means, a container based on Windows Server 2019 (Version 10.0.17763.4010) cannot run on Windows Server 2022 (Version 10.0.20348.1547). The isolation that is built on Hyper-V virtualization does not have this requirement. Microsoft provides several image tags ("ltsc2019", "ltsc2022", and so on) for different kinds of OS versions[32, 38]. They also offer one general image tag for multiple architectures[38]. However, they do not offer a more generous tag having a broad version coverage, even though the CRIs support it. This is why the container image tag always has to match the host OS.

There are two different ways to solve this flaw of OS inconsistency. One method is given by K8s. K8s supports filtering based on the build number of a node[39]. For this, a separate "`nodeSelector`" key is defined as shown in Listing 5.3.

[1]For showcasing the problem, the line has been shortened. In the real scenario, the command expects more parameters and therefore is longer.

```
nodeSelector:
  "beta.kubernetes.io/os": windows
  "beta.kubernetes.io/osbuild": "14393.1715"
```

LISTING 5.3: K8s manifest with node selector for specific Windows build.

A second method is to build different versions of one image and assign them the same tag. If rolled out via an image registry, the node would automatically retrieve the suitable architecture. Google showed that so called “multi-arch” images can also be created with multiple build versions[40].

5.1.3 Local images namespace

If images are present locally on the node, they have to be in the ContainerD namespace ”k8s.io” to get recognized by K8s. However, ContainerD on Windows is not able to build images because the Windows support for the build extension “build-kit” is still under development[41]. If images are built with Docker, they are not recognized by ContainerD and additionally get the default namespace assigned. Although, images pulled with K8s from a image registry are getting imported correctly. The faulty namespace behavior cannot be changed during the image build, because the CLI of Docker does not know about image namespaces and therefore does not provide the necessary option. Thus, the change of the namespace is required for locally provided images on the one hand, and it requires a transfer of the image from the Docker environment to the ContainerD environment on the other hand. So the only way to use local images without uploading them to an image registry is to perform the following steps for each image:

1. Build the image in Docker (using `docker build`)
2. Export the image to an archive file (using `docker save`)
3. Load the image to ContainerD to correct namespace (using `ctr --namespace k8s.io image import`)

Because the application images of OT become very large (~2.8 Gigabyte), the three steps taking about five to ten minutes each. Depending on the amount of container images to process and the performance of the host computer, this consumes about half an hour.

To automate this process for a batch of container files, the PowerShell script *build-image.ps1* was created. The script reads container files in a directory and afterwards performs the described steps on it as shown in Listing 5.4. The defined variable "`$ContainerRegistry`" is set to the address of the image registry for tagging the image. In this case it is the stub domain "local.dev". If an image is not tagged with a proper domain name, the CRI is unable pull it and it cannot be found by K8s. "`$BaseName`" is the filename of the container file and therefore the name of the image. The created temporary file *container.tar* is deleted by the script after it has finished.

```
$containerfiles | ForEach-Object {
  docker build -f "$($_)" -t "$($ContainerRegistry)/opentwin-$($_.BaseName):latest" \
    "..\..\Deployment\"
  docker save "$($ContainerRegistry)/opentwin-$($_.BaseName):latest" -o container.tar
  ctr --namespace k8s.io image import .\container.tar }
```

Listing 5.4: Crucial commands used in the custom PowerShell script to automate process of image building.

5.2 Kubernetes

Flaws with the documentation and the compatibility between different versions have been found while working with K8s. This highlights the special treatment that is required to add Windows nodes to the cluster. The details are described in this section.

5.2.1 System requirements

Because K8s does not support containers based on Hyper-V isolation[42], host-process containers must be used. For using host-process containers on Windows, at least Windows Server in version 2019 with build number "17763.379" is required or the update "KB4489899" needs to be installed[10]. Furthermore, K8s also has the fixed requirement of the usage of "Windows Server 2019" and above[42]. This reduces the subset of compatible node OSs, because Windows client machines are not able to host containers for the usage in K8s and a minimum version of "Windows Server" is required. Moreover, the update cannot be installed manually which means it is impossible to get support for "Windows 10" based systems that are not part of this group. In the business area, Microsoft sells major upgrades of Windows Server as separate product. That is why the necessity of upgrading to at least “Windows Server Build 17763” can induce a higher expense.

Additionally, there is the incompatibility with the control plane on Windows. This requires an additional Linux node for running the K8s control plane and hence is not conducive for a homogeneous system landscape.

5.2.2 Documentation

The official documentation of K8s also hides some obstacles while implementing a cluster. It turned out to be incomplete or outdated on some pages. To be more precise, there is partially work in progress and while consuming the documentation it is hard to detect if a specific article is applicable for the current version of K8s.

For example, this was fundamental on the page “Adding Windows nodes”[25]. At the current time being, the actual page is only available as an archived version for K8s version “v1.23”. However, if the same URL is called for the unarchived version, the user gets redirected to a similar page “Creating a cluster with kubeadm”[43]. This can easily happen if a link is followed from an issue or a web search. Reading the page, people could assume that they can follow the steps described there to add a Windows node to a cluster. Nevertheless, the fact that the guide is aligned to Linux is only barely mentioned and thus is misleading. That also means that there is no guide for adding Windows nodes in the current version of the documentation of K8s.

The guide for adding Windows nodes provided from SIG “Windows Tools”[10] has a more streamlined approach and benefits from provided PowerShell scripts, like “PrepareNode.ps1” (which is used in subsection 4.2.2). However, due to the small community[44], the guide still stays faulty or outdated. For example, the guide shows a warning, “The instructions and scripts in the directory DO NOT configure a CNI solution for Windows nodes running containerd. There is a work-in-progress PR to assist in this at #239”[10]. But the guide does explain how to configure a CNI and the linked pull request is merged since December 2022. Thus, the warning is misleading. Furthermore, it reads “Do not forget to add –cri-socket [...] at the end of the join command, if you use ContainerD”[10]. The explicit appending of the CRI socket however is not necessary for versions higher than “v1.25” as mentioned in subsection 4.2.2.

The guide from SIG “Windows Tools” was still work in progress while the cluster was set up. A change in the documentation for applying a required daemon set on the cluster was only added after three months. Taking into account that Windows nodes are officially supported for over year, a functional documentation was expected.

5.3 Interfaces within the abstraction layers

The communication within the different abstraction layers is a crucial part of a complex system like K8s. Kubectl has to communicate the commands to the container runtime and Flannel, which communicate with the OS layer, and the hosted applications exchange their status with the CRI. It is crucial for the correctness of the system to have all parts compatible to each other. The flaws related to these problems are described here.

5.3.1 Version interoperability

The huge amount of systems and third-party tools the whole K8s landscape consists of burdens the risk of version incompatibilities. The versions of Flannel and the CNI that are written in the manifest have to match the version installed on every node in the cluster (including the control plane). For example, the Flannel version on the node system has to match the version defined in the Flannel pod manifest. This induces the necessity of string replacements to hand in the correct versions and increases complexity.

Another finding covers the fact that support for Docker was removed from K8s before properly introducing the alternative ContainerD to not only Linux but also Windows. The majority of guides that can be found online still stick to Docker as basis. Guides that cover specific cases with ContainerD are rare. This can also be seen on the missing build support for Windows containers in ContainerD. Until now the container images have to be built in Docker and need to be shifted to ContainerD, because the required tools ("build-kit") are not yet available for Windows[41].

5.3.2 Error handling

The additional abstraction layer for the HCS is loosing information in error cases. Errors on the OS level are passed to the user, but Windows does not provide much additional information alongside the error. For example, in cases with an incorrect container image entry point as described in subsection 5.1.1 the error message returned to the K8s control plane is “hcs::CreateComputeSystem session: The system cannot find the file specified.: unknown”. With no further arguments given, it is hard to break the issue down to the actual file.

Furthermore, error cases on one Windows version differ from the one on other versions. This has been found while setting up the “kube-proxy” pod with the exact same steps on Windows Server 2022 and Windows Server 2019. While the error message on 2019 is rather generic (“The directory name is invalid.”), the error message for 2022 is more explanatory (“IP address is either invalid or not part of any configured subnet(s)”).

Additionally, the error reporting of OT itself can still be more improved. The logging mechanisms on OT are still under development and thus logging is rare. If a service crashes it is tricky to reproduce the cause. Especially for the compute services the generated output does not contain useful information. Due to the fact that the console window does not stay open after a crash, the output is hard to catch. In some cases throughout the code base exceptions are not caught or just silently swallowed, where it is better to treat them and return the exception message. Also, since Rust cannot catch foreign exceptions, they cannot be exchanged between the service DLL and the main executable. This is why all exceptions have to be fully treated inside the service DLL.

5.4 Analysis

The evaluation of the results shows, that a containerized deployment on Windows is still in a early state. Even though, Docker and its CRI was used for a long time, containerization using Windows host process isolation is not convenient or error resilient.

Nevertheless, it is feasible to containerize OT using Windows containers. The deployment on K8s comes with issues, though. K8s provides a checklist for testing the connectivity[45]. In this checklist the following points for checking connectivity should be performed after applying an application manifest on the cluster:

1. Pods are shown in “`kubectl`”[45]
2. Node-to-pod communication across the network[45]
3. Pod-to-pod communication[45]

4. Service-to-pod communication[45]
5. Service discovery[45]
6. Inbound connectivity[45]
7. Outbound connectivity[45]

Considering the checklist above, only points 1, 2 and 3 are working after following the instructions. The pods are accessible by their direct IP address, whereas host names are not resolved. It turns out, the connectivity of the pods in the developed cluster is lacking domain name resolution. During the investigation, the Kube-Proxy service did not work in combination with Flannel on Windows. However, support is rare, because the community is still small and therefore the development team is overloaded[44, 46]. As a consequence, some of the tracked issues on the GitHub project website are not solved and closed out of triage[44, 46].

Chapter 6

Conclusion and future work

This chapter will conclude the final results, and provides an outlook for the future.

6.1 Conclusion

This thesis shows a proof of concept for deploying a microservice application. The application under study is a distributed modeling software that works with services on multiple layers. The application comes with a UI front end which acts as a client, and several services that run on a a centralized server. The server services spawn compute services for delivering the actual results and data.

In the beginning of this thesis the application was given in a locally installed manner. First tests already showed that the application is able to run on a distributed system. The program was containerized using standard container technology from Docker and ContainerD. In the first prototype, the main services for GSS, AUTH and LSS were shifted into a containerized environment. The containerization of OT uncovered some problems. For example, the application had to be adapted to the requirements of the container network. This involves a change of the server binding to all interfaces, to be able to host the service inside a containerized environment, but also the change of the process exit behavior.

The Hyper-V isolation shipped with Windows abstracts some issues with the underlying container isolation and also flattens the requirement of having the same OS kernel on the host system and inside the container. Problems on the OS level are mostly solved and inputs are properly validated so that errors are more understandable. The host-process isolation, however, is providing a pure abstraction layer to the containers. This means that processes inside the container are not virtualized and still use parts of the host system. This hides pitfalls as, for example, networking is not yet fully functional, and errors are sometimes confusing.

Furthermore, the cluster was set up and different paradigms for cluster design have been probed. As it turned out in the end, some major issues were present because of the network topology used for the investigation. The chosen approach was a rather simple one where all nodes are located in the same network. In the end, connectivity within the pods is not fully functional, which shows the struggle of setting up a Windows based cluster.

The results showed many complications that one has to be aware of. This includes missing quality in documentation and difficulties in the error handling of the K8s systems. When it comes to the community, it is hard to find help for the special case with ContainerD on Windows. This might be reasonable due to the small size of the community. From this point of view it is obvious that Windows is not fully supported on K8s yet.

In general the thesis comes to the result that the development of a cluster with Windows nodes using ContainerD is still in a prototype state. ContainerD is just not fully established yet in the area of Windows clustering and therefore comes with problems. Furthermore, it proofs that implementing a cluster with Windows nodes is a process that requires special treatment.

6.2 Future work

The results showed that there is still room for improvement. Using the findings as a basis, there are plenty of opportunities to implement a cluster and enhance it in the future. This section mentions what can be optimized in the future and provides an outcast for future implementations. First, further investigation has to be done on the network connectivity of the pods. Currently, there is no functional domain name system within the K8s cluster, because the Kube-Proxy does not work in combination with Flannel on Windows.

For a production environment, the certificates should not be build as part of the container image build process. Instead the certificate files should be injected into the container file system via mounts.

Moreover, the images size should be more reduced, because roughly three gigabyte per image can cause problems later on. For example, when using a central image repositories like Artifactory[1] this would reduce the required storage space and the download duration during an image pull. A first step in this direction can be to only keep necessary dependencies in an image for a corresponding service.

Since the compute services of OT are not yet containerized each, horizontal scaling of the application is is currently not possible. Having a container image for each compute service, would allow having multiple replicas for the compute services and therefore would improve scaling.

The development team currently works on porting OT from Windows to Linux. Since it was shown that a major problem is caused by the insufficient support of Windows on K8s, the development on the Linux port should be kept up. Majority of the faced problems might be solved on homogeneous Linux clusters.

This might then also simplify the utilization of the graphic processing unit in K8s pods. Since some services of OT perform complex computations on the graphics card, OT would benefit from this change.

In general, keeping an eye on the documentation of K8s is crucial, since as shown in this thesis, the guides that can be found online are mostly still under development.

[1]Artifactory: https://jfrog.com/de/artifactory/

Appendix A

Overview about submitted code

The created code and configuration file was uploaded to the project GitHub repository. The repository has private access only. Access to the repository can be granted by request.

The GitHub repository is located at: https://github.com/pth68/SimulationPlatform

The location of the additional files that are relevant for creation of the cluster are located in the **"Distribution"** sub directory of the repository. Table A.1 describes the submitted files (relative from the "Distribution" directory) and their intention.

Path	**Filename**	**Intention**
Container/	build-image.ps1	Automation script for building, saving, and re-importing container images for ContainerD.
	setup-node.ps1	Script for automated installation of Kubernetes node prerequisites and setup by treating the flaws of a manual installation.
	authorisation.Containerfile	Containerfile for OpenTwin's Authorisation service.
	globalsession.Containerfile	Containerfile for OpenTwin's global session service.
	mongodb.Containerfile	Containerfile for the containerized MongoDB server.
	session.Containerfile	Containerfile for OpenTwin's local session service.
	compose.yaml	A container compose file for testing the setup of containerized services.
Container/mongodb/	0-init.js	A initialization script for setting up the admin user and roles of the OpenTwin database.
	mongodb.conf	The MongoDB Server configuration file.
Controlplane/	kubeadm_config.yaml	The configuration which should be passed during initialization of the cluster using kubeadm.
Kubernetes/	open_twin.yaml	The Kubernetes configuration file for deploying the cluster.

TABLE A.1: List of added files in the code repository and their usage description.

Acronyms

OT	OpenTwin	1
Windows	Microsoft® Windows®	1
OS	Operating System	6
cgroup	control group	23
SIG	Special Interest Group	39
DLL	Dynamic Link Library	4
URL	Uniform Resource Locator	7
UI	User Interface	3
TLS	Transfer Layer Security	8
mTLS	mutual Transfer Layer Security	8
CA	Certificate Authority	9
CSR	Certiticate Request	10
GSS	Global Session Service	5
AUTH	Authorization Service	6
LSS	Local Session Service	6
HTTP	Hypertext transfer protocol	9
API	Application Programming Interface	17
CLI	Command line interface	19
VPN	Virtual Private Network	42
VxLAN	Virtual extensible Local Area Network	21
K8s	Kubernetes	2
IP	Internet Protocol	1
NAT	Network Address Translation	1
VNI	Virtual Network Identifier	38
CNI	Container Network Interface	21
CRI	Container Runtime Interface	19
NSSM	Non-Sucking Service Manager	39
VM	Virtual Machine	11
HCS	Host Compute Service	24

Bibliography

[1] Request for Comments, “The transport layer security (tls) protocol version 1.3,” 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.html

[2] “Mutual tls - what is mtls?” [Online]. Available: https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/ (Visited on 2022-12-11)

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

[4] A. Warrier, “Containers vs vms,” *eG Innovations*, 2020-11-20. [Online]. Available: https://www.eginnovations.com/blog/containers-vs-vms/ (Visited on 2022-11-12)

[5] M. Lukša, *Kubernetes in action: Anwendungen in Kubernetes-Clustern bereitstellen und verwalten*, ser. Hanser eLibrary. München: Hanser, 2018. ISBN: 9783446456020. [Online]. Available: https://www.hanser-elibrary.com/doi/book/10.3139/9783446456020

[6] D. Schott, “Introducing: Kubernetes overlay networking for windows: Windows server networking blog,” 2019. [Online]. Available: https://techcommunity.microsoft.com/t5/networking-blog/introducing-kubernetes-overlay-networking-for-windows/ba-p/363082 (Visited on 2022-03-12)

[7] Scooley, “Windows container platform: Microsoft learn,” 2022. [Online]. Available: https://learn.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/containerd (Visited on 2023-02-28)

[8] “Pod lifecycle: Documentation.” [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/ (Visited on 2022-11-08)

[9] N. Basappa, “Kubernetes on debian 11 (bullseye): Blog article in "day in my life",” 2022. [Online]. Available: https://www.natarajmb.com/2022/06/kubernetes-debian/ (Visited on 2022-12-18)

[10] GitHub, “Guide for adding windows node: Documentation,” 2022-12-27. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/blob/727707fa7d83d401956b467a5ce41700cce7d9a3/guides/guide-for-adding-windows-node.md (Visited on 2023-02-13)

[11] Microsoft, “Windows container version compatibility,” 2022. [Online]. Available: https://learn.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/version-compatibility (Visited on 2023-03-17)

[12] Kubernetes, “Don’t panic: Kubernetes and docker,” 2020. [Online]. Available: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/ (Visited on 2022-11-02)

[13] Request for Comments, “The websocket protocol,” 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6455

[14] Hugo Krawczyk, “A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in tls 1.3),” *Cryptology ePrint Archive*, 2016. [Online]. Available: https://eprint.iacr.org/2016/711

[15] “Hyper-v support in windows 8: Windows experience blog,” 2013. [Online]. Available: https://blogs.windows.com/windowsexperience/2013/06/20/hyper-v-support-in-windows-8/ (Visited on 2022-12-17)

[16] N. Marathe, A. Gandhi, and J. M. Shah, “Docker swarm and kubernetes in cloud computing environment,” in *Proceedings of the International Conference on Trends in Electronics and Informatics (ICOEI 2019).* Piscataway, NJ: IEEE, 2019, pp. 179–184.

[17] B. Kang, J. Jeong, and H. Choo, “Docker swarm and kubernetes containers for smart home gateway,” *IT Professional*, vol. 23, no. 4, pp. 75–80, 2021.

[18] Kubernetes, “Daemonset: Kubernetes documentation,” 2022-08-31. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ (Visited on 2022-11-08)

[19] ——, “Configmaps: Documentation,” 2022-10-24. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/ (Visited on 2022-11-08)

[20] ——, “The kubernetes api: Documentation,” 2022-10-24. [Online]. Available: https://kubernetes.io/docs/concepts/overview/kubernetes-api/ (Visited on 2022-11-08)

[21] GitHub, “Version 3.1.0 of the openapi-specification: Documentation: Github - oai/openapi-specification,” 2023-01-10. [Online]. Available: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md (Visited on 2023-01-10)

[22] Suse Rancher Community, “Comparing kubernetes cni providers: Flannel, calico, canal, and weave,” 2023-02-12. [Online]. Available: https://www.suse.com/c/rancher_blog/comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/ (Visited on 2023-02-12)

[23] “Flannel backends documentation,” 2023-02-12. [Online]. Available: https://github.com/flannel-io/flannel/blob/master/Documentation/backends.md (Visited on 2023-02-12)

[24] “Project calico,” 2023-02-10. [Online]. Available: https://www.tigera.io/project-calico/ (Visited on 2023-02-13)

[25] Kubernetes, “Adding windows nodes: Kubernetes - documentation (archived / v1.23),” 2022-04-19. [Online]. Available: https://v1-23.docs.kubernetes.io/docs/tasks/administer-cluster/kubeadm/adding-windows-nodes/ (Visited on 2022-11-12)

[26] ——, “Kubernetes is moving on from dockershim: Commitments and next steps,” 2022. [Online]. Available: https://kubernetes.io/blog/2022/01/07/kubernetes-is-moving-on-from-dockershim/ (Visited on 2022-11-02)

[27] “Introducing the host compute service (hcs): Blog post, archived technet article,” 2017. [Online]. Available: https://techcommunity.microsoft.com/t5/containers/introducing-the-host-compute-service-hcs/ba-p/382332 (Visited on 2023-02-27)

[28] GitHub, “Guide for adding windows node: Rbac config not found: Issue #261 - kubernetes-sigs/sig-windows-tools,” 2023-02-02. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/261 (Visited on 2023-02-02)

[29] ——, “Windows node with containerd can’t run flannel and kubeproxy daemonsets: Issue #128 - kubernetes-sigs/sig-windows-tools,” 2023-02-02. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/128 (Visited on 2023-02-02)

[30] amaltinsky, "Windows containers issue: Windows server 2022 container on windows 10 (10.0.19044) host." 2022. [Online]. Available: https://github.com/microsoft/Windows-Containers/issues/258 (Visited on 2022-10-18)

[31] 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. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/277 (Visited on 2023-02-02)

[32] Mattbriggs, "Windows container base images: Microsoft documentation." [Online]. Available: https://learn.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/container-base-images (Visited on 2022-11-14)

[33] GitHub, "Mongo_initdb_root_username not working with nanoserver image: Discussion #599 - docker-library/mongo," 2023. [Online]. Available: https://github.com/docker-library/mongo/discussions/599 (Visited on 2023-03-17)

[34] "Conditional compilation: The rust reference," 2023-02-09. [Online]. Available: https://doc.rust-lang.org/reference/conditional-compilation.html (Visited on 2023-02-12)

[35] Kubernetes, "Container runtimes: Documentation," 2019. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/container-runtimes/ (Visited on 2022-12-28)

[36] ArchWiki, "cgroups," 2023. [Online]. Available: https://wiki.archlinux.org/title/cgroups (Visited on 2023-03-16)

[37] ManKier, "Containerfile format: automate the steps of creating a container image." [Online]. Available: https://www.mankier.com/5/Containerfile (Visited on 2023-02-23)

[38] Microsoft, "Windows server core container image," 2022-11-27. [Online]. Available: https://hub.docker.com/_/microsoft-windows-servercore?tab=description (Visited on 2022-11-27)

[39] Brasmith-ms, "Versionskompatibilität von windows-containern." [Online]. Available: https://learn.microsoft.com/de-de/virtualization/windowscontainers/deploy-containers/version-compatibility?tabs=windows-server-2022%2Cwindows-11#mitigation---using-node-labels-and-nodeselector (Visited on 2023-03-01)

[40] "Building windows server multi-arch images: Kubernetes engine documentation," 2023-03-13. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/tutorials/building-windows-multi-arch-images (Visited on 2023-03-13)

[41] GitHub, "As a developer i want to be able to build windows containers with buildkit: Issue #34 - microsoft/windows-containers," 2022-12-25. [Online]. Available: https://github.com/microsoft/Windows-Containers/issues/34 (Visited on 2023-01-02)

[42] Kubernetes, "Windows containers in kubernetes: Documentation," 2022-10-15. [Online]. Available: https://kubernetes.io/docs/concepts/windows/intro/ (Visited on 2022-10-31)

[43] ——, "Creating a cluster with kubeadm: Documentation," 2022-11-07. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/ (Visited on 2023-03-14)

[44] GitHub, "Kube proxy container on windows fails to bootup: Issue #235 - kubernetes-sigs/sig-windows-tools," 2022. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/235 (Visited on 2023-03-12)

[45] Kubernetes, "Guide for scheduling windows containers in kubernetes," 2022-12-05. [Online]. Available: https://kubernetes.io/docs/concepts/windows/user-guide/ (Visited on 2023-03-17)

[46] 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. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/277 (Visited on 2023-03-13)