

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

MASTER'S THESIS

Containerized multi-level deployment for a distributed adaptive microservice application

Author:

Tim WIGMANN

Supervisors:

Prof. Dr. Peter THOMA

Prof. Dr. Eicke GODEHARDT

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

in the course

Allgemeine Informatik Master

March 17th, 2023

Declaration of Authorship

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

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

Signed:

Date:

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

Dave Barry

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Abstract

Faculty 2 - Computer Science and Engineering

Allgemeine Informatik Master

Master of Science

Containerized multi-level deployment for a distributed adaptive microservice application

by Tim WILDMANN

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

Declaration of Authorship	i
Abstract	iii
Acknowledgements	iv
Contents	v
List of Figures	vii
List of Tables	viii
Abbreviations	ix
1 Introduction	1
1.1 Scope	1
1.2 Intended audience	1
1.3 Limitations	1
1.4 Outline	1
2 Background and related work	2
2.1 Baseline architecture	2
2.2 Problem statement	3
2.3 Related Work	4
3 System design	5
3.1 Applied technologies	5
3.1.1 Orchestration engine	5
Hyper-V Replication	6
Docker Swarm	6
Kubernetes	6
3.1.2 Kubernetes in Detail	6
3.1.2.1 Entities	7
Deployment	7
Pod	7
Daemon set	7
User	7

Node	7
3.1.2.2 Services	7
etcd	8
API server	8
Kubelet	8
Kube Proxy	8
Scheduler	8
Controller manager	8
3.1.2.3 Cluster networking	9
flannel	9
calico	9
3.1.2.4 Container environment	9
docker-cri	9
containerd	9
LXC container	9
3.2 Target architecture	9
4 Implementation	10
4.1 Cluster Setup	10
4.1.1 Creating the master node	10
Installing a Container Network Interface	11
Adding the proxy daemonsets	11
4.1.2 Creating the worker node	11
4.2 Containerization of Application	12
5 Discussion	13
5.1 Analysis	13
5.2 Dunno whata write yet...	13
6 Conclusion and future work	14
6.1 Future work	14
6.2 Conclusion	14
A Appendix Title Here	15
Bibliography	16

List of Figures

3.1	Core and compute services for Kubernetes	7
-----	--	---

List of Tables

Abbreviations

LAH List Abbreviations Here

Chapter 1

Introduction

1.1 Scope

1.2 Intended audience

1.3 Limitations

1.4 Outline

Chapter 2

Background and related work

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

Each microservice of the application is included dynamically and linked as a Dynamic Link Library (DLL) file. For starting the microservice environment, a central executable ("open_twin.exe") is started with the corresponding arguments for the services (like binding address, port numbers, encrypted passwords) and the path to the DLL file itself. The UI frontend, which is started by the user directly, is compiled in its own executable ("uiFrontend.exe"). 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. The system consists of the following microservices that are permanently accessible: Global Session Service (GSS), Authorization Service (AUTH) and the database. The database is running on MongoDB¹. Another Service is the Local Session Service (LSS) that spawns the so called application services. Those are services for logging, scripting, 3D modeling, kriging and so on. While GSS, AUTH and database are globally accessible, the LSS can theoretically run

¹MongoDB: <https://www.mongodb.com/>

on a dedicated host and is only communicated to other parties after it has registered itself to the [GSS](#).

Once started, the user can login. In order to connect to the database, the following steps are performed:

1. The User Interface ([UI](#)) frontend requests further service information from the [GSS](#). It responds with Uniform Resource Locations ([URLs](#)) to the database and the [AUTH](#).
2. The [UI](#) frontend connects to the [AUTH](#) using the authentication information provided by the user.
3. If the [AUTH](#) replies with a positive authentication, the [UI](#) frontend connects to the database and lists the projects.
4. Once a project is opened or created, the [UI](#) frontend requests a new session from the [GSS](#). The [GSS](#) replies with the connection [URLs](#) from the [LSS](#). The [LSS](#) has been registered to the [GSS](#) during its initialization.
5. The [UI](#) frontend 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](#) frontend communicates with the application services.

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

2.2 Problem statement

Even though, the application is clearly based on a microservice architecture, it is not able to run on a distributed system yet. It consists of multiple processes that have to run on the same system and need a full working operating system as baseline. Containerization of the system has never been tested (!!erprobt!!) and needs to be introduced. Additionally, the automatic

service extension requires communication between the cluster orchestration management and the applications running on the nodes.

Regarding logging, while the frontend application does, the microservices currently do not produce log files. Instead, all sub processes write the information on its standard output stream.

2.3 Related Work

Chapter 3

System design

3.1 Applied technologies

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

3.1.1 Orchestration engine

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

The engine of choice was Kubernetes ([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. [1] 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. [2] 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 [2]. Since they focused on container distribution and management methods this might get handy in the use case scenario under study.

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

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 API, using this orchestration technology involves the risk of inflexibility later on ("vendor lock-in").

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

3.1.2 Kubernetes in Detail

Since [K8s](#) is the chosen orchestration engine, the following sections are taking a deeper look inside its architecture.

3.1.2.1 Entities

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

Deployment Deployments are used to define declarative states for Pods.

Pod A pod represents a set of running containers on a node.

Daemon set

User

Node A node represents a physical machine inside the cluster.

3.1.2.2 Services

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

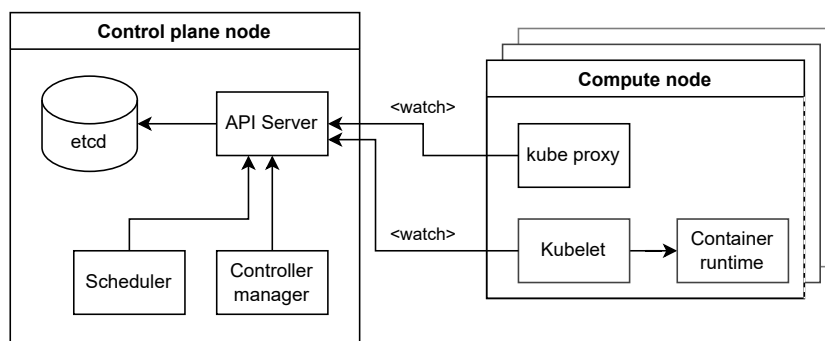


FIGURE 3.1: Core and compute services for Kubernetes

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

etcd The etcd¹ 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[?]. 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?] and allows observing changes on the entities. The **cli!** (**cli!**) tools "kubectl" and "kubeadm" both interact with the API server.

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

Kube Proxy The Kube-Proxy runs as a separate pod on every compute node. It maintains the connectivity between the services and pods[3]. For a given IP address and port combination it assures the connection to the corresponding pod. If multiple pods can offer a service, the proxy also acts as a load balancer[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, operating system, 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.

¹etcd: <https://etcd.io/>

3.1.2.3 Cluster networking

Cluster network is achieved using two components: The network overlay and the **CNI!** (**CNI!**).

flannel - noch nicht vollständig ausgereift

calico -> bevorzugt?

3.1.2.4 Container environment

As container environment `containerd` is used. `Containerd` is.. In comparison to other container environments this is ...

At the current time being, container networking with `containerd` is not well-established on Windows [4-7] even though `docker-cri` is already removed in current versions of [K8s](#) [8]. However, these are the only two working container backends for Windows containers. Therefore `containerd` as container backend was chosen.

docker-cri

containerd - container runtime which is used in `docker` -> Only tech with support for Windows - equal tech stack everywhere

LXC container linux containers

3.2 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 Cluster Setup

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

4.1.1 Creating the master node

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

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

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

LISTING 4.1: Bash command for setting up containerd config

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

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

LISTING 4.2: Bash command for setting up the cluster

Installing a Container Network Interface After successfully running the initialization, the cluster overlay network `flannel` needs to be setup. This is required for working with Windows worker nodes. To setup `flannel` the respective pod description can be directly downloaded from the vendor¹. In the configuration the VNI (4096) and port (4789) for Flannel on Windows were set. Afterwards the configuration has been applied on the cluster. After linking `kubectl` to the local control plane node, the successful setup of the cluster can be checked with the `kubectl` command.

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

```
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/kube-proxy.yml |
kubectl apply -f https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/flannel
```

4.1.2 Creating the worker node

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

¹<https://raw.githubusercontent.com/flannel-io/flannel/master/Documentation/kube-flannel.yml>

After successfully setting up the **NAT!** and installing containerd as a service, the node will be finally prepared to host tasks. For this, another powershell script "PrepareNode"² from the Kubernetes Special Interest Group ([SIG](#)) was run. After running the script the resulting "StartKubelet" file needs to be changed to drop invalid arguments.

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

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

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

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

4.2 Containerization of Application

²<https://github.com/kubernetes-sigs/sig-windows-tools/releases/download/v0.1.5/PrepareNode.ps1>

Chapter 5

Discussion

5.1 Analysis

5.2 Dunno whata write yet...

Chapter 6

Conclusion and future work

6.1 Future work

- Linux port and cluster based on linux - Automated image building using Packer.io (for multiple platforms) - Ranger for streamlining kubernetes deployment

6.2 Conclusion

Appendix A

Appendix Title Here

Write your Appendix content here.

cgroup control group

SIG Special Interest Group

DLL Dynamic Link LibraryDynamic Link Libraries

URL Uniform Resource Location

UI User Interface

TLS Transfer Layer Security

mTLS mutual Transfer Layer Security

GSS Global Session Service

AUTH Authorization Service

LSS Local Session Service

K8s Kubernetes

Bibliography

- [1] Nikhil Marathe, Ankita Gandhi, and Jaimeel M. Shah. Docker swarm and kubernetes in cloud computing environment. In *Proceedings of the International Conference on Trends in Electronics and Informatics (ICOEI 2019)*, pages 179–184, Piscataway, NJ, 2019. IEEE. ISBN 978-1-5386-9439-8. doi: 10.1109/ICOEI.2019.8862654.
- [2] Byungseok Kang, Jaeyeop Jeong, and Hyunseung Choo. Docker swarm and kubernetes containers for smart home gateway. *IT Professional*, 23(4):75–80, 2021. ISSN 1520-9202. doi: 10.1109/MITP.2020.3034116.
- [3] Marko Lukša. *Kubernetes in action: Anwendungen in Kubernetes-Clustern bereitstellen und verwalten*. Hanser eLibrary. Hanser, München, 2018. ISBN 9783446456020. doi: 10.3139/9783446456020. URL <https://www.hanser-elibrary.com/doi/book/10.3139/9783446456020>.
- [4] GitHub. Guide for adding windows node: Rbac config not found: Issue #261 - kubernetes-sigs/sig-windows-tools, 2023-02-02. URL <https://github.com/kubernetes-sigs/sig-windows-tools/issues/261>.
- [5] GitHub. Windows node with containerd can't run flannel and kubeproxy daemonsets: Issue #128 - kubernetes-sigs/sig-windows-tools, 2023-02-02. URL <https://github.com/kubernetes-sigs/sig-windows-tools/issues/128>.
- [6] amaltinsky. Windows containers issue: Windows server 2022 container on windows 10 (10.0.19044) host., 2022. URL <https://github.com/microsoft/Windows-Containers/issues/258>.
- [7] GitHub. kube-flannel for linux fails to (re)start after rbac for kube-flannel for windows is applied: Issue #277 - kubernetes-sigs/sig-windows-tools, 2023-02-02. URL <https://github.com/kubernetes-sigs/sig-windows-tools/issues/277>.

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