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

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

Dave Barry

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Abstract

Faculty 2 - Computer Science and Engineering
Allgemeine Informatik Master

Master of Science

Containerized multi-level deployment for a distributed adaptive microservice application

by Tim Wißmann

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

Acknowledgements

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

Contents

Abstract ii

Acknowledgements iii

Contents iv

List of Figures vii

List of Tables ix

Abbreviations x

1 Introduction 1
- 1.1 Scope . . . 1
- 1.2 Intended audience . . . 1
- 1.3 Outline . . . 1

2 Background and related work 2
- 2.1 Application under study . . . 2
- 2.2 Baseline architecture . . . 2
- 2.3 Network traffic encryption . . . 6
  - 2.3.1 Transport Layer Security . . . 7
  - 2.3.2 Mutual authentication . . . 8
  - 2.3.3 Certificate creation . . . 8
- 2.4 Problem statement . . . 10
- 2.5 Limitations . . . 10
- 2.6 Related Work . . . 11

3 System design 12
- 3.1 Orchestration engine . . . 12
  - 3.1.1 Hyper-V Replication . . . 12
  - 3.1.2 Docker Swarm . . . 13
  - 3.1.3 Kubernetes . . . 13
- 3.2 Kubernetes . . . 13
  - 3.2.1 Entities . . . 14
  - 3.2.2 Services . . . 14
  - 3.2.3 Pod life cycle . . . 16

3.2.4 Cluster networking . . . 17
3.3 Container environment . . . 18
3.3.1 ContainerD . . . 19
3.3.2 Docker Container Runtime Interface . . . 20
3.4 Container Image . . . 20
3.4.1 Base image . . . 21
3.4.2 Custom image . . . 21
3.5 Target architecture . . . 21

4 Implementation 22
4.1 Containerization of Services . . . 22
4.1.1 Code changes on core application . . . 22
4.1.2 Container definition . . . 27
4.2 Cluster Setup . . . 28
4.2.1 Setting up the master node . . . 29
4.2.1.1 Installing prerequisites and ContainerD . . . 29
4.2.1.2 Applying a Container Network Interface . . . 31
4.2.1.3 Adding the proxy daemon sets . . . 31
4.2.2 Setting up the worker node . . . 32
4.3 Automatic setup . . . 33
4.3.1 Cluster design . . . 34
4.3.2 Deployment of the application . . . 36

5 Results 38
5.1 Containerization . . . 38
5.1.1 Container manifest . . . 38
5.1.2 Windows Base Image . . . 39
5.1.3 Local images namespace . . . 40
5.2 Kubernetes . . . 41
5.2.1 System requirements . . . 41
5.2.2 Documentation . . . 42
5.3 Interfaces within the abstraction layers . . . 43
5.3.1 Version interoperability . . . 43
5.3.2 Error handling . . . 43
5.3.3 Database server . . . 44
5.4 Analysis . . . 44

6 Conclusion and future work 45
6.1 Lessons learned . . . 45
6.2 Conclusion . . . 45
6.3 Future work . . . 46

A Overview about submitted code 48

Acronyms 50

Bibliography **51**

List of Figures

2.1 The OpenTwin (OT) project overview. 3
2.2 The OT login screen. 3
2.3 A opened project inside OT with a few created geometric models and subtracted computation. 4
2.4 Communication overview and service organization for OT main services. In 1.1 the Local Session Service (LSS) registers at Global Session Service (GSS). As soon as the User Interface (UI) front end connects to the GSS (2.1), service information is exchanged (2.2) and the user is authenticated (2.3). As a consequence, the GSS creates a new session and tells the LSS to spawn new compute services. From now on the UI front end communicates directly with the Compute services via the Relay Service over a web socket connection. 6
2.5 Service initialization of OT processes. In the beginning, the main services GSS, Authorization Service (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.) 7
2.6 Simplified visualization of the Transfer Layer Security (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. 8
2.7 Simplified visualization of the mutual Transfer Layer Security (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. 9

3.1 Core and compute services for Kubernetes[3] 15
3.2 Abstraction layers for ContainerD on Windows. The image shows the technology stack from the Docker and Kubernetes (K8s) command line to the Container layer.[4] 19

4.1 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 Virtual Machine (VM) to have another isolation layer and full admin privileges on the worker node. 35
4.2 Cluster design scenario where worker node is running on a dedicated machine inside the protected University network. 36
4.3 Cluster design scenario where master and worker node were both running on a dedicated physical machine each. 36

List of Tables

2.1 List of compute services and their corresponding tasks. 5

3.1 List of K8s states during pod life cycle[5]. 16

A.1 List of added files in the code repository and their usage description. 49

Abbreviations

e.g.	**e**xempli **g**ratia

Chapter 1

Introduction

This thesis is about the deployment of a microservice application.

ContainerD is a relatively new container runtime and replaces the runtime of Docker (mostly referred to as Docker). The research field of cluster deployment is still in the beginning state for Microsoft® Windows® (Windows) in relationship to ContainerD.

1.1 Scope

1.2 Intended audience

1.3 Outline

Chapter 2

Background and related work

2.1 Application under study

OpenTwin (OT) is a 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 weak computers can run the application.

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

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

2.2 Baseline architecture

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

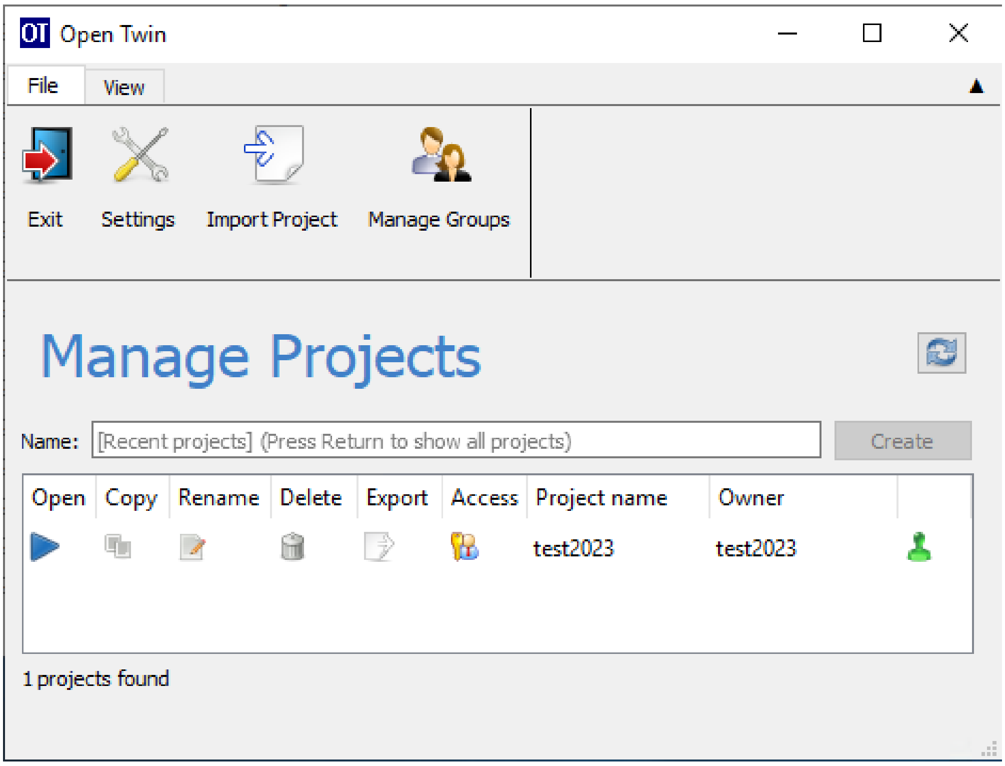

FIGURE 2.1: The OT project overview.

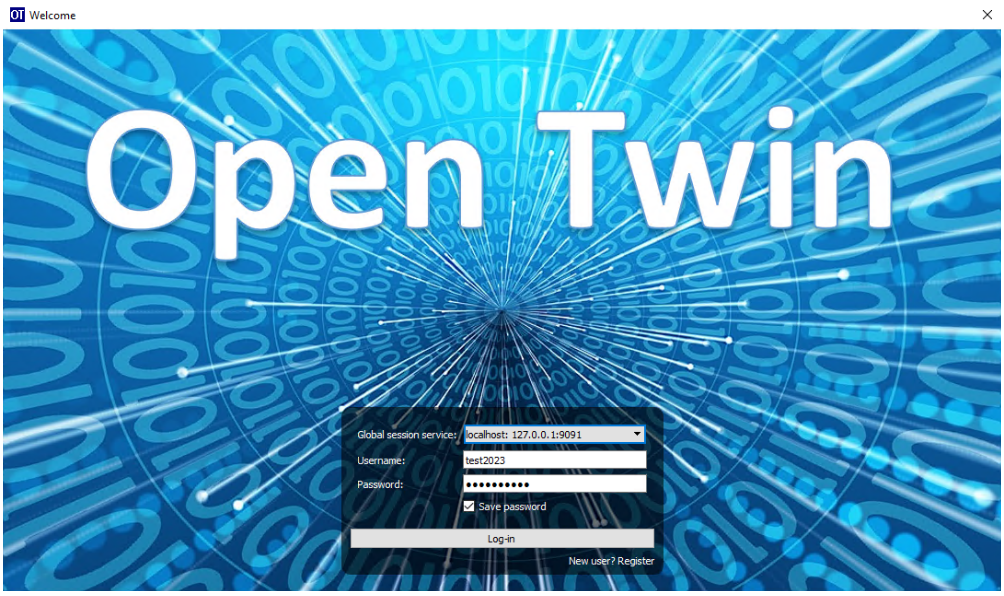

FIGURE 2.2: The OT login screen.

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_

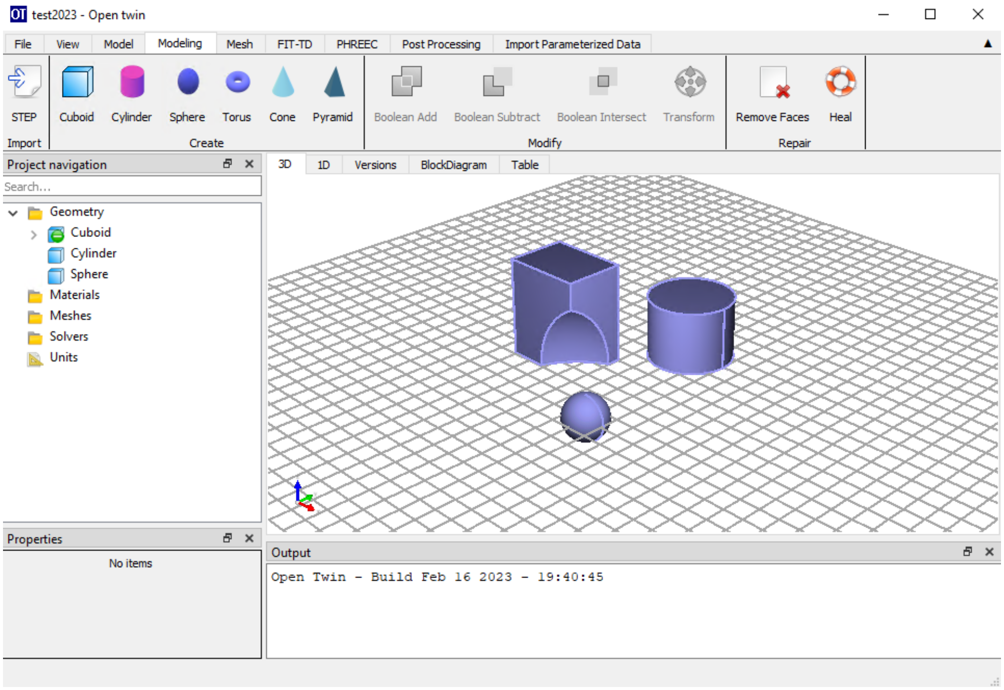

FIGURE 2.3: A opened project inside OT with a few created geometric models and subtracted computation.

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

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

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

a project that can dynamically spawn and exit. A 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.

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

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

1. The UI front end requests further service information from the publicly available GSS. The address for this service is provided by the user. The GSS 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.

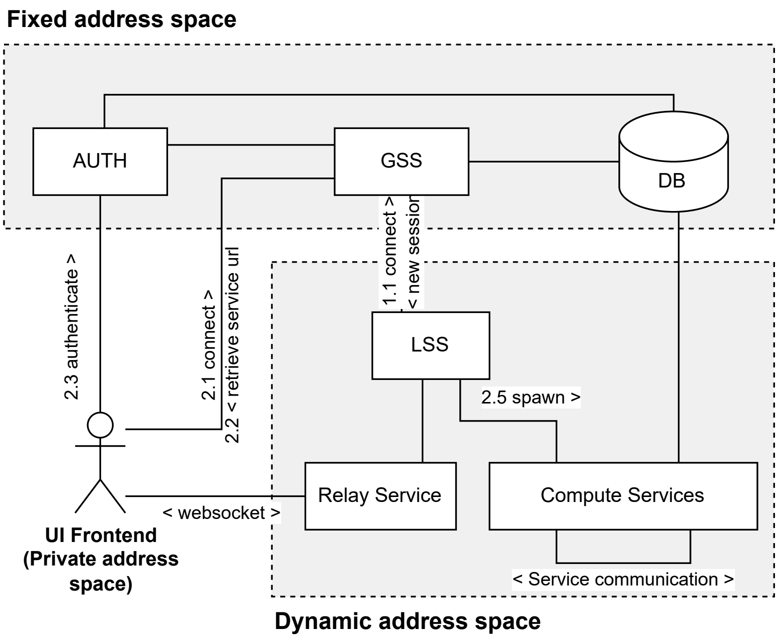

FIGURE 2.4: Communication overview and service organization for OT main services. In 1.1 the LSS registers at GSS. As soon as the UI front end connects to the GSS (2.1), service information is exchanged (2.2) and the user is authenticated (2.3). As a consequence, the GSS creates a new session and tells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

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

4. Once a project is opened or created, the UI front end requests a new session from the GSS. The GSS replies with the connection URLs 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 web socket.

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

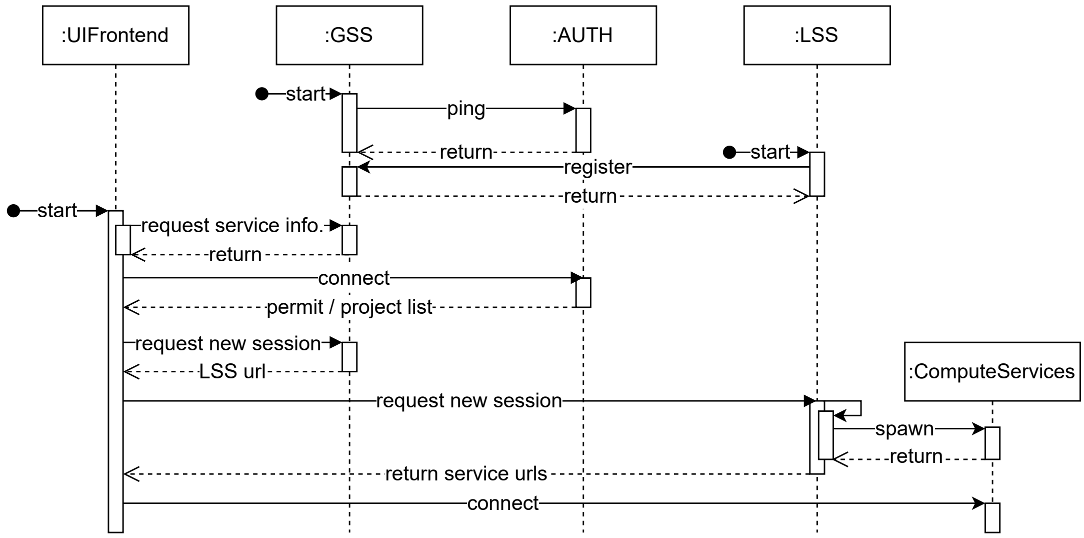

Figure 2.5: Service initialization of OT processes. In the beginning, the main services GSS, AUTH and an optional LSS are initialized. While the GSS checks the reachability of AUTH, the LSS registers itself at the GSS. After starting the UI front end, the service information is requested from a GSS and 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.)

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.

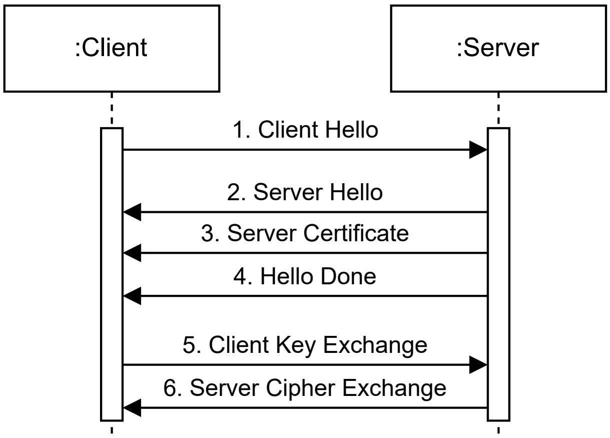

FIGURE 2.6: Simplified visualization of the TLS handshake[1]. After initialization of the handshake (1,2), the server sends its certificate (3) and finishes with a message "Hello Done" (4). The client then validates the certificate and compares it against the chain of trust. Afterwards, the key exchange 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?]. 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].

2.3.3 Certificate creation

For creation of certificates in the application landscape of OT, CloudFlare's public key infrastructure toolkit "cfssl"[2] 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

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

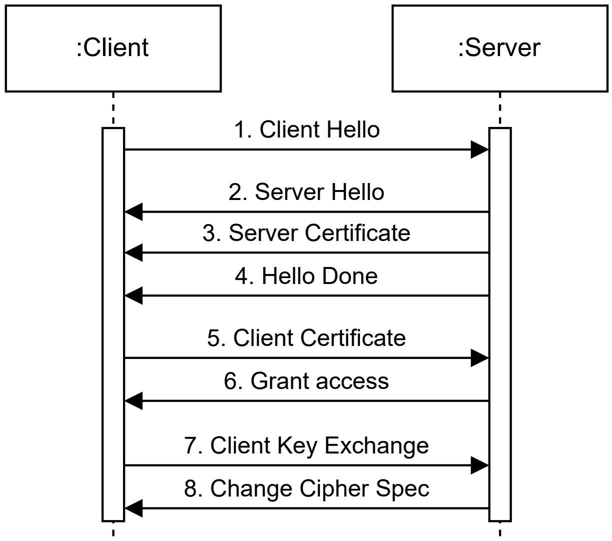

FIGURE 2.7: Simplified visualization of the mTLS handshake[1, 2]. After initialization of the handshake (1,2), the server sends its certificate (3) and finishes with a message “Hello Done” (4). The client then validates the certificate and compares it against the locally stored root authority. Afterwards, 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.

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.

```
{
  "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 Problem statement

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

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

2.5 Limitations

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

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

2.6 Related Work

Chapter 3

System design

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

3.1 Orchestration engine

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

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 a 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 harder to configure in the beginning.

The engine of choice was K8s because of its rich feature set. Also studies showed that K8s outperforms Docker Swarms when it comes to performance. For example, Marathe et. al. [6] 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. [7] 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 [7]. Since they focused on container distribution and management methods this might get handy in the use case scenario under study.

3.2 Kubernetes

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

3.2.1 Entities

There are many entities for objects inside the cluster. For description of those entities the configuration language YAML[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 Internet Protocol (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[8]. 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[9]. Data stored in a configuration map can be mounted as volume, environment variable or command line argument to make applications more portable[9].

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

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

3.2.2 Services

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

In the following paragraphs, the crucial services are described in detail. Since every service is a pod, they can have multiple replicas. Only the core service have to run on a dedicated Linux

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

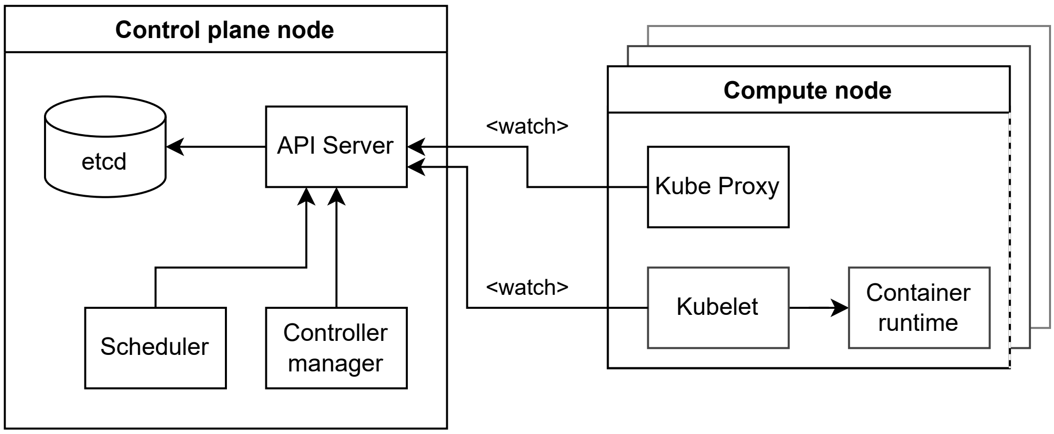

FIGURE 3.1: Core and compute services for Kubernetes[3]

node the so called "control plane node" or "master node". The underlying OS must be Linux, because Windows is not supported as control plane. The other services can run on nodes (with any OS) for executing the applications and perform computations ("compute node" or "worker node").

etcd The etcd[2] database server is a key-value store designed for distributed systems[3]. That means it could run with multiple replications and would still be able to keep a persistent storage synchronized across multiple instances. It contains the applied configuration of several cluster entities (e.g. User configurations, deployments, pod configurations).

API server This is a RESTful web server that serves the Kubernetes API via HTTP[10]. It is the central joint between the services and establishes communication between users, external components and other core services. It makes the objects stored in etcd accessible over an Open API specification[3, 11] and allows observing changes on the entities. The Command line interface (CLI) tools "kubectl" and "kubeadm" both interact with the API server.

Kubelet Kubelet is the service on the 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[3]. For a given IP address and port combination

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

it assures the connection to the corresponding pod. If multiple pods can offer a service, the proxy also acts as a load balancer[3].

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[3].

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[3]. 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[5]. After creation on the cluster, a unique identifier is assigned before a pod gets scheduled to an available node[5]. The pod keeps alive until its termination or deletion[5]. For distinguishing different kind of states of a pod life cycle, K8s defines the pod states as described in Table 3.1.

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[5].

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 back-off delay (10s, 20s, 40s, ...), that is capped at five minutes"[5]. Furthermore, it is explained that the back-off time gets reset, once a container keeps running for 10 minutes[5].

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 Kubernetes directly. By using a CNI the automated generation of network addresses and their inclusion is achieved when new containers are create or destroyed. It is crucial that pods share the same subnet across all the nodes in a cluster and Network Address Translation (NAT) is avoided[3].

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][12]. 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[13]. VXLAN is an overlay protocol and encapsulates layer 2 Ethernet frames within datagrams[12]. It is similar to regular VLAN, but offers more than 4,096 network identifiers[12]. Thus, VXLAN is a good choice for highly scalable systems.

Even though, the team behind K8s do 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[12, 14]. Calico comes with a sophisticated access control system[14] and more configuration options. However, its advanced configuration makes it hard to maintain long-term.

[3] Flannel: https://github.com/flannel-io/flannel
[4] CoreOS: https://getfedora.org/en/coreos
[5] Tigera’s Calico: https://www.tigera.io/project-calico/

For this use case, Flannel is used as network plugin, since it is the described plugin used in the documentations for setting up K8s with Windows containers[15, 16]. Hence, support for this CNI plugin in relation with Windows containers is assumed to be larger than with 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 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 actual container runtime runs the isolation layer between the physical host machine and the K8s cluster by using containerization of processes. This is what can be selected when working with K8s.

While K8s used to support Docker as their standard container runtime, they announced it to be deprecated in 2020, and finally removed the support in February 2022[17, 18]. 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.

At the current time being, container networking with ContainerD is not well-established on Windows [19–22] even though the docker runtime is already removed in current versions of K8s [17]. However, these are the only two working container backends for Windows containers. Therefore ContainerD as container backend was chosen.

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

3.3.1 ContainerD

ContainerD is a native version of a container runtime. Newer versions of Docker on Linux, are running ContainerD under the hood for process isolation. On Windows, ContainerD uses slim host process isolation. The process isolation with ContainerD consists of multiple abstraction layers (shown in Figure 3.2). 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*[4]. Using *runc*, new containers can be created by running a simple command[4]. 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[23]. 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[17]. Since version v1.23 of K8s, Docker was fully removed which lead to ContainerD being the only available CRI for Windows containers.

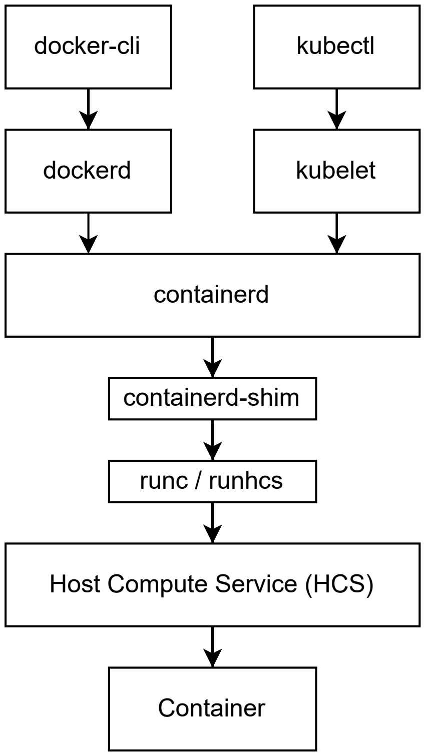

FIGURE 3.2: Abstraction layers for ContainerD on Windows. The image shows the technology stack from the Docker and K8s command line to the Container layer.[4]

3.3.2 Docker Container Runtime Interface

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

For Windows, there are two different modes available. The first option is using the process isolation mode offered by ContainerD. It can be enabled by switching to "Windows Containers". It is the default mode for Windows Server systems. However, older versions of Docker and Docker on Windows 10 and above have the opposite behavior[24]. On client versions of Windows, a dedicated hypervisor-isolated virtualization is the default option[24]. This means, during the installation of Docker on Windows, the underlying Windows container host creates a separate Hyper-V Virtual Machine (VM) to run container images. However, this is not a regular Hyper-V VM. Instead, it is a purpose-built VM, often referred to as utility VM or UVM that can't be managed directly and is fully controlled by the Windows container runtime[24]. 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[24].

3.4 Container Image

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

Container images are described, using the Containerfile[9] format. There are container images for each process of the system architecture, each of them having their own Containerfile definition with different command line arguments and environment variables.

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

3.4.1 Base image

The container images to use for running the OpenTwin processes need to run a Windows base image. Beside the full Windows images, Microsoft® offers the more common images "Windows Server Core" and "Windows Nanoserver"[25]. They significantly differ in the download size, their on-disk footprint and the features supported[25]. As Microsoft states, "Nanoserver was built to provide just enough API surface to run apps that have a dependency on .NET core or other modern open source frameworks. PowerShell, Windows Management Instrumentation, and the Windows servicing stack are absent from the Nanoserver image"[25].

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

Chapter 4

Implementation

4.1 Containerization of Services

Container images are provided for running the application inside the CRI. Definition of the container manifest is done in the “Containerfile”[1] format.

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 return code indicates a successful termination. Even though the 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.

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

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

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.

```
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. 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 a 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 windows subsystem.

```
#![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[26]. 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

[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.

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 error messages are not able to be read conveniently in cases where the application crashes. 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 invented to the batch file for 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 with higher verbosity. As can be seen in Listing 4.4, with “/V” appended, two new variables “`pause_prefix`” and “`pause_suffix`” are set (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. Thus, the user is now able to read error messages if application crashes occur.

Enhanced logging and error tracing

For improving the error tracing, the overall log amount has been increased. This involves enabling the logging of the central logging functions inside the library “OpenTwinCommunication”. The logging mechanism did not use logging to standard output. Instead, calls to the respective functions for logs were just ignored. This was changed and logging to standard output has been introduced.

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

LISTING 4.6: Comparison of the set log level with the log severity of the message. If the converted log severity is less than the defined log level, the message is ignored. (*/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.

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.

For the current time being, there is an ongoing work to replace the logging to standard output by forwarding the log lines to a central logging library.

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. (*/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 image 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.

```
150 let service_port = Url::parse(&format!("https://{}", service_url))
151 .expect(&format!("Invalid service url. Unable to parse service url: {}", service_url))
152 .port();
153 if service_port.is_none() {
154         panic!("Invalid service url. Service url is lacking port defintion: {}",
    service_url);
155 }
156 let binding_address = format!("0.0.0.0:{}", service_port.unwrap().to_string());
157 let listener = net::TcpListener::bind(&binding_address).await?;
158 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.

```
21 ENTRYPOINT ["cmd", "/C"]
22 CMD ["open_twin.exe", "SessionService.dll", "0", \
23 "%OPEN_TWIN_LSS_SERVICE_ADDRESS%:%OPEN_TWIN_LSS_PORT%", \
24 "%OPEN_TWIN_GSS_SERVICE_ADDRESS%:%OPEN_TWIN_GSS_PORT%", \
25 "%OPEN_TWIN_AUTH_PORT%"]
26 EXPOSE 8093
27 WORKDIR C:/app/Deployment/Certificates
28 COPY ./ ../
29 RUN createCertificate.bat && certutil -addstore root %OPEN_TWIN_CA_CERT%
30 WORKDIR C:/app/Deployment
31 RUN C:\app\Deployment\VC_Redist\VC_redist.x64.exe /install /quiet
```

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 to C++ libraries that are not present in the base image, the Microsoft Visual C++ Redistributable must be installed as well (line 31).

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 “v1.25.3”. 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[27] and the online tutorial of Basappa[28].

4.2.1.1 Installing prerequisites and ContainerD

After installing and setting up the OS, 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.

```bash
$ 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. 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 (*/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 4.13.

```bash
$ 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 4.13: 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 is added to the runtime options of ContainerD and its service is restarted. For this, the commands from Listing 4.14 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 4.14: 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. Then, the cluster can be initialized by running the command line tool as shown in Listing 4.15. 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 4.15: 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"[15]. Additionally, 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. Thus, the cgroup driver on Linux worker nodes should be aligned to the master node. The information passed as configuration is presented in Listing 4.16.

```
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 4.16: 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]. Since networking has to blend with Flannel on Windows, 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 must be performed. 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 4.17. 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 4.17: 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 Flannel. The Special Interest Group (SIG) "Windows Tools" of K8s provides the additional objects[15]. 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 4.18.

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

```
$ 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 4.18: Bash command for adding the flannel overlay configuration[15]

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[15]. Although there is an original guide from K8s[16], 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 4.19. 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.

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

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

```
> 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 4.19: Retrieval of node preparation scripts[15]

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. The token is copied to the preapared node and used as part of a “join” command. Listing 4.20 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 4.20: 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. 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.

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[15]. 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 4.21.

```
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 4.21: 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 firewall as shown in Listing 4.22. 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 4.22: 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 probed in three different scenarios. Starting point of the investigation was the most complex approach (Figure 4.1). 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 4.1, 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 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.

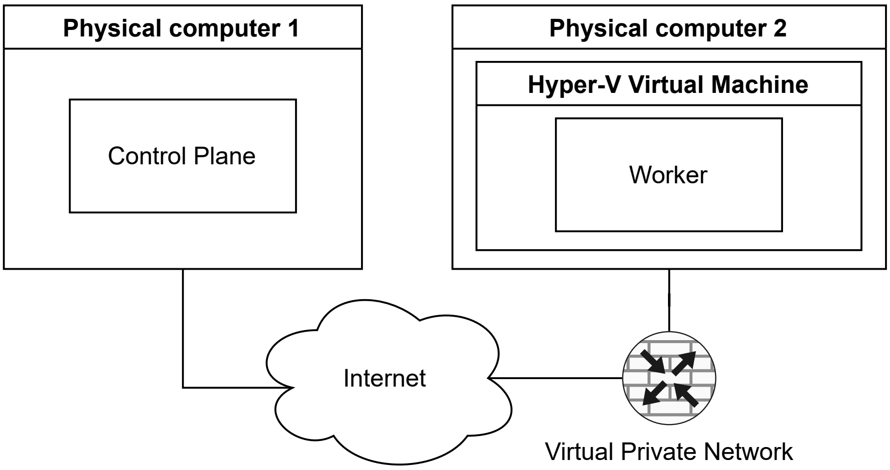

FIGURE 4.1: Cluster design scenario where worker node is running on a dedicated machine inside the protected University network. Additionally, the worker node is hosted inside a VM to have another isolation layer and full admin privileges on the worker node.

The problems in the first scenario is the additional abstraction layer between Hyper-V VM for administer the worker node and the physical machine. This 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 4.2. Here, the worker is not inside a 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 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

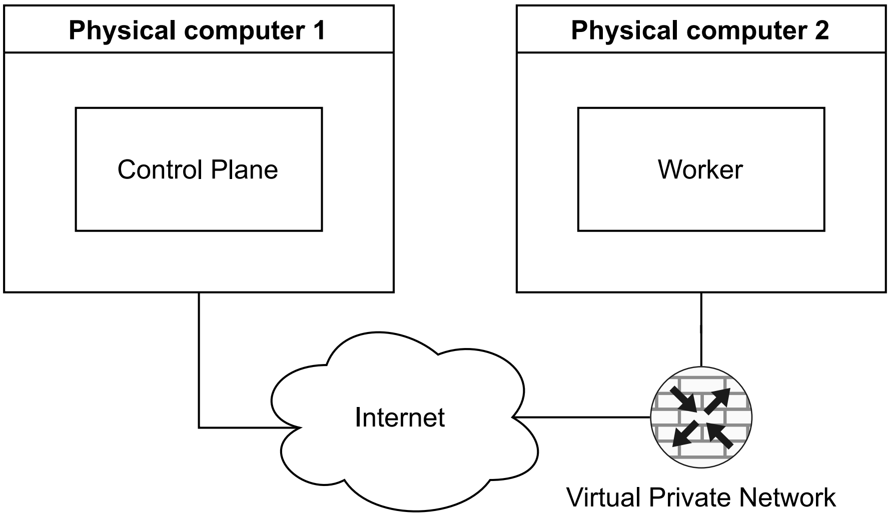

FIGURE 4.2: Cluster design scenario where worker node is running on a dedicated machine inside the protected University network.

subnet(s)". This was justified by the VPN network interface, since it has to support a large amount of subnets.

This leads to the third scenario, shown in Figure 4.3, which is the simplest. Here, the two physical computers are connected directly in the same network. Since this expects mostly the real world scenario this is the preferable one. In this scenario, the containers were able to start and got an IP address assigned.

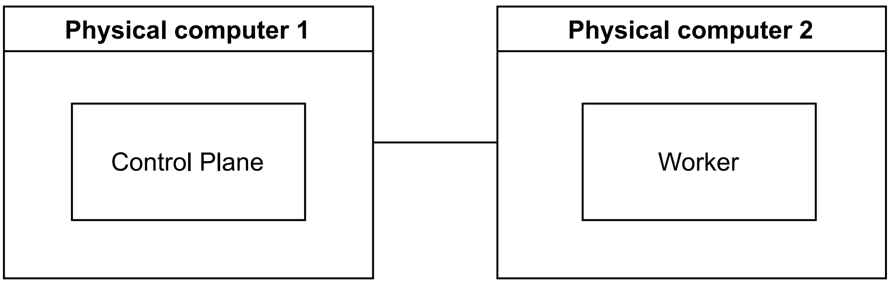

FIGURE 4.3: Cluster design scenario where master and worker node were both running on a dedicated physical machine each.

4.3.2 Deployment of the application

As a first draft, only the LSS of the application is deployed. Deployment of the application is done by applying the manifest as shown in Listing 4.23. 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 registry, the images are 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 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`".

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

Chapter 5

Results

The development of the K8s cluster and the containerization of OT hides several pitfalls. 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 everywhere[29] 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[30].

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

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

Listing 5.1 shows an example of the problem. While the entry point in Docker is interpreted as "`open_twin.exe Service.dll`", the interpreter in ContainerD only reads the first line as entry point and ignores the line break character "\". Therefore it results in just "`open_twin.exe`" as interpreted entry point. This causes troubles during build and execution of a container image. The container image is built in Docker, whereas it does not run in ContainerD. Since, ContainerD on Windows cannot build container images, it does also not work the other way around. To overcome this flaw, the *ENTRYPOINT* definition has to be written in JSON notation. If defined as in Listing 5.2 the interpreter

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

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

5.1.2 Windows Base Image

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[25, 31]. They also offer one general image tag for multiple architectures[31]. 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[32]. For this, a separate "`nodeSelector`" key is defined as shown in Listing 5.3.

```
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[33].

5.1.3 Local images namespace

If images are locally imported, 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[34]. 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 provide the necessary option. Thus the change of the namespace is required for locally imported 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[35], 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[15]. Furthermore, K8s also has the fixed requirement of the usage of "Windows Server 2019" and above[35]. This reduces the subset of compatible node OS, 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 became fundamental on the page "Adding Windows nodes"[16]. 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"[36]. 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. However, 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"[15] 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, 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"[15]. 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"[15]. 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 supported since 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[34].

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 stays not open after a crash, the output is hard to catch. In some cases throughout the code base exceptions are not caught or just 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.

5.3.3 Database server

- TODO: MongoDB server container image cannot be initialized easily on Windows

5.4 Analysis

- The Kube-Proxy does not work in combination with Flannel on Windows.

Chapter 6

Conclusion and future work

6.1 Lessons learned

6.2 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 Frontend 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. Furthermore, the general product quality was improved.

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 understanding. 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.

6.3 Future work

The results showed that there are still some points open. 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.

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 OT performs 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	2
Windows	Microsoft® Windows®	1
OS	Operating System	5
cgroup	control group	18
SIG	Special Interest Group	31
DLL	Dynamic Link Library	3
URL	Uniform Resource Locator	5
UI	User Interface	2
TLS	Transfer Layer Security	6
mTLS	mutual Transfer Layer Security	7
CA	Certificate Authority	7
CSR	Certiticate Request	8
GSS	Global Session Service	4
AUTH	Authorization Service	4
LSS	Local Session Service	4
HTTP	Hypertext transfer protocol	7
API	Application Programming Interface	13
CLI	Command line interface	15
VPN	Virtual Private Network	35
K8s	Kubernetes	13
IP	Internet Protocol	14
NAT	Network Address Translation	17
VNI	Virtual Network Identifier	31
CNI	Container Network Interface	17
CRI	Container Runtime Interface	18
NSSM	Non-Sucking Service Manager	32
VM	Virtual Machine	20
HCS	Host Compute Service	19

Bibliography

[1] *The Transport Layer Security (TLS) Protocol Version 1.3*, Request for Comments Std. 8446, 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.html

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

[3] 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

[4] 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)

[5] "Pod lifecycle: Documentation," Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/ (Visited on 2022-11-08)

[6] 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.

[7] 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.

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

[9] "Configmaps: Documentation," Kubernetes, 2022-10-24. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/ (Visited on 2022-11-08)

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

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

[12] 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)

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

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

[15] "Guide for adding windows node: Documentation," Kubernetes SIG Windows Tools, 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)

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

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

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

[19] "Guide for adding windows node: Rbac config not found: Issue #261 - kubernetes-sigs/sig-windows-tools," Kubernetes SIG Windows Tools, 2023-02-02. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/261 (Visited on 2023-02-02)

[20] "Windows node with containerd can't run flannel and kubeproxy daemonsets: Issue #128 - kubernetes-sigs/sig-windows-tools," Kubernetes SIG Windows Tools, 2023-02-02. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/128 (Visited on 2023-02-02)

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

[22] "kube-flannel for linux fails to (re)start after rbac for kube-flannel for windows is applied: Issue #277 - kubernetes-sigs/sig-windows-tools," Kubernetes SIG Windows Tools, 2023-02-02. [Online]. Available: https://github.com/kubernetes-sigs/sig-windows-tools/issues/277 (Visited on 2023-02-02)

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

[24] 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

[25] 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)

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

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

[28] 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)

[29] S. C. Gadde, "failed to create containerd task: hcsshim::createcomputesystem kube-proxy: The directory name is invalid." [Online]. Available: https://stackoverflow.com/questions/74799620/kubernetes-windows-worker-node-addition-failed-to-create-containerd-task-hcss

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

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

[32] Brasmith-ms, "Versionskompatibilität von windows-containern," Microsoft. [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)

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

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

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

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