BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

SETUP OF APPLICATION-COMPUTATION ON-PREMISE MINI-CLOUD BASED ON KUBERNETES

SESTAVENÍ APLIKAČNĚ-VÝPOČETNÍHO ON-PREMISE MINI-CLOUDU ZALOŽENÉHO NA KUBERNETES

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **SAMUEL STUCHLÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. KAMIL JEŘÁBEK**
VEDOUCÍ PRÁCE

BRNO 2020

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Stuchlý Samuel**
Program: Informační technologie
Název: **Sestavení aplikačně-výpočetního on-premise mini-cloudu založeného na Kubernetes**
Setup of Application-Computation On-Premise Mini-Cloud Based on Kubernetes
Kategorie: Počítačová architektura

Zadání:

1. Seznamte se s technologiemi pro kontejnerizaci a management nasazení kontejnerů používaném v cloudu, jako jsou Docker, Kubernetes a případně další dle doporučení vedoucího. Nastudujte technologie pro automatizaci správy IT infrastruktury, pro použití při nasazení výsledného řešení.
2. Nastudujte možnosti monitoringu takovéto infrastruktury a řešení správy ukládání dat na discích.
3. Po dohodě s vedoucím navrhněte strukturu aplikačně-výpočetního klastru za pomocí nastudovaných a vybraných technologií podle bodu 1 a 2 zadání.
4. Podle návrhu sestavte výsledný cluster, vytvořte instalační příručku a jednoduchou webovou stránku s popisem struktury a návodem k obsluze.
5. Ověřte funkčnost výsledného řešení při nasazení aplikace na této platformě.

Literatura:

- BERNSTEIN, David. Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing, 2014, 1(3): 81-84.
- VOHRA, Deepak. Kubernetes microservices with Docker. Apress, 2016.
- SAYFAN, Gigi. Mastering Kubernetes: Master the art of container management by using the power of Kubernetes. Packt Publishing Ltd, 2018.
- HOCHSTEIN, Lorin; MOSER, Rene. Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way. O'Reilly Media, Inc., 2017.
- BURNS, Brendan; BEDA, Joe; HIGHTOWER, Kelsey. Kubernetes. Dpunkt, 2018.
- VAYGHAN, Leila Abdollahi, et al. Deploying microservice-based applications with Kubernetes: experiments and lessons learned. In: 2018 IEEE 11th international conference on cloud computing (CLOUD). IEEE, 2018. s. 970-973.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Jeřábek Kamil, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 27. října 2020

Abstract

Kubernetes is a container orchestration platform for deployment and management of applications on a cluster. The goal of this thesis is to understand kubernetes and its components, and then design and set up an optimal kubernetes cluster architecture for a small kubernetes-based on-premise mini-cloud on the VUT University grounds. This Bachelor thesis explores basics of containers, container runtimes, container orchestration tools, Kubernetes architecture and its components and Ansible automation platform. It further includes description of designed architecture of the cluster, that will be implemented. Contribution of the this thesis resides in the architectural design of kubernetes cluster, that will be later installed on the university grounds and will be ready to use by university.

Abstrakt

Kubernetes je platforma na orchestráciu kontajnerov, na nasadenie a správu aplikácií v klastri. Cieľom tejto práce je porozumieť kubernetes a jeho komponentom a následne navrhnúť a sprevádzkovať optimálnu architektúru kubernetes klastru pre malý mini-cloud založený na kubernetes v areáli univerzity VUT. Táto bakalárska práca rozoberá základy kontajnerov, runtime kontajnerov, nástroje na orchestráciu kontajnerov, architektúru Kubernetes a jej komponenty a automatizačnú platformu Ansible. Ďalej obsahuje popis navrhnutej architektúry klastra, ktorá bude implementovaná. Príspevok tejto práce spočíva v návrhu architektúry kubernetes klastra, ktorý bude neskôr nasadený na pôde univerzity, pripravený na použitie.

Keywords

container, Docker, container orchestration, kubernetes, Ansible, cluster, mini-cloud, Kubespray

Kľúčové slová

kontejner, Docker, orchestrace kontejneru, Kubernetes, Ansible, klastr, mini-cloud, Kubespray

Reference

STUCHLÝ, Samuel. *Setup of Application-Computation On-Premise Mini-Cloud Based on Kubernetes*. Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Kamil Jeřábek

Setup of Application-Computation On-Premise Mini-Cloud Based on Kubernetes

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Kamil Jeřábek. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Samuel Stuchlý
May 12, 2021

Acknowledgements

I would like to thank Ing. Kamil Jeřábek for guidance, willingness and valuable advice during work on this assignment.

Contents

Chapter 1

Introduction

The main focus of this thesis is on the kubernetes container orchestration platform, which is a very popular open-source project. Orchestration technology has been around for long time, but it is only in the recent years that it has become really popular, and now it is basically a standard for any application using microservice architecture. Microservice architecture has many advantages over monolithic architecture and has become very popular in recent years and also seems to be the way of the future. Kubernetes is still a relatively new technology and not everybody is familiar with it. With kubernetes being an open source project, it has opportunity to grow more and more with increase of its popularity. These are some of the reasons I was interested in this theme for my bachelor thesis.

1.1 Thesis goal

This bachelor thesis is about setting up a small on-premise kubernetes cluster to be used on VUT FIT and tasks related to this goal. In this thesis we will explore Docker as a software for running containers, container technology and its runtimes. Then we will further explore Kubernetes as an industry leading container orchestration platform, and explain its inner workings. We will familiarize ourselves with other aspects of kubernetes such as persistent storage options, and cluster and application monitoring. Based on our researched information we will then design an optimal cluster architecture for our on-premise kubernetes cluster based on machines available to us on the university grounds. We will also look into automation tools which can be used effectively with kubernetes, particularly Ansible automation tool, and will construct a series of configuration files called playbooks for smooth and simple integration of our machines into our kubernetes cluster. When our kubernetes cluster on the university grounds is up and running, we can then start experimenting and deploying applications such as kubernetes dashboard, or a storage solution. Since this on-premise cloud is small and quite isolated compared to hosting a kubernetes cluster on some cloud provider outside VUT like Amazon Web Services or Google Cloud Platform, it might have different requirements and therefore different optimal solutions than those that dictate common practices. Due to this fact we will be also looking for fitting persistent storage solution. Another part of thesis will be setting up a cluster-wide monitoring system deployed on the cluster. Final part of this thesis will focus on testing implemented cluster‘s functionality. Additionally a short website documentation on how to set up and deploy example application, will be created and then deployed onto

the cluster as a form of example application deployment.This can serve as a guide for the people running their applications on the kuberentes on-premise cluster.

Another reason of my interest and a benefit of this thesis is that the outcome of the thesis should produce a production ready kubernetes cluster that academics on the VUT FIT can use and benefit from.

1.2 Thesis structure

The theoretical part of the thesis starts with section 2 entitled 'Container technology and Docker'. Container technology and containerization is described first to provide basis of technology which docker stands on. When this basis is explained, we move on the description of docker and example of its architecture, where we discover that the most interesting component of docker in relation with container technology is container runtime. We then further explore what container runtime is and different types of available container runtimes. Then it is briefly mentioned that Kubernetes orchestration platform decided to drop support for docker as its container runtime. After that in chapter 3 the topic container orchestration is explained, along with its importance and three main examples of container orchestration tools are listed :

- Docker Swarm;
- Apache Mesos;
- Kubernetes.

Kubernetes being the main subject of interest here, is then described into further detail, starting with its main components and then overall architecture. After that, we look into the topic of Persistent storage in kubernetes cluster. The most important storage related resources are mentioned and described. Then the focus is shifted in the direction of monitoring kubernetes cluster with some built-in tools mentioned, as well as the most popular monitoring software called Prometheus. Following section 3.5 is about automation in general and specifically about the Ansible automation tool. The last chapter 4 of the Theoretical part of this bachelor thesis focuses on the design and architecture of our small on-premise kubernetes cluster. The architecture is described as well as reasoning behind the choice of such design. The Implementation**??** part the thesis begins with describing the choice of kubernetes installation tool, being Kubespray. Then it moves into the description of required preparation of servers that are to become kubernetes nodes. Section 'Setup with Kubespray' 5.3 notes changes done to forked kubespray repository to configure the cluster based on architecture requirements. Theory also get a bit into the topic of loadbalancers and difference between baremetal and cloud-provider kubernetes architecture. n the next section 'Running Kubespray' 5.4.3 ,an actual installation of cluster with the help of Ansible and Kubespray is outlined with highlighted commands and regarding details. Process of accessing cluster5.5 is described for kubectl tool. Moving to the GUI based cluster access, two popular choices (5.6, 5.7) are mention with again requirements and steps to gain access. Next two sections are dedicated to monitoring 5.8 and storage 5.9 solutions applied to the cluster. They each provide implementation details, deployment process and description of their available dashboards. The last chapter of the thesis is devoted to testing 6 the functionality and features of implemented running on-premise kubernetes cluster.

Chapter 2

Container technology and Docker

To introduce kubernetes, some terms need to be explained first. Although this thesis is focusing mainly on Kubernetes and its architecture and usage with a small cluster, it is important to know what kubernetes as a technology stands on. As an intermediary to understanding kubernetes in this thesis will be used Docker. Docker is chosen for this role for number of reasons. Starting with Docker being probably the most recognizable software in association with container technology. Docker engine has also been the preferred technology for running contaners with kubernetes, for couple of years since its inception. As of time of writing of this thesis, Kubernetes has made a decision to drop Docker in favour of other more suitable container runtimes. This fact will be explored further in the thesis. Even though this decision was made, Docker is still a good example and a segue to underlying technologies of kubernetes. To understand docker we should first define what docker stands on and what is container technology.

2.1 Containter technology

Container technology, has been around for a long time, even though, it has only gained most of its popularity in recent years.

In 1979 the Container technology was born alongside Unix version 7 and the introduction of chroot system. The chroot system isolates a process by restricting an application's access to a single specific directory which consists of its own root and child directories. This was the first sight of an isolated process, and not so long after, in 1982 it was also added into BSD OS.

No signifact progress happened for more than a decade. At the beginning of 21st century the container technology started to slowly come back with the progress and improvements to Linux kernel. First in 1998, an extended version of chroot, by the name of 'jail' became a part of FreeBSD. This was later adopted by Solaris 10 and then Solaris 11 full feature called first 'zones' and then finally 'containers'. In coming years as Linux became more and more popular, this technology was adapted into its standard LXC. The most significant implementation of container technology came with Docker in 2013, and became widely popular among developers around the world. [28] [27]

2.1.1 What is a container?

"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another [10]."

Containers just like virtual machines are form of virtualization technology. Containers and virtual machines are based on similar resource isolation and allocation benefits. The difference between them is that instead of the hardware, containers virtualize the operating system, which makes them more efficient and portable .

Containers are an abstraction at the application layer that packages code and dependencies together. Multiple containers are able to share the OS kernel with other containers, and run simultaneously on the same machine, while each running as isolated processes in user space. Containers, being typically only size of tens of MBs, are usually smaller than Virtual Machines, but are able to handle more applications and require less resources.

Official docker website defines docker container image : "A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings [10]."

Docker uses Docker Engine to run container images, that are configured as a Dockerfile and then turned into containers at runtime. An advantage of container technology is that containerized software will always run the same, regardless of the infrastructure. Thanks to the software isolation, containers are able to make sure, that it works uniformly, not depending on environment where it is run. [10]

2.2 Docker

What is Docker? Official docker documentation describes docker as " an open platform for developing, shipping, and running applications. [3]"

Docker is a platform that enables application's separation from any infrastructure, and eliminates compatibility and environment dependant issues. Docker is used to run applications in an isolated environment called container. Docker creates and manages containers, which are basically a lightweight versions of virtual machines, but unlike virtual machines, containers do not require a hypervisor, but are able to run directly on the host machine's kernel. "Docker extends LXC with a kernel and application-level API that together run processes in isolation [28]."

Base of docker container creation is a docker image. It can either include the very basic specification, such as OS, or it can consist of a mulitple pre-defined applications and commands for their configuration and launch. Docker images are specified within a configuration file called Dockerfile. This Dockerfile is then used to create a Docker Image that can then be run with Docker Engine that creates a container.

Docker is a great tool for running containers. And as mentioned previously, containers and container technology is also the core of Kubernetes. Docker is actually a fairly complex piece of software that consists of many components. For the purpose of this thesis only couple of them are important. [28] [3]

2.2.1 Docker architecture

Docker uses a client-server architecture. The Docker client communicates with the Docker daemon, which is responsible for building, running, and distribution of Docker containers.[3]

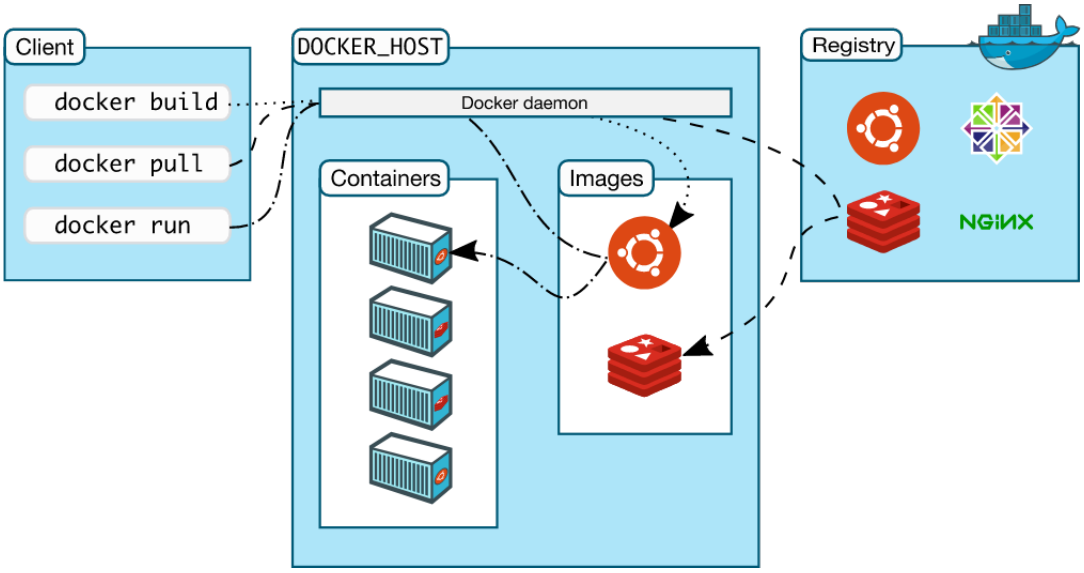

Figure 2.1: Docker architecture [3]

The Docker daemon

The Docker daemon (dockerd) is a persistent background process that listens to the Docker API and receives requests based on which it manages Docker objects such as images, containers, networks, and storage volumes. A daemon is also able to communicates with other daemons. [2] [3]

The Docker client

The primary way of interaction with the Docker for a user is Docker client (docker). When commands such as 'docker run' are called, the client forwards these commands to dockerd, which carries them out. The docker uses Docker API for communication. The Docker client is able to communicate with multiple deamons which can be running on the same host as client or on a remote one. [3] [2]

Docker registries

A Docker registry is where Docker images are stored. A publicly available registry for anyone to use is called a Docker Hub and it is the default place where Docker looks when looking for an image. Users are also able to run their own private registry. [3]

The interesting part of the Docker Engine for this thesis and Kubernetes as well, is what actually happens inside the docker deamon. That is how these mentioned Docker Images are being run as containers. The component responsible for running containers is called a Container Runtime. Although Docker itself has been called a container runtime, it actually consists of a high-level container runtime called containerd and a low level container runtime called runc.

2.2.2 Container Runtimes

When talking about container runtimes, it is important to understand that there is no current standard of what container runtime is. There are two main forms of container

runtimes, and those are – high-level container runtimes and low-level container runtimes. Low level container runtimes are programs that actually run the cointainer. High-level container runtimes are on the other hand more complex programs that incorporate more functionality, and contain a low level container runtime for its purpose.

It is also important to get familiar with certain terms concerning container runtimes that have been set in place for the purpose of standardisation.

The Open Container Initiative

"The Open Container Initiative is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes [1]." the Open Container Initiative's (OCI) was established in 2015 by Docker, CoreOs and other container industry leaders. Its mission is to create open industry standards around container formats and runtimes. To be considered OCI-compliant, container runtime must implement two OCI defined specifications: the Runtime Specification (runtime-spec) and the Image Specification (image-spec). Runtime Specification defines an interoperable format to build, transport and prepare a container image to run, and the Image Specification describes the lifecycle of a running container and how a tool executing such a container must behave and interact with it. [1] [38]

The Container Runtime Interface

The Container Runtime Interface (CRI) is "a plugin interface which enables kubelet to use a wide variety of container runtimes, without the need to recompile [35]." It was first introduced with Kubernetes 1.5 release. Before CRI, kubelet and container runtime were working together closely, which meant more difficult and complicated integration of new container runtimes. CRI was created to eliminate this issue by setting standard interface that container runtimes could easily implement. Meaning, a container runtime that implement the CRI is guaranteed to be compatible with Kubernetes.

Examples of container runtimes

Docker As mention before, primary container runtime used in Kubernetes has been Docker. Docker is by far the most known container runtime in the the world, and during the years have gone through some changes in its internal architecture. Couple of most used container runtimes as containerd and runc were developed from docker, and are still used within docker. Docker uses a stack consisting of docker deamon dockerd, which is responsible for managing and calling a high-level container runtime Containerd, which then again is responsible for managing and calling a low-level container runtime Runc. This amount of overhead is a reason for many developers leaning away from docker in relation with kubernetes in the recent years.

Containerd is a standalone high-level container runtime. It enables image pulling and pushing, as well as storage managmenet and network capabilities definition. It can manage containers lifecycle by the use of a low-level container runtime like runc, by passing commands to it. This has already been seen in Kubernetes, so containerd is able to perfectly replace a Docker with its cri-containerd implementation. Containerd conforms to OCI, meaning it implements CRI, therefore it is fully compatible with kubernetes. [38]

RunC is a universal lightweight low-level container runtime. Runc provides a command-line interface for creating and running containers. It conforms to the Open Container Initiative (OCI) specification. To create a container with all its required configuration, runC uses the original lower-layer library interface 'libcontainer'. [34]

CRI-O is a lightweight alternative for a kubernetes container runtime. It is an implementation of the Kubernetes CRI (Container Runtime Interface) and it serves to allow usage og OCI (Open Container Initiative) compatible container runtimes. It enables Kubernetes to run pods with the help of any container runtime that is also OCI-compliant. As a default it supports runC but it should run without issues with any OCI-compliant container runtime. CRI-O can pull from any container registry. [12]

2.2.3 Choice of the right container runtime

The choice of which container runtime to use with kubernetes on the implementaion of practical part of this bachelor thesis was an interesting process. Just like for many for developers working with kubernetes, the first and best looking choice is the most popular choice – Docker. But the reality is that in recent years more and more developers are leaning away from docker and towards other container runtimes, such as CRI-O. One of the biggest kubernetes based projects – Openshift[1] (developed by RedHat) has also replaced docker with CRI-O for their platform. RedHat has also developed docker-like software called Podman.

"Podman is an open-source daemonless container engine for developing, managing, and running Open Container Initiative (OCI) containers and container images on your Linux System [8]." It provides a command line interface which is not only Docker-compatible, but it is similar in such extend that docker and podman commands can be just exchanged with an alias `docker=podman`.

Podman uses the libpod library to manage the entire container ecosystem, including pods, containers, container images, and container volumes. Podman is a tool that allows use of all of the commands and functions, such as pulling and tagging, used to maintain and modify OCI container images. Podman is able to create, run, and maintain containers created from container images.[13]

This type of shift away from docker, and on such a scale, signalizes even bigger shift in the community to other containter runtimes. This shift shows that community slowly begins to recognize dockers unnecessarily complicated architecture as unsuitable for the kubernetes project. This shift took a concrete effect during the time of writing of this thesis in late 2020, when official kubernetes upstream made a decision to deprecate Docker as a container runtime after kubernetes v1.20, which should be released some time in late 2021.

2.2.4 Kubernetes drops docker

Docker is aside containerd and CRI-O a popular choice for container runtime, but Docker was initially not designed as a contianer runtime used inside Kubernetes, and that causes a problem. As mentioned before, `Docker` is not actually one thing – it is an entire tech stack, with part of it called `containerd`, a high-level container runtime by itself. Docker is cool and useful because it has a lot of UX enhancements that make it really easy for humans to

[1] https://www.openshift.com/

interact with while doing development work, but those UX enhancements are not necessary for Kubernetes. As a result of this human-friendly abstraction layer,to get to containerd a Kubernetes cluster has to use another tool called Dockershim. This created another issue, since there is one more thing that has to be maintained and can possibly break. Why does Kubernetes need the Dockershim? Docker is not compliant with the Container Runtime Interface (CRI), which means kubernetes must prefer another supported container runtime. The upcoming change is that Dockershim is being removed from Kubelet in v1.23 release, which therefore removes support for Docker as a container runtime for Kubernetes. [29]

Chapter 3

Container orchestration and Kubernetes

Previous part of the thesis was dedicated to container technology and container runtimes. This part will move a step forward and describe how container technology can be used for enhancements of development and production environments. Containerization is a very useful technology, but its effectivness and perks depend on the use-case. If it is utilized for running some sort of monolithic application, its benefits are limited to basics like unified environments, and easy portability. However, where containerization really shines, is in microservice architecture, where parts of application are broken each seperated into one small contained microservice that is running in a separate container. This design helps massively with repairability, CI\CD practices, but most importantly scalability. In huge distributed applications it is very important to be able to scale the applications easily. Dealing with small amount of running cointainers is humanly manageable. But after certain threshold it is impossible to maintain without help of some sort of automation. That is where Container Orchestration comes in.

What does container orchestration mean? Container orchestration automates the deployment, management, scaling, and networking of containers. Container orchestration is the most beneficial to the systems that deploy and manage hundreds or thousands of Linux containers.

Container orchestration can be useful in any system that is running containers. It helps with deployment of the same applications across multiple different environments without the need of redesign. Microservices in containers make it easier to orchestrate services and their life cycles. Container orchestration tools allow users to integrate container deployment and so on into CI/CD workflows which is greatly appreciated by the DevOps teams. Containerized microservices are the foundation for cloud-native applications.

Container orchestration takes care of:

- Provisioning and deployment of containers.
- Redundancy and availability of containers.
- Scaling up or removing containers to spread application load evenly across host infrastructure.
- Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies.

- Allocation of resources between containers.
- External exposure of services running in a container with the outside world.
- Load balancing of service discovery between containers.
- Health monitoring of containers and hosts.
- Configuration of an application in relation to the containers running it.

When using a container orchestration tool, like Kubernetes or Docker Swarm , user typically describes the configuration of the application to be deployed in a YAML or JSON file, depending on the orchestration tool. These configurations files (for example, docker-compose.yml) specify to orchestration tool a location to gather container images from, (for example; Docker Hub),a networking configuration between containers, storage volumes specifications and logging locations for the described application. These configuration files are representation of concrete application, that is to be deployed. The same files can be later used or adjusted to deploy the same application on different environments. DevOps teams often use these configuration files to deploy application to development and testing environments before deploying to production cluster. [11]

Containers are deployed onto hosts, usually in replicated groups. When a new container is to be deployed into a cluster, the container orchestration tool schedules the deployment and searches the cluster for the most appropriate host to run the container on, based on predefined conditions such as CPU or memory requirements and availability. It is possible to set other constrains on the deployment as well, for example set constraints can be depending on labels or even metadata. There are many options to manipulate and influence orchestration tools choice of host on which to deploy application. Once the container is up and running, its lifecycle is managed by the orchestration tool, that makes sure everything is running as specified. [33]

3.1 Orchestration Tools

As described in previous section, container orchestration tools, platforms or software are important and useful for managing deployment and life-cycles of containers. Although there are many orchestration tools available, This section will explore three most significant and popular orchestration tools in the recent years.

3.1.1 Docker Swarm

Docker does have its own native Container Orchestration Engine called Docker Swarm. It was released in November of 2015 and just like Docker, it is written in Golang. Swarm is closely integrated with the Docker API, therefore it is strongly compatible with Docker. Swarm uses the same primitives as a single host docker cluster. That means no separate configuration of orchestration engine is required, and also from user perspective, Swarm does not require to re-learn any docker concepts. Swarm's deployment model uses YAML-based configuration file called Docker Compose. Swarm's features include auto-healing of clusters, overlay networks with DNS, high-availability through the use of multiple masters, and network security using TLS with a Certificate Authority [14]. As of time of writing this thesis, Swarm does not yet support native auto-scaling or external load balancing. Scaling must be done manually or through third-party solutions. Although ingress load balancing

is included with Swarm, external load balancing would have to be implemented with the help of third-party loadbalancer such as AWS ELB. Another disadvantage is that Swarm does not include any web interface. Even though Swarm is greatly outperformed by tools like Kubernetes, and the amount of users is very low compared to other orchestration tools. Swarm seemed to be headed toward discontinuation, but as of late 2020, its support is still being renewed. [14]

The main architecture components of Swarm include:

Swarm A Swarm is a set of nodes with at least one master node and several worker nodes that can be either virtual or physical machines. [33]

Service A service is the tasks a manager or agent nodes must perform on the swarm, as defined by a swarm administrator. A service defines which container images the swarm should use and which commands the swarm will run in each container. A service in this context is analogous to a microservice. It is where for example configuration parameters would be defined for an nginx web server running in the swarm. Parameters for replicas are also defined in the service definition. [33]

Manager node Upon deploying an application into a swarm, the manager node provides several functions. It manages the state of the swarm and delivers tasks to worker nodes. The manager node is able to act as worker node as well and run the same services as the worker nodes, but it is also possible to configure a manager node to only run manager node-related services. [33]

Worker nodes These nodes run tasks distributed by the manager node in the swarm. Every one of these worker nodes, is running an agent, responsible for reporting the state of tasks assigned to it back to master node, so the manager node is able keep track of tasks and services running in the swarm. [33]

Task Tasks are Docker containers that execute the commands defined in the service. Manager nodes assign tasks to worker nodes. This assignment is final ans cannot be changed during task's life-cycle. If the task fails in a replica set, the manager node will look for a new available worker node inside the swarm and assign a new replica of previously failed task to it. [33]

3.1.2 Apache Mesos

Apache Mesos was first released in 2016, it has been in development since 2009 by PhD students at UC Berkeley. Unlike Kubernetes and Swarm which are written in Golang, Mesos is written in C++. Mesos uses a bit different, more distributed approach to datacenter and cloud resources management, compared to Kubernetes and Swarm. Mesos ensures a high-availability cluster with the help of Zookeeper, that tracks cluster state between multiple Mesos masters. Other container management frameworks such as Kubernetes, can also be run on top of Mesos. It also has a Distributed Cloud Operating System 'Mesosphere DC/OS', which is based on Mesos itself. Mesos takes a more modular approach container management, which allows more flexibility in the scale and types of applications. Mesos is able scale to a very large number of nodes, and is used by the huge Socal media and other tech companies. "Apple even has its own proprietary framework based on Mesos called Jarvis, which is used to power Siri [14]." Mesos supports multiple types of container engines, including Docker, and is able to run on multiple operating systems, including

Linux, OS X, or even Windows. Due to high complexity and flexibility, Mesos might be challenging to learn, but its complexity and flexibility are what makes a good fit for many large-scale applications.

The main architecture components of Mesos include:

Master daemon Part of the master node responsible for agent daemons management. To achieve high availability Mesos Master Quorum, cluster needs at least three master nodes. [7] [33]

Agent daemon Part of each cluster node that executes tasks sent by the ramework. [7]

Framework A framework running on top of Mesos consists of a scheduler, whose responsibility is to select which available resources to use, and an executor process on agent nodes to run the framework's tasks. Framework can accept selected resources, and pass tasks to Mesos. Mesos then launches the tasks on the corresponding agents. [7] [33]

3.1.3 Kubernetes

Kubernetes (also known as "k8s") is a container orchestration platform released in 2014, and just like Docker, it is written in Go. The parent company which started an open-sourced Kubernetes project is Google, which is very experienced in big scale container management.

Kubernetes is seen today as a sort of standard for container orchestration. It is the flagship project of the Cloud Native Computing Foundation, which means it is also supported by a large number of industry leading companies such as Google, Amazon Web Services (AWS), Microsoft, IBM and RedHat. Although developed at Google, Kubernetes has a large open-source community behind it. Google uses Kubernetes for its own Container as a Service (CaaS) offering, called Google Container Engine (GKE). Kubernetes is aslo supported by many other patforms such as Microsoft Azure and Red Hat OpenShift. Kubernetes uses configuration files written in YAML to specify deployments and other kubernetes resources. Kubernetes provides many features along with container orchestration , such as built-in auto-scaler, load balancing, volume and secret management. Kubernetes also provides a web UI. Due to Kubernetes rich catalog of features, it does not require inclusion of many third-party software solutions. It is the amount and quality of features that make Kubernetes more attractive to users then Swarm or Mesos might be. Other factor to this, may be that kubernetes is very portable and is able to multiple cloud providers such as Amazon Web Services (AWS), Microsoft Azure, the Google Cloud Platform (GCP), as well as on-premise clusters. [14] [33] [23]

Kubernetes has a bit more challenging learning process and can take more effort to configure, compared to for example Swarm.

Cluster A cluster is a set of nodes with at least one master node and several worker nodes, that can be virtual or physical machines. [33] [40]

Kubernetes master The master is responsible for management, scheduling and deployment of application instances across cluster nodes. The full set of services which master nodes run is defined as control plane. The master communicates with nodes through the Kubernetes API server. The scheduler assigns to each new pod a node to run on. [33] [40]

Kubelet Each Kubernetes node runs an agent process called a kubelet. Kubelet manages the state of the node: starting, stopping, and maintaining application containers

based on instructions from the control plane. A kubelet communicated and recieves information from the control plane via the Kubernetes API server. [33] [40]

Pods The basic scheduling unit, which is made out of either one or more containers that share resources and are guaranteed to be running on the same host machine. A unique IP address is assigned to every pod within the cluster, allowing the running applications to use ports without conflict. Configuration of desired state of pod's containers is defined in the PodSpec resource, which is a YAML or JSON configuration file. These objects are passed to the kubelet through the API server. [33] [40]

Deployments, replicas, and ReplicaSets A deployment is a YAML specification object that defines the pods and the number of container instances, called replicas, for each pod. ReplicaSet defines the desired number of replicas to be running in the cluster at any point in time. ReplicaSet is a part of the deployment object. In case a running a pod dies, the ReplicaSet ensures that another instance of a pod is to be scheduled on an available node. [33] [40]

Kubernetes architecture and components will be explored in greater detail in the next section.

3.2 Kubernetes architecture

Deploying Kubernetes to a set of machines, creates a Kuberentes cluster. A Kubernetes cluster consists of a control plane and a set of worker machines, called worker nodes, that run containerized applications. Every cluster has at least one worker and master node. A node can be configured to serve as a master and a worker node at the same time. The worker nodes host the application workload which is assigned in smallest workload resources called Pods. The control plane manages the worker nodes and most of the clusters resources. In production environment,to achieve high availability and fault-tolerance, cluster usually consists of multiple nodes and the control plane usually runs across at least three master nodes. We will now further describe various components needed to have a complete and working Kubernetes cluster.

This is the diagram of a Kubernetes cluster with all the components tied together.

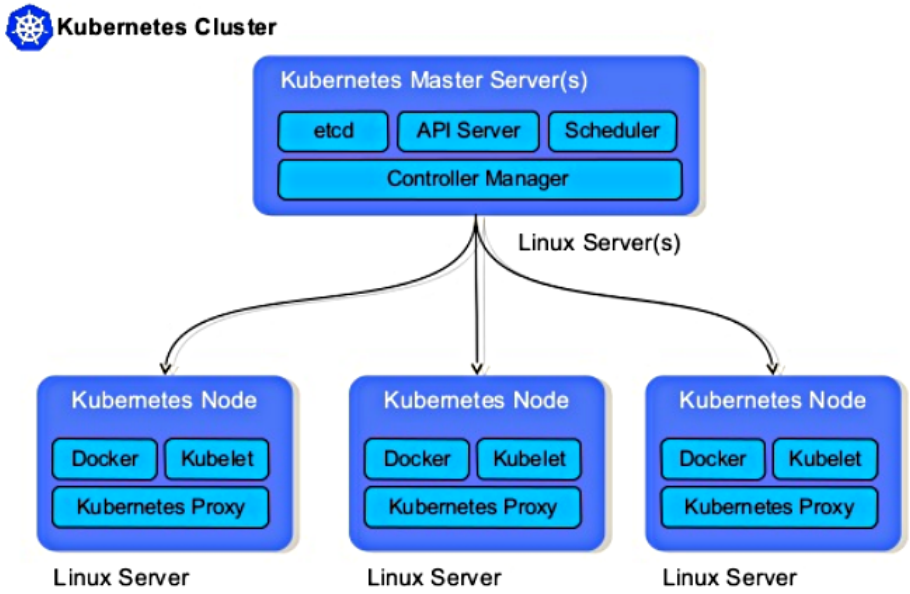

Figure 3.1: Kubernetes architecture [42]

3.2.1 Control plane

The control plane's components are responsible for global decisions about the cluster, such as scheduling and resource managment, as well as constatnly keeping alert for changes to the cluster state and be ready to respond. Simple example of this is when one replica of some pod fails, a new replica needs to be assigned resources and spun up again. This is managed by control plane. The control plane is in constant contact with clusters nodes, making sure the cluster runs exactly as it was configured.

kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane, handling internal and external requests. The main implementation of a Kubernetes API server is kube-apiserver. Kube-apiserver is designed to scale horizontally, meaning it scales by deploying

more instances. It is possible to run several instances of kube-apiserver concurrently and balance traffic between those instances. The API server determines validity of a request and then proceeds to process it. The API is accessible through REST calls, through the kubectl command-line interface, or through other command-line tools such as kubeadm.

kube-scheduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on. Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines [17].

kube-controller-manager

Control Plane component that runs controller processes. Controllers are responsible for checking and managing resource status. Each controller is its own separate process, but they are all compiled into a single binary and run in a single process. These controllers include:

Node controller Responsible for noticing and responding when nodes go down.

Replication controller Responsible for maintaining the correct number of pods for every replication controller object in the system.

Endpoints controller Populates the Endpoints object (that is, joins Services Pods).

Service Account Token controllers Create default accounts and API access tokens for new namespaces.

[17] [5]

etcd

Kubernetes documentation define etcd as "Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data [18]."

3.2.2 Nodes

A Kubernetes cluster needs at least one worker node, but will generally consist or have multiple worker nodes. Every kubernetes node runs several Kubernetes components such as kubelet and kube-proxy. These components are responsible for maintaining running pods and providing the Kubernetes runtime environment. Pods are scheduled and orchestrated to run on nodes. Nodes can be added or removed from the cluster based on need.

kubelet

Each compute node contains a kubelet, an agent that communicates with the control plane. The kublet makes sure containers are running in a pod. When the control plane needs something to happen in a node, the kubelet executes the action. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet does not manage containers which were not created by Kubernetes.

kube-proxy

Kube-proxy is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. Kube-proxy maintains network rules on nodes. These network rules allow network communication to Pods from network sessions inside or outside of the cluster. Kube-proxy uses the operating system packet filtering layer if there is one and if it is available. Otherwise, kube-proxy forwards the traffic itself.

Pods

A pod is the smallest and simplest unit in the Kubernetes object model. It represents a single instance of an application. Each pod is made up of a container or a series of tightly coupled containers. Pods are considered an ephemeral resource, meaning it is expected from pods to exist for a short time, and can be deleted and spin up anytime in a short time by user as well. [40]

Container runtime

The container runtime is the software that is responsible for running containers. Kubernetes supports multiple container runtimes such as Docker (support ends with k8s v1.20 [29]), containerd, CRI-O, and any other runtime that implements the Kubernetes CRI (Container Runtime Interface).

Information for the previous section were taken from the book 'Mastering kubernetes'[40] and Online kubernetes documentation [17].

3.3 Storage

An important part to mention in association with kubernetes is storage, meaning storing of persistent data. The basic abstraction for storage in kubernetes are Volumes. Volumes are mounted inside containers and then can be used as if they were containers own file systems. Kubernetes does not provide persistent data storage out off the box. Since data stored on pods naturally does not stay there after the end of pods life cycle. Also when the new replica of pod is being spin up, it might be running on a different node, therefore unable to access the data written by its predecessor somewhere else. If the goal is to store persistent data within the cluster, it needs to be separately configured. Requirements for the persistent data storage are:

- storage cannot be dependent on pods life-cycle, storage;
- storage must be available on every node of the cluster;
- stored data have to be able to survive even if the cluster crushes.

The basic Kubernetes storage abstraction is the volume. Containers mount volumes that bind to their pod and they access the storage, wherever it may be, as if it is in their local file system. [40]

3.3.1 Persistent Volume

Kubernetes has configurable kubernetes cluster resource for this purpose, called Persistent Volume. As well as other kubernetes resources, Persistent Volumes are specified with a yaml file of kind PersistentVolume and can configure its parameters, such as type and capacity. Persistent Volume is kubernetes cluster resource, but as such it is not necessarily a part of the kubernetes cluster. It is not a namespaced resource, meaning it is not bound to a specific namespace, but accessible across the entire kubernetes cluster. Although the Persistent Volume might be storage space located on the nodes belonging to our cluster, it can also be located on an external server, or it can be a cloud storage hosted by some cloud provider like AWS or GCP. Of course it is possible to combine types of storage in our kubernetes cluster and use local storage for smaller applications, and cloud storage for an application dealing with larger amounts of data. Kubernetes currently support more than 20 different storage backends. When comparing local persistent storage to remote options, it is important to mention that local storage does not comply with all previously mentioned desired requirements, in that:

- It is stored on a specific node, and therefore might not have the same accessibility to all nodes.
- It is not able to survive cluster crash.

Due to these two failed requirements, it is almost always preferable to use remote storage. However there can be exceptions to this preference, as will be explored in further sections of this thesis. [26] [40]

3.3.2 Persistent Volume Claim

Once the Persistent Volume resource is defined, pods need to claim this volume to use as storage for the application they are running. For a pod to get access to a volume and be able to mount it onto its containers, another kubernetes resource needs to be created, and that is Persistent Volume Claim. It is also a yaml file that defines parameters of the claim , such as its name, volume mode, access type and storage size. Persistent Volume Claim is not bound to a single Persistent Volume resource, but functions as a claim on storage of certain requirements and size and when satisfying this claim, kubernetes will look at all the available Persistent Volume resources and find the best fit. Found volume is then mounted into the pod and then into the pods containers as well. Persistent Volume Claim is unlike the Persistent Volume resource a namespace resource, and so it has to belong to the same namespace as its application. [26] [40]

3.3.3 Storage class

Storage class is a kubernetes resource that is responsible for dynamic provisioning of Persistent Volumes. With storage class, cluster administrators do not need to manually provision Persistent Volumes to be used by applications, but these volumes are provision dynamically by the storage class to satisfy applications PersistentVolumeClaims. Storage class specifies a 'provisioner' attribute which defines which provisioner and therefore storage backend to use. Desired storage class is specified in the applications PersistentVolumeClaim resource. By using storage classes in the Persistent Volume Claims, kubernetes does not look for fitting volume within its already defined Persistent Volumes, but requests the Persistent

Volume from the specified storage class, which then create Persistent Volume that satisfies the needs of the Claim. [21] [40]

3.3.4 ConfigMaps and Secrets

Last type of volumes to mention are configMaps and Secrets. These two types are different than the previously mentioned ones in that these are their own components managed by kubernetes itself. These are both local volumes not created by Persistent Volumes nor Persistent Volume Claims. These components define configuration files and certificates that can be mounted as volumes into the pod that uses them. [16]

Storage solutions with kubernetes on our on-premise kubernetes cluster will be further explored and compared in the Implementation chapter of this thesis.

3.4 Monitoring

An important part of maintaining a running kubernetes cluster is monitoring. Monitoring information about status of the cluster and setting up alerts might not seem very important at first, but it is an essential part of running applications on the cluster. Monitoring mostly helps with detecting issues, debugging found issues, but also with performance evaluation and all kinds of custom metrics we might want to track about our deployed application. Kubernetes has couple of built-in monitoring tools for metrics collection.

3.4.1 Metrics server

For tracking resource metrics, meaning metrics about resources such as CPU storage and memory, kubernetes provides a Metrics Server and Metrics API. Metrics server is a cluster add-on for tracking resource usage data from a kubelet for each node of the cluster and provides aggregated metrics with the help of Metrics API. In some installations of kubernetes, Metrics Server might not be deployed automatically. In that case we need to deploy it ourselves. When the server is running, then with the use of Metrics API we are able to see resource metrics of our entire cluster. They are accessible through different kubectl commands, such as top, get and describe. [30] [36]

3.4.2 Kube-state-metrics

Kube-state-metrics docs describe kube-state-metrics as “a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects. It is not focused on the health of the individual Kubernetes components, but rather on the health of the various objects inside, such as deployments, nodes and pods [6].” Kube-state-metrics is also a cluster add-on, although unlike Metrics Server, it does not usually come with the installation of kubernetes. Kube-state-metrics needs to be run as a deployment with one replica. There are some resources that needs to be configured to run kube-state-metrics as its service and rbac rules. Kube-state-metrics docs contains predefined yaml manifests that can be used for this purpose. When kube-state-metrics service is running , it exposes metrics through a HTTP endpoint `/metrics`. Exposed metrics are accessible on the mentioned endpoint in the plain text, so they are easily readable by human as well as any scraper, or Prometheus instance. [30] [36]

3.4.3 Prometheus

Unlike the previously mentioned metrics providers, Prometheus is not part of the kubernetes, but is its own open-source project. It is a widely popular systems monitoring and alerting toolkit. As of 2016 Prometheus is also a part of Cloud Native Computing Foundation , along with kubernetes. Prometheus is a very powerful tool and can be used for multiple things in association with kubernetes. It is usually running as single server collecting metrics from the cluster. Although also scheme is used where the single cluster scope Prometheus server is configured to scrape metrics from other Prometheus server instances running with cluster namespaces. Prometheus uses a pull model over HTTP to collect metrics from its targets. These targets can be configured either by static configuration, or by usage of its service discovery feature. This feature can be used for many systems, and it is very powerful with kubernetes as well. With kubernetes, it can be configured to scrape metrics from every node, pod, services, or endpoints. It can be configured to scrape resource metrics, and or any custom metrics exposed by a running application. It does this by interacting directly with kubernetes API. This configuration is a very powerful and easy way of collecting and distinguishing exposed metrics in our cluster. Prometheus has its own dashboard to which can be accessed via browser to quickly look up and visualize collected metrics. It uses powerful query language called PromQL. PromQL queries allow the use of visualization software such as Grafana, which is a very popular option to visualize data collected by Prometheus. Alert Manager is a also a built in part of Prometheus for configuring and managing alerts based on collected metrics. Prometheus deployment within the kubernetes cluster can be a bit complicated. It will be further described in the implementation section of this thesis. [9]

3.5 Automation

With increase of complexity of a system, rises often not only its efficiency and functionality but also maintenance requirements which cost a lot of effort , time and money. Kubernetes is certainly one of the complex systems out there and that is why its maintenance also suffers from its complexity. Fortunately there are automation tools to eliminate these issues. The most popular automation tool with relation to kubernetes is Ansible.

3.5.1 Ansible

Red Hat describes Ansible as a "radically simple IT automation engine that automates cloud provisioning,configuration management,application deployment, intra-service orchestration, and many other IT needs [4]." Ansible is a push configuration tool, which means it does not need any sort of client to be installed on the nodes for it to work. It only requires Ansible to be installed on the one main local machine that then pushes configuration to the nodes.

Ansible architecture consists of:

Local machine - machine which administrator uses for automation, Ansible needs to be installed on this machine, contains Module and Inventory.

Nodes – systems to be configured, controlled by the local machine.

Inventory – document that holds networking info about nodes and groups under labels.

Module – collection of configuration files also called Playbooks.

Playbook – set of instructions to configure a node, written in yaml, each of these sets is called a play.

How Ansible works, is that we can sort hostnames of our nodes in the Inventory into the groups based on what configuration we want to apply to them. Then Ansible connects to these nodes and pushes out Ansible modules consisting of Playbooks. Then these modules are executed by Ansible over SSH, a secure connection. This ensures equal and consistent environment and configuration among nodes. [4]

Ansible will also be used in this thesis for applying configurations to cluster nodes.

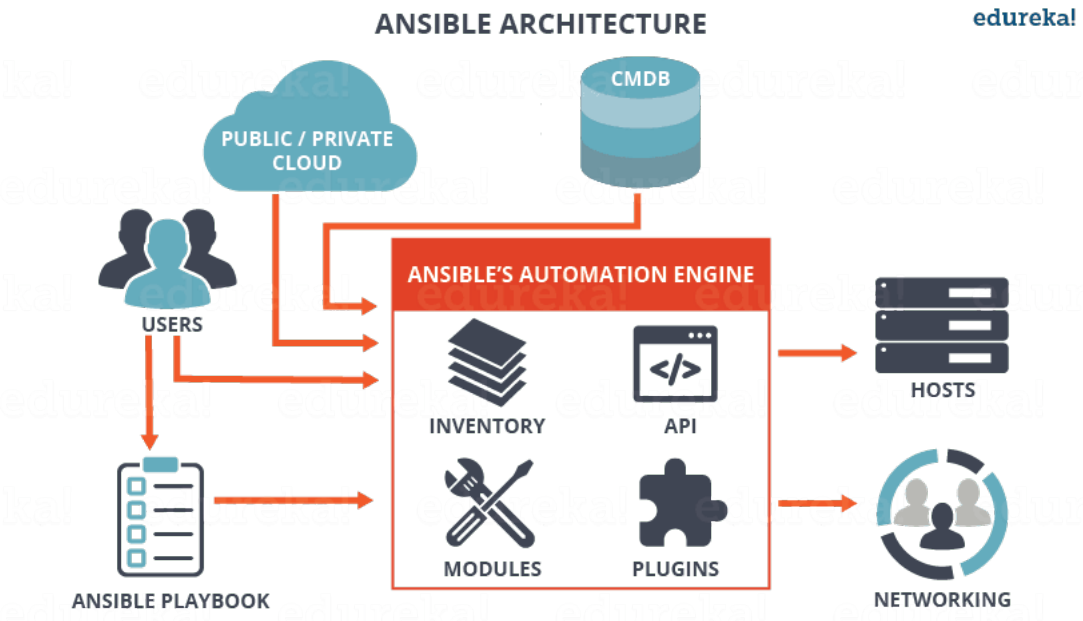

Figure 3.2: Ansible architecture [37]

Chapter 4

Architectural design of small on-premise kubernetes cluster

The most important variable in terms of architectural design of kubernetes cluster is of course number and type of available machines that will be part of that cluster. For practical part of this thesis we have available eight machines total, being:

- 7 machines with 16GB of RAM and 24 CPUs;
- 1 machines with 256GB of RAM and 64 CPUs.

4.1 High Availability

With these machines available, we can build a solid kubernetes cluster. First big decision about kubernetes architecture is control plane. Of course control plane is the most crucial part of entire kubernetes cluster. There are two main options for kubernetes architecture. If the cluster consists of very few nodes, then we do not consider high availability a crucial feature of our cluster, one master node setup good enough, since it does not waste resources on multiple master nodes, and leaves the most amount of resources possible for running deployed applications.

Generally this is not the advised control plane architecture since, in most cases, as well as in our cluster, administrators want to ensure high availability of their kubernetes cluster. High Availability means that even if the master node of the cluster happens to crash for whatever reason, the entire cluster is not compromised, but is still able to function as normal because it still has at least one more master node running with same shared data as the one that crashed. As mentioned before, cluster state is saved in etcd database, which is running on the master nodes. For database to function correctly in a distributed environment, there needs to be a strict majority to elect a leader. Etcd is a reason why it is recommended to use three master nodes for the kubernetes cluster control plane. Etcd‘s basis is a distributed consensus algorithm called `Raft`[32]. The equation that governs over the `Raft` is $N/2 + 1$. This means that for every action to take place, there needs to be a strict majority of at least 51% of the members. There for updates to be recorded, at least 51% members of the etcd cluster need to reach a quorum to elect a leader. Therefore if the cluster has two master nodes and one goes down, only 50% of etcd cluster can reach quorum. Therefore cluster is not operable anymore. That is why at least three master nodes are needed, so if one goes down, there are still 66% members that can reach quorum and function properly. Three

is also better choice than four master nodes, because it requires three running masters to reach a quorum, so it can still withstand just one master node failure. Meaning it is the same as with three master nodes, but consumes more resources. [39]

Even though our cluster does not consist of particularly high number of nodes, the high availability three-master node architecture is the best possible configuration for our cluster. Master nodes do not need as much resources as worker nodes, so the most suitable solution for them would be to use the 3 out of 4 machines with worse hardware to be master nodes, and all the other machines as worker nodes. Although it is possible for a node to be a master and a worker node at the same time, it is almost impossible to accurately predict the scope of workload that will be required for the master nodes to run under, so for an initial set up, these master nodes, will be configured as master nodes only. However if the future of actual cluster usage shows that the master nodes are not being used to their full potential, it is very easy to configure them to be worker nodes as well and run small less demanding applications on them, such as monitoring Prometheus server.

4.2 Node Affinity

Due to the nature of our differing types of machine, it is best to find a way to be able to differentiate certain high resource consuming applications and isolate them to run exclusively on the more powerful machines, instead of splitting the workload between more less powerful machines and there costing some computing time. In Kubernetes this is done with the help of Labels and Affinity or Taints. Taints allow a node to choose and decline the set of pods to run on itself. The opposite of that is Affinity. We can specify node affinity and anti-affinity with the use of labels for certain applications, which we want to run only on our more powerful machines. We can either configure the application pods to run only on certain nodes, or to preferably run on chosen nodes, if it is possible at the time, otherwise ignore the affinity label. [15] [22]

4.3 Storage and Monitoring

Concerning design of persistent storage of our cluster, even though it is the preferable choice in most cases, we will not use remote cloud based storage, due to universities choice to try to keep their data on their own servers, unless necessary. Exact details of storage type and its implementation will be described further in the implementation part of this thesis, since it is a matter of experimentation and will be chosen and adjusted based on gathered data. The same applies to Prometheus configuration.

4.4 Cluster architecture

This diagram represents architecture of the designed kubernetes cluster.

As depicted on the diagram 4.1, we will be using a multi-master High-availability cluster architecture. Multi-master architecture will be using Stacked etcd topology [20]. Meaning its control plane will consist of three master nodes, each storing etcd on its node. When setting up this multi-master topology with three master nodes, control plane suddenly has three replicas of every controller residing on master node. Only one instance of Scheduler and Controller manager is active in a cluster. However etcd and API servers are active on each master node. Etcd on each node communicates only with an API server on the

same master node. Existence of three API servers creates confusion in the communication between kubelets on worker nodes and control plane. Therefore there is a need of loadbalancer to be running on every worker node to route the traffic into the API server on correct master node. [19]

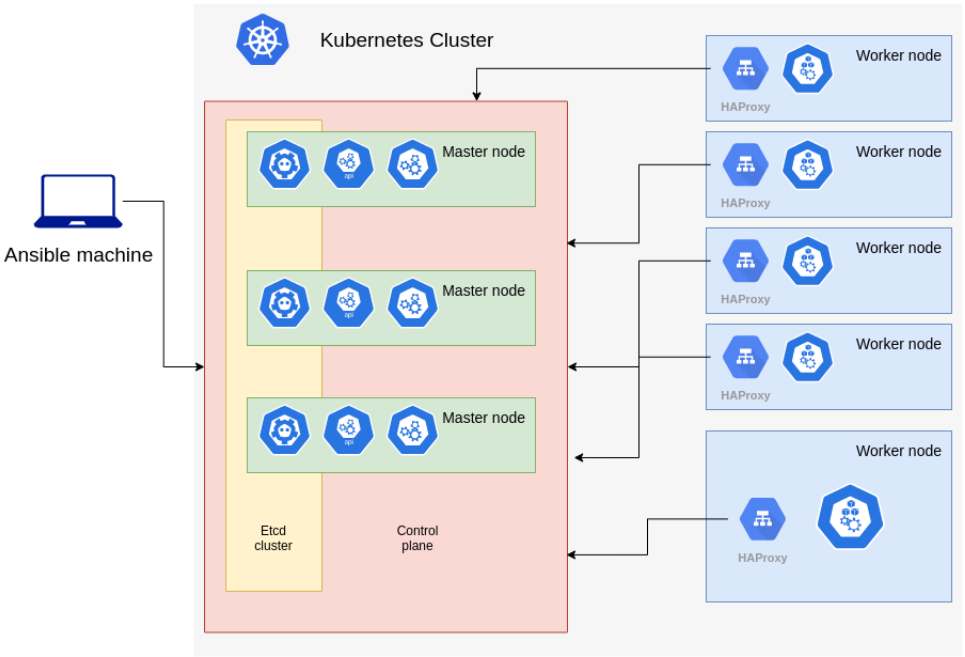

Figure 4.1: Cluster architecture

Chapter 5

Implementation

Implementation part of this bachelor thesis consists of setting up a kubernetes cluster on the university servers. This goal consisted of seven parts, such as:

- preparing the nodes;
- configuring Kubespray;
- setting up cluster;
- installing dashboard;
- installing monitoring;
- installing storage;
- testing.

There are multiple ways of setting up the kuberentes cluster. Kubernetes community provides multiple tools for setup on different platforms. One of the universal tools is kubasdm[1]. This tool was originally considered for the kuberentes setup in this bachelor thesis. However this tool requires to run multiple command on each machine. Implementation of this would have to consists of multiple scripts having to be run on multiple machines. This is not very time and energy efficient, therefore an automation tool like Ansible comes in play. Ansible allows to run entire setup on multiple machines at the same time with user providing just one command. Combination of kubeadm and ansible therefore seems very natural. Hence the birth of another kubernetes cluster setup tool named Kubespray.

5.1 Kubespray

Kubespray[41] is a project build by kuberentes community that uses ansible playbooks and kubeadm tool for bootstraping kubernetes clusters on variety of platforms, including baremetal servers with variety of customization. Although this tool is far from perfect, it can be a huge help for more complicated cluster setup, however if the user requires some custom non-traditional cluster setup, kubeadmn might be more rewarding. For the purpose of this thesis, kubespray was chosen as a better fitting tool.

[1] https://github.com/kubernetes/kubeadm

5.2 Preparing the nodes

Our servers are all running a minimal installation of CentOS 8. Kubespray needs to be able to have reliable communication with the use of Ansible from admin machine to all the servers, since it will execute multiple commands simultaneously on multiple machines. To allow for this communication to happen we need to generate a SSH key on the admin machine and then with use of `ssh-copy-id` command copy this SSH key to every server of the cluster. This will ensure we do not need to input password every time we attempt a connection from admin machine to the servers, which will be quite a lot. Another requirement to run Kubespray on our servers is that Kubespray uses `tar` program to decompress some files needed for kubernetes cluster setup. Therefore a simple Ansible playbook was created for the purpose of installing it. This playbook is run before the Kubespray to ensure setup does not fail.

CentOS specific requirements for ensuring communication inside kuberentes cluster and cluster setup is allowing communication on certain ports on all master and worker nodes. Communication on this ports must be allowed, otherwise Kubespray will run into issues when attempting to send API calls from the first master node to the other nodes, eventually fail to connect receive any response and fail the setup. To avoid this, a simple bash scripts were written for this purpose. One for master and one for worker nodes and an Ansible playbook was made to run these scripts on our servers before cluster setup. Additionally script for stopping `firewalld` on all nodes was added, since nodes could not communicate to the API with `firewalld` running.

5.3 Setup with Kubespray

A fork of the Kubespray repository on GitHub[2] was used for this setup. Kubespray repository provides example setup playbooks and a documentation about inventory setup and customization. For implementation of our cluster, we decided to go with high availability three master setup as previously explained 4.4. As a container runtimes we used `crio` and as pod network we kept `calico`, which is a default setting on Kubespray. Pod network could be changed by setting a `kube_network_plugin` variable to desired pod network plugin name.

Default container runtime for Kubespray is still docker. Choice for this cluster was `cri-o` container runtime, since docker will no longer be supported from kubernetes 1.22 version. Current version of kubernetes, which was installed on our cluster was v1.20.4. Therefore to avoid difficulties when updating versions in the future, `cri-o` was chosen. To use `cri-o` as a container runtime with Kubespray, the right configuration needed to be specified. First of all a `container_manager` variable was set as `crio` inside the `inventory/mycluster/group_wars/k8s-cluster/k8s-cluster.yml` file. Then in `inventory/mycluster/group_wars/all/all.yml` three variables were set:

```
download_container: false
skip_downloads: false
etcd_kubeadm_enabled: true
```

Then in `inventory/mycluster/group_wars/etcd.yml`, `etcd_deployment_type` was changed from `docker` to `host`.

[2] https://github.com/kubernetes-sigs/kubespray

At last a new configuration file called crio.yml was added into the `inventory/mycluster/group_wars/all` directory, to enable Docker Hub[3] registry mirrors. The content of the file was taken from Kubespray documentation[41].

Another detail mentioned in the Kubespray documentation[41] was concerning running kubernetes cluster on CentOS 8 servers, explaining the need to ensure that when using `Calico` container network interface, then `calico_iptables_backend` variable needs to be set to either `„NFT"` or `„Auto"`, since CentOS 8 ships only with iptables-nft.

5.4 Loadbalancer

When choosing loadbalancer for the cluster, it is important to understand that usually in kuberenetes we talk of two different loadbalancers being external and internal. Most of kuberentes clusters are run on server infrastructure already provided by some sort of cloud provider, such as Google Cloud platform, Amazon Web services, Microsoft Azure and others. Most of these cloud provider platforms already implement their way of external loadbalancing for clusters. Meaning these platforms already have dedicated loadbalancer machines in place to balance incoming traffic to the servers. Therefore only type of loadbalancer, users usually care about, is internal `localhost loadbalancer`, that is responsible for communication inside the cluster itself. It is good practice to have a separate loadbalancer machine as a part of multi-master cluster that handles request from the nodes and forwards them in the appropriate available API-server.

5.4.1 Internal vs External loadbalancer

However, in case of cluster hosted on baremetal servers there is no already provided loadbalancer, therefore loadbalancing incoming traffic is left to the user. As mentioned before, a way to balance and route traffic within you cluster is an ingress resource. Ingress resource allows user to define which pods should receive which incoming traffic through services. Services can be of more than one type. Internal as `ClusterIP`, external as `NodePort`, or a `Loadbalancer` type. `Loadbalancer` type service is able to balance incoming traffic between multiple pods. It is common to define a service as a loadbalancer type, mainly when it is exposing some sort of domain that will receive a lot of outside traffic. Service of type `Loadbalancer` uses external loadbalancer maintained by cloud provider platform as mentioned. Since baremetal clusters do not have this by default. Users have two options. Either configure an external loadbalancer themselves, such as MetalLB[4], or just avoid `loadbalancer` service type for their workloads. If user does create a loadbalancer type service on a baremetal cluster without external loadbalancer running, the service will stay stuck at `Provisioning` state forever.

5.4.2 Choice of internal loadbalancer

For the cluster there was no need for an external loadbalancer, since it was not build for a purpose involving major amount of traffic. For loadbalancing of traffic to pods through services it is possible to configure an ingress resource instead. However since this cluster consists of mulit-master setup, it required an internal loadbalncer, referred to as `localhost loadbalancer` for communication to masters API-servers.

[3] https://hub.docker.com/
[4] https://metallb.universe.tf/

There are two possible solution for this:

- First solution, which is usually preferred one, is having a separate dedicated loadbalancer machine running inside a cluster. Sole purpose of this machine is to loadbalance traffic between worker and master nodes. Only program running on the machine is loadbalancer. Most popular loadbalancer choices are Nginx[5] and HAProxy[6].
- Second solution is having a loadbalancer running on every worker node, so that when worker nodes tries to communicate with master nodes, it goes through loadbalancer on the worker node itself and from there is navigated to available master node API-server.

For our cluster's loadbalancer implementation the second solution was chosen, strictly due to not having enough resources. Since our on-premise cluster consists of only small number of machines. It would be wasteful to dedicate one of these machines solely as a loadbalancer. Therefore the second solution was chosen, meaning every worker node has a its own instance of loadbalancer running. Software chosen for the loadbalancer was HAProxy, instead of Nginx which is set as default on Kubespray. HAProxy was chosen based on the fact that products such as Openshift[7] which are build upon kubernetes chose to use HAProxy instead of Nginx as default. To configure Kubespray to use internal loadbalancer `loadbalancer_apiserver_localhost` variable needs to be set to `true`, and `loadbalancer_apiserver_type` to `haproxy`.

5.4.3 Running Kubespray

Kubespray setup followed the documentation. Ansible runs on python, so python3 needs to be installed on the admin machine. Since the setup is working with Kubespray repository on GitHub, git also needs to be installed. Documentation recommends to use python virtual environment.
It can be set up with following commands:

```
python3 -m venv venv
source venv/bin/activate
```

Next step is to clone the repository to the admin machine. And checkout the latest release branch. In this case it was 2.15.

```
git clone https://github.com/kubernetes-sigs/kubespray.git
cd kubespray
git checkout release-2.15
```

next install the dependencies:

```
pip install -r requirements.txt
```

copy the sample cluster configuration to a `mycluster` directory:

```
cp -rfp inventory/sample inventory/mycluster
```

[5]https://www.nginx.com/
[6]https://www.haproxy.org/
[7]https://www.openshift.com/

At last the inventory configuration is needed. Kubespray provides an easy way to configure inventory, by declaring a variable with machines' IP addresses and then uses inventory builder script to build our inventory with provided addresses.

```
declare -a IPS=(<comma-seperated-list-of-node-IP-addresses>)
CONFIG_FILE=inventory/mycluster/hosts.yaml
python3 contrib/inventory_builder/inventory.py ${IPS[@]}
```

After that the inventory files are build. The inventory can be accessed `inventory/mycluster` directory. Master nodes, worker nodes and etcd nodes, can be changed in the inventory to desired architecture. There is a constraint on etcd node amount, ensuring number of etcd nodes assigned is always an odd number. The first three of cluster nodes were configured to be master and etcd nodes as well, and the rest are the worker nodes.

Once everything is configured as needed, cluster setup can be triggered by running an Ansible playbook titled `cluster.yml`. The `-i` flag is used to specify the inventory file. It is recommended to run this command as a root user. However in this case, the use of root is not possible, since the only access provided by university was a user account with `sudo` privileges. Kubespray needs to be able to use `sudo` for running tasks on the servers. To do this Ansible needs to have a `-become` flag set which can be also set with `-b`. The access to the servers is already authorized from our admin machine via previously configured SSH keys. However for privilege escalation with `become` the user password is required. The `-K` flag can be used to input password. The user and password are the same on all servers.

Finally the cluster is set up by running the command:

```
ansible-playbook -i inventory/mycluster/hosts.yaml  -b cluster.yml -K
```

The command runs multiple Ansible playbooks, which are part of Kubespray projects on all our servers and sets up our highly available multi-master kuberentes cluster. This process takes around 10 to 20 minutes. Depending on machines running the processes and connection speed.

5.5 Accessing cluster

After the `cluster.yml` playbook is finished, the kubernetes cluster should be up and running. The `kubectl` command line tool can be used to access the cluster. A special file called `kubeconfig` is needed to use this tool to connect to the running cluster. `Kubeconfig` is a file generated by kuberenetes, that contains information about cluster, along with user information and authentication credentials. Kubespray does not provide this file, however kubernetes generates one for `kuberentes-admin` user which was responsible for cluster setup, and this file can be found on any of the master notes. Kubeconfig on the master node is depicted on 5.1.

To be able to access the cluster from the admin machine, this file needs to be copied from one of the master nodes. Right after successful kuberentes cluster setup, `kubeconfig` is located inside `/etc/kubernetes` directory under the filename `admin.conf`. This `admin.conf` file is however created by root user, therefore before it can be copied, a simple script needs to be run to change its owner to the current user and copy it to a newly created directory `.kube/`.The file also needs to be renamed from `admin.conf` to `config`. The `.kube` directory is the default path, where the `kubectl` tool looks for a `kubeconfig` under the name

of `config`. Therefore it is custom to create such directory on a master node and place the `kubeconfig` there. This can be also beneficial in the future while debugging certain internal kubernetes issues, since they might require direct access to the master node, and having `kubectl` ready is essential in such situations.

After that the `kubeconfig` is ready to be copied to the admin machine. A simple `scp` command can be used to copy the `kubeconfig` into `.kube` directory on the admin machine.

```
scp -P <port> <master-node>:.kube/config .kube/
```

After getting `kubeconfig`,if the command was run from home directory, `kubectl` will find the `kubeconfig`, since its default is `$HOME/.kube/config`. Otherwise it can be either moved to `$HOME/.kube/config`, which might not be optimal in case there already is a different `kubeconfig` for connecting to some other cluster. In that case there are two choices:

- `-kubeconfig=path/to/kubeconfig` option can be attached at the and of every `kubectl` command.
  `kubectl get nodes -A -kubeconfig=.kube/config`
- Or the path to kubeconfig can be set as an environmental variable inside the shell.
  `export KUBECONFIG=.kube/config`

After that the `kubeconfig` is on the admin machine and `kubectl` knows where to find it. There is a small last adjustment that needs to be done. `Kubeconfig` currently residing on the admin machine contains information on how to connect to the cluster, however it was taken from one of the cluster‘s master nodes, therefore its connection server‘s IP address to the cluster is a localhost address `127.0.0.1`. To be able to connect to the cluster from our admin machine, this address needs to be set inside `kubeconfig` to the address of one of our master nodes.

.kube > config

```
1  apiVersion: v1
2  clusters:
3  - cluster:
4      certificate-authority-data: LS0tLS1CRUdJTiBDRVJ
5      server: https://<master-node>:6443
6    name: cluster.local
7  contexts:
8  - context:
9      cluster: cluster.local
10     user: kubernetes-admin
11   name: kubernetes-admin@cluster.local
12 current-context: kubernetes-admin@cluster.local
13 kind: Config
14 preferences: {}
15 users:
16 - name: kubernetes-admin
17   user:
18     client-certificate-data: LS0tLS1CRUdJTiBDRVJUSU
19     client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURS
20
```

Figure 5.1: `kubeconfig/admin.conf` example

```
server: https://<master-node>:6443
```

Now it is finally possible to test connection to the kubernetes cluster.

```
$ kubectl cluster-info
Kubernetes control plane is running at https://<master-node>:6443

To further debug and diagnose cluster problems, use 'kubectl cluster-
    info dump'.
```

And availability of cluster's nodes can be checked:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node0	Ready	control-plane,master	11h	v1.20.4
node1	Ready	control-plane,master	11h	v1.20.4
node2	Ready	control-plane,master	11h	v1.20.4
node3	Ready	<none>	11h	v1.20.4
node4	Ready	<none>	11h	v1.20.4
node6	Ready	<none>	11h	v1.20.4
node7	Ready	<none>	11h	v1.20.4
node8	Ready	<none>	11h	v1.20.4

Figure 5.2: `kubectl get nodes` command output

5.6 Kubernetes Dashboard

Even though `kubectl` is a powerful command line tool that allows user to do pretty much everything around kubernetes cluster. Some people prefer to have some sort of graphical user interface as well. It is possible to get information about kubernetes resources with `kubectl describe` command, however it can be way more useful and transparent, when more data can be displayed at once in some form of interactive GUI containing elements as lists and graphs with filtering options. Kubernetes also has a deployable kubernetes dashboard, that provides just that. Dashboard allows users to easily navigate around cluster resources and components, create or apply new resources and workloads, view logs, and even SSH into containers or nodes.

Kubernetes dashboard does not come with base installation of kubernetes, and has to be deployed manually. To setup the kuberenetes dashboard, there are multiple resources needed. However most of these resources are provided by community in the kubernetes documentation[8].

Kuberetes dashboard configuration `yaml` file prepared by kubernetes community can be applied as described in the documentation with `kubectl apply` command.

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/
    v2.0.0/aio/deploy/recommended.yaml
```

This will create a bunch of resources such as:

[8] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

- `kubernetes-dashboard` namespace;
- service account with role, clusterrole, rolebinding and clusterrolebinding for that service account;
- a `kubernetes-dashboard` deployment and service;
- related secrets and configMap;
- and deployment and service for dashboard-metrics-scraper.

After these resources are created and running on the cluster. Kubernetes dashboard should be accessible. However the applied `kubernetes-dashboard` service is of type `ClusterIP`, meaning it will be only accessible from inside the cluster. To access the dashboard from outside without changing configuration, a `kubectl proxy` command can be run on the admin machine, which will make dashboard accessible on `http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services /https:kubernetes-dashboard:/proxy/`. However it will only be accessible from this one machine and will not be accessible at all after `kubectl proxy` command exits. Users probably want the dashboard to be accessible from anywhere in browser and without need of running `kubectl` commands.

To achieve that, the service type needs to be changed from `ClusterIP` to `NodePort`. This can be done with edit command:

```
kubectl edit svc kubernetes-dashboard -n kubernetes-dashboard
```

Actual port number can be specified with field `nodePort`, but if left unspecified, it will choose a viable port automatically.

`NodePort` type service can be accessed from outside, since kubernetes dashboard is now running on the cluster, its port can be checked with command:

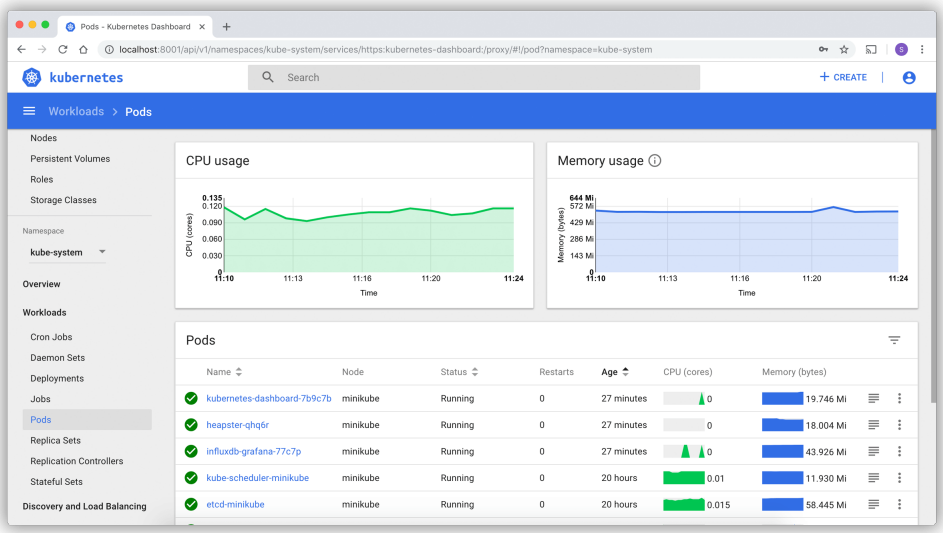

Figure 5.3: Kubernetes Dashboard

`kubectl get svc -n kubernetes-dashboard`

Kuberentes dashboard is running and accessible on URL `https://<worker-node-IP>:<service-nodePort>`.
Since `kubernetes-dashboard` is deployed with `https` protocol, it requires some sort of authentization to access. It is possible to deploy dashboard with `http` protocol instead, bypassing the authentication, however this is not recommended since it cause significant security concerns.

There are two possible ways of logging into to the dashboard right away:

- with the use of user's `kubeconfig` file,
- with the use of a bearer token.

Easy way of connecting is with `kubeconfig`, however `kubeconfig` needs to either contain username and password, or a token for authentication. `Kubeconfig` which was downloaded from the master node was however an `admin.conf` file, meaning it is a `kubeconfig` for `cluster-admin` user. `Admin.conf` does not contain bearer content, only client certificate, therefore it will not connect log in and instead display an `Internal error (500): Not enough data to create auth info structure.`

So initially to log into the kubernetes dashboard, a bearer token is required either way.

Applied `kuberentes-dashboard` configuration already created a service account with a bearer token. This token can be accessed by with `kubectl describe` command on the token from `kuberentes-dashboard` service account in the `kuberentes-dashboard` namespace.

```
kubectl get sa kubernetes-dashboard -n kubernetes-dashboard
kubectl describe secret <sa-token-secret-name> -n kubernetes-dashboard
```

Described secret shows token that can be used to access the kubernetes dashboard. However this token belong to a service account which has access only to the `kubernetes-dashboard` namespace. Therefore it is not able to view any other namespaces on the dashboard. This is of course not an optimal solution. Bearer token of any service account can be used to access the dashboard, however permissions of that service account also translate to what can be viewed on the dashboard. This is a good feature to ensure each user can only interact with resources which it has rights to and can not cause mischief in other places. Though initially it means admin cannot see everything in the dashboard. There are two ways admin can go from there to get full access:

- Use an existing token with admin privileges,
- grant `kubernetes-dashboard` service account admin privileges,
- create new admin user account.

5.6.1 Use an existing token with admin privileges.

Some resources running on a fresh cluster already need to have admin permissions to keep cluster alive and interact with other resources. One such resource is a `deployment-controller` service account. Therefore token belonging to this service account can be taken and added into `admin.conf/kubeconfig` file or just paste is as a bearer token to log into the dashboard with. Logging in with bearer token only, can be uncomfortable, since every time session is interrupted or expires, kubernetes dashboard requires you to log in again.

5.6.2 Grant `kubernetes-dashboard` service account admin privileges.

Instead of taking token from existing service account with admin privileges, it is possible to just grant those privileges to created `kubernetes-dashboard` service account. This can be done by defining a `clusterrolebinding` resource binding `kubernetes-dashboard` service account with `cluster-admin` clusterrole. This way it is even possible to skip the log in screen since the very service account running dashboard has already admin privileges. However accessing the kubernetes dashboard this way is not recommended for actual production clusters, since it may open the cluster up to security vulnerabilities. In production clusters, it is best practice to not grant the admin privileges to any other than essential components.

5.6.3 Create a new admin user service account.

Another way to access kubernetes dashboard is to actually create a new service account meant for an admin user. The service account can be created in the `kubernetes-dashboard` namespace. And just as described in previous option, bound to a `cluster-admin` clusterrole. This service account then exists with admin privileges, therefore its token can be used to access the dashboard. `Kubectl` command to access token directly:

```
kubectl -n kubernetes-dashboard get secret /
$(kubectl get sa/<new-admin-service-account> -n kubernetes-dashboard /
-o jsonpath="{.secrets[0].name}")  /
-o go-template="{{.data.token | base64decode}}"
```

This option is secure if the admin user has adequate knowledge and can be trusted.

5.7 Lens IDE

An alternative to a GUI based cluster access such as kubernetes dashboard is an application called Lens[9]. Unlike kubernetes dashboard, Lens is not a web based application, nor does it require to run cluster itself. Lens advertises itself as *The Kubernetes IDE*. It is a free software, available on each Mac, Windows and Linux platforms. Lens is built with electron, and takes advantage of `kubectl` tool for its functionality. It is an open-source project, available for contributions residing on Github[10] Lens provides the same features as `kubernetes-dashboard`, and more. Lens connects to the cluster with the use of `kubeconfig` file. And as an IDE, it is able to manage multiple clusters at the same time. Lens IDE has proven very useful during this bachelor thesis, therefore it deserves to be mentioned as valuable tool and alternative to `kubernetes-dashboard` and command-line `kubectl` tool. Lens IDE GUI is shown in 5.6.

5.8 Setup Monitoring

Monitoring is an important part of every running kuberentes cluster. Enabled monitoring in cluster can provide valuable info about health and status of clusters components or running workloads. Popular solution for monitoring kubernetes clusters is aforementioned `Prometheus`.

[9] https://k8slens.dev/
[10] https://github.com/lensapp/lens.

5.8.1 Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit[11]. It is also a graduated project by Cloud Native Computing Foundation, and it was a second hosted project by the CNCF after Kubernetes. That is why it is the choice for monitoring solution for this kubernetes cluster.

There are more than one way to set up a Prometheus monitoring for a kubernetes cluster. The more complicated way is to write all the required manifests and apply them in correct order based on dependencies. This however includes resources as configMaps, secrets, etc. Which can be needlessly complicated to manage. Prometheus Operator exists for this reason. Operator is a similar resource to replicaSet, in a way that, it basically defines a set of rules for its controller, to keep and eye on health and status of the resource, it is responsible for. A Prometheus operator therefore manages these Prometheus resources and saves user time and work. Prometheus configuration also does not usually differ between kuberenetes clusters. Therefore it is a practical solution to use already predefined Prometheus operator. Such valuable and common resources as Prometheus operator, are often maintained by the community to be easily applicable. The most comfortable way of deploying Prometheus operator on a kubernetes cluster is with a use of aforementioned tool called `Helm`[12]. Prometheus community maintains a helm chart containing Prometheus operator along with other monitoring related components called `kube-prometheus-stack`[13]. This allows user to deploy full monitoring stack on their kubernetes cluster just by installing this one chart. Components of this `kube-prometheus-stack` are:

prometheus-operator Prometheus components management;

kube-state-metrics Kuberentes components metrics collection;

[11] https://prometheus.io/docs/introduction/overview/
[12] https://helm.sh/
[13] https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack

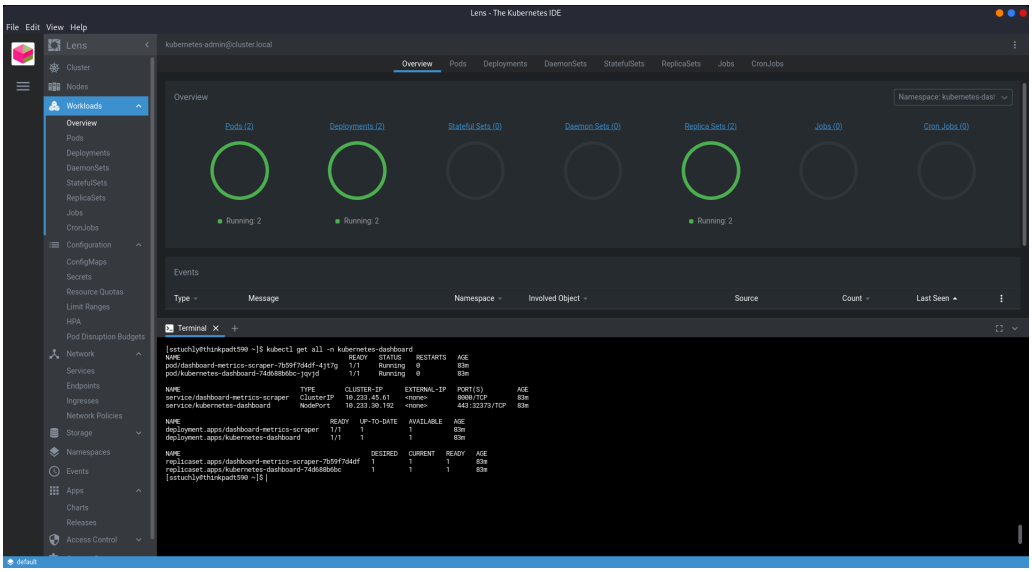

Figure 5.4: Lens IDE

prometheus-node-exporter DaemonSet that collects CPU and memory metrics for each node;

grafana Metric visualization dashboard;

alertmanager Manager for metrics based alerting.

To add the `kube-prometheus-stack` helm chart use following commands:

```
helm repo add prometheus-community  \
https://prometheus-community.github.io/helm-charts
helm repo update
```

Then create a namespace for `prometheus-stack` and install the chart in the namespace:

```
kubectl create ns monitoring
helm install <release name> prometheus-community/kube-prometheus-stack \
-n monitoring
```

Prometheus metrics and configuration can be viewed in the browser at URL `<node-IP>:<9090>`. 5.5 Prometheus can be configured by a configMap. This configMap contains configuration on where should Prometheus look for metrics. This configuration option can be very useful when trying to retrieve custom metrics. Within kubernetes, Prometheus supports a kuberentes service discovery feature, which makes it very easy to collect custom metrics from specified services or endpoints. A configMap with Prometheus rules, can be used to define rules for useful alerts based on metrics, such as defining CPU usage threshold on a certain node and firing alert once the received metrics reach the threshold.

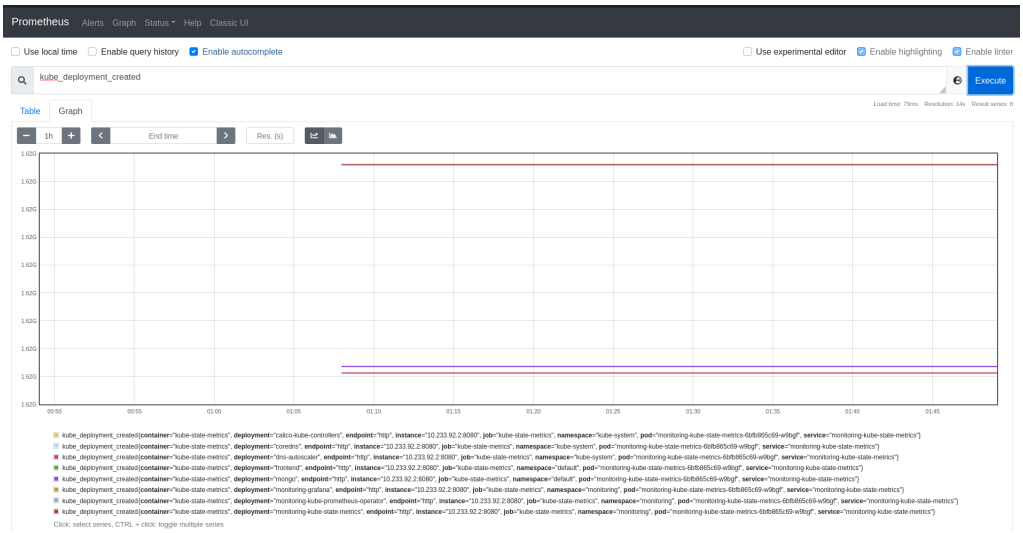

Figure 5.5: Prometheus web UI

5.9 Setup persistent storage

As described earlier in this thesis, kubernetes uses containers to run workloads on the cluster, data used by container get deleted with the container. Therefore to be able to

store data, declaration of persistent storage is needed. Also as mentioned earlier, for this purpose there are PersistentVolume and PersistentVolumeClaim resources, as well as a bit more advanced StorageClass resource. When using storage options provided by a some cloud provider, these cloud provider usually have a StarageClass configured for their storage options. Cluster created by this thesis however is owned by university and is made up of baremetal servers with storage disks attached. University prefers to not use any cloud provider for the storage but instead use the storage available on the servers as the only persistent storage for the cluster.

Managing persistent storage in kubernetes cluster can be a very complicated task, worth of another bachelor thesis by itself. That is why, it is in our best interest to use any available tool to make this process less complex, and do most of the work by itself. There are multiple tools for storage orchestration in kubernetes clusters. One of the most popular ones is Rook.

5.9.1 Ceph with Rook

Rook is an open source cloud-native storage orchestrator, providing the platform, framework, and support for a diverse set of storage solutions to natively integrate with cloud-native environments[14]. Rook is also one of the graduated projects of Cloud Native Computing Foundation[15]. Rook supports multiple storage solutions, their setup and maintenance. Probably the most popular storage solution for kubernetes clusters is Ceph[16]. Ceph is a highly scalable distributed storage solution supporting both block and object storage, as well as shared filesystems[31]. Although Ceph project itself does feature tools for Ceph storage configuration, such as `cephadm`. Configuring Ceph with Rook has its advantage, because user do not have to configure Ceph manually, which can be somewhat complex process, but Rook handles it for them. Users are able to mount volumes inside kubernetes cluster with help of just Rook. This great benefit and the fact that Rook is supported by Cloud Native Computing Foundation are the reasons for choosing Rook as this cluster‘s storage orchestration tool.

Before setup some of the basic Ceph components should be explained. Ceph consists of three main components:

Monitors Main responsibility of monitors is storing the main copy of a cluster map which allows Ceph daemon coordination. Monitors are also responsible for establishing cluster quorum. Every change in the rook cluster gets reported to the monitors. It is recommended to have at least three monitors in rook cluster. [31]

Manager Managers are responsible for monitoring runtime metrics and cluster state. At least two managers are required for a high availability cluster setup. [31]

Object storage devices Devices responsible for storing and retrieving data from the actual physical storage disk, running a filesystem. At least three Object storage devices (`OSDs`) are required for high availability cluster. [31]

[14] https://rook.io/docs/rook/v1.6/
[15] https://www.cncf.io/
[16] https://docs.ceph.com/en/latest/

5.9.2 Setup

The Rook documentation offers advice on how to configure Ceph storage for the kubernetes cluster with Rook. First step is to clone Rook code from its GitHub[17] repository:

```
git clone https://github.com/rook/rook.git
```

Then after navigating to the `rook/cluster/examples/kubernetes/ceph` directory, there are multiple configuration files available. Four of these files are required to install setup Rook in the kubernetes cluster. First file `crds.yaml` consists of multiple customs resource definitions such as `cephblockpool, cephclients, cephclusters` and so on, required by rook operator to manage Ceph storage. The `common.yaml` contains `rook-ceph` namespace, clusterroles, clusterrolebindings, service accounts, and many other required resources for Ceph cluster. And `operator.yaml` contains deployment for rook operator, which will be responsible for storage orchestration in the kuberentes cluster. These configuration files need to be created in the cluster:

```
kubectl create -f crds.yaml -f common.yaml -f operator.yaml
```

And finally `cluster.yaml` file needs to applied as well. This is where users can configure details about Ceph cluster and physical disks that it will consist of. In case of this cluster, it is to use mounted disk on every single node. The storage disk location is `/dev/sdb` path and is of size 1.8Gb on every node.This storage has to be of `raw` type, since `Ceph` will not register the disks with any mounted filesystem. Users can choose here to either use storage on every node, or configure which select nodes to use. This cluster uses disks on all nodes, therefore `useAllNodes` and `useAllDevices` are both set to true. Number of monitors is also kept at three, as is recommended, and it is not allowed to have more than one on the same node. `monitoring:enabled:true` variable can be set in case the kubernetes cluster has already configured Prometheus monitoring. Also Ceph dashboard is enabled by default, which is a desirable feature.

[17] https://github.com/rook/rook

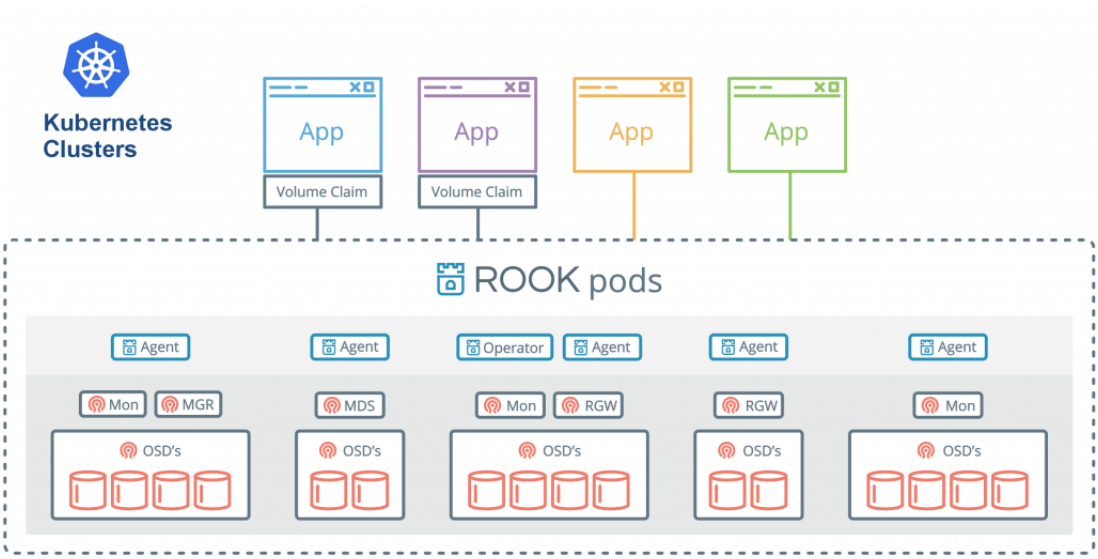

Figure 5.6: Rook cluster architecture [24]

```
kubectl create -f cluster.yaml
```

Rook takes a while to set everything up. It is recommended to check progress of setup with command:

```
kubectl get pods -n rook-ceph
```

After all pods are in either `Completed` or `Running` state. It is time to define example Storage Class for block storage.There is already prepared configuration file in `csi/rbd/` directory called `storageclass.yaml`. This file defines a storage class for ceph block storage. Create the storage class by:

```
kubectl create -f csi/rbd/storageclass.yml
```

Then is is possible to run ceph toolbox pod with a command:

```
kubectl create -f toolbox.yaml
```

When the `rook-ceph-tools` pod is `Running`, it can be used to check status of the Ceph cluster. To to that, it is possible to open a bash shell inside of it by either clinking the exec into container button on top right of Kuberenetes dashboard, or Lens IDE, or by `kubectl` command:

```
kubectl exec -it <rook-ceph-tools-pod-name> -n rook-ceph -- bash
```

Inside the pod show status of Ceph cluster with:

```
ceph status
```

Another useful feature of Ceph storage is Ceph dashboard. This is already deployed since it was specified by default in the `cluster.yaml` file, and was not altered. To access the dashboard, there are multiple options, but the easiest one is to create a new `NodePort` service as specified in Rook documentation.

Use this yaml file 5.7 to create a `nodePort` type service and then check running services and its assigned port.

```
kubectl -n rook-ceph get svc
```

Ceph dashboard 5.8 should then be accessible from outside the cluster on URL `<node-IP>:<new-nodePort-service-port>`.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: rook-ceph-mgr-dashboard-external-https
  namespace: rook-ceph
  labels:
    app: rook-ceph-mgr
    rook_cluster: rook-ceph
spec:
  ports:
  - name: dashboard
    port: 8443
    protocol: TCP
    targetPort: 8443
  selector:
    app: rook-ceph-mgr
    rook_cluster: rook-ceph
  sessionAffinity: None
  type: NodePort
```

Figure 5.7: `nodePort` service for ceph dashboard

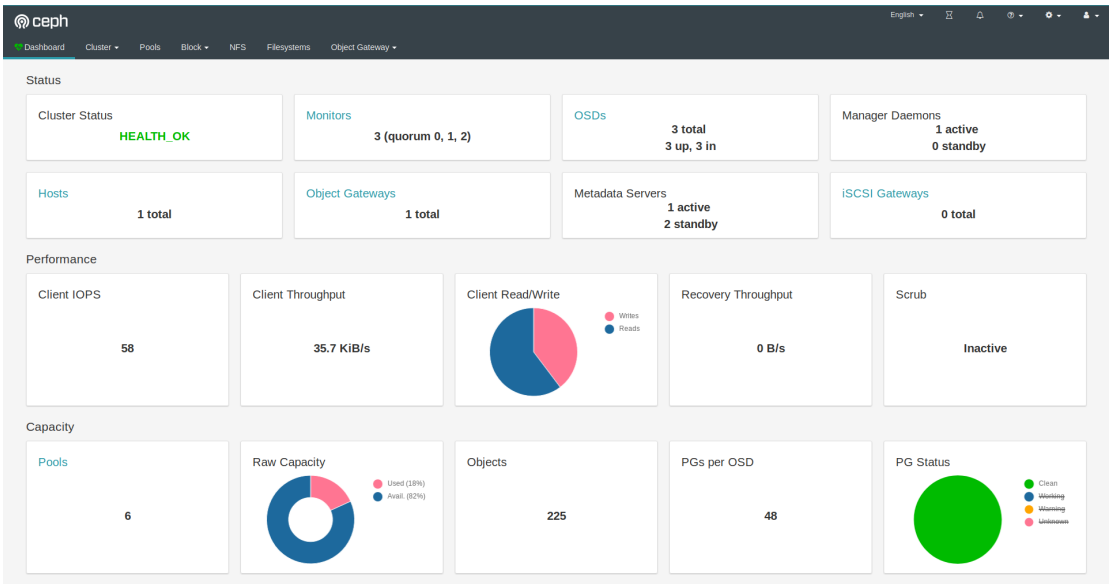

Figure 5.8: Ceph dashboard[18]

Chapter 6

Testing

When the kubernetes cluster is successfully running, it is time test out some of its features. To test kubernetes cluster functionality, an example application is needed to be deployed on the cluster and then experimented with. Such example application is provided on kubernetes documentation.[25] It is a simple PHP Guestbook application with MongoDB.

Testing scenarios:

- Deploy application and scale its parts up and down;
- Set affinity on the application making it only run on two worker nodes;
- Scale application up and disconnect one of the nodes the app is running on;
- Disconnect master node and see what node is in charge.

6.1 Deploying an example application

First step is to deploy application to the cluster. Since the example application is provided by kubernetes, it is best to use their configuration files for its deployment.

```
kubectl apply -f https://k8s.io/examples/application/guestbook/mongo-
    deployment.yaml
kubectl apply -f https://k8s.io/examples/application/guestbook/mongo-
    service.yaml
kubectl apply -f https://k8s.io/examples/application/guestbook/frontend-
    deployment.yaml
kubectl apply -f https://k8s.io/examples/application/guestbook/frontend-
    service.yaml
```

After this it can be checked that services and deployments were created. There are three `frontend` pods running and one MongoDB pod.

The application is not accessible from outside the cluster yet, therefore its `frontend` service type needs to be changed to `NodePort` with a random port. After then application can be accessed at URL `<localhost>:<generated-port>` 6.2.

```
NAME                        READY   STATUS    RESTARTS   AGE
frontend-848d88c7c-52xwn    1/1     Running   0          7s
frontend-848d88c7c-ft7jp    1/1     Running   0          7s
frontend-848d88c7c-wg2xz    1/1     Running   0          7s
mongo-75f59d57f4-sm2sm      1/1     Running   0          9s
```

Figure 6.1: Application pods running

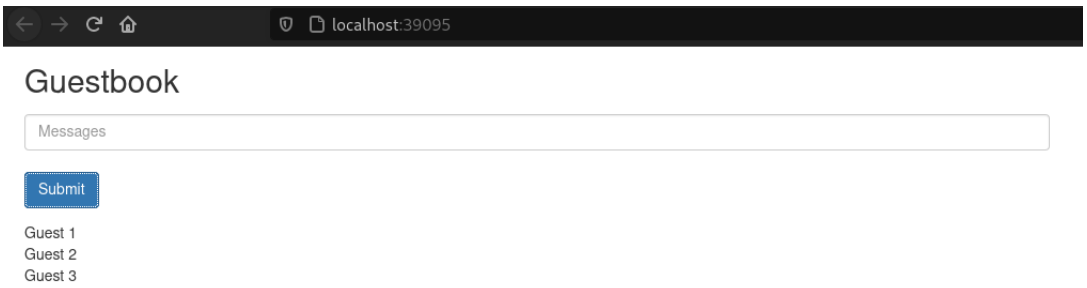

Figure 6.2: Guestbook frontend

6.2 Scaling application workload

Now testing scalability of the application. If there is a need for expanding the `frontend` workload, there is an easy way to handle this. Scale the `frontend` deployment up to 8 replicas.

```
kubectl scale deployment frontend --replicas=8
```

Kubernetes accepts the change and spins up 5 more replicas of `frontend` pods to satisfy the requirement 6.3.

6.3 Removing node running application workload

Now when there are running 8 running pods, and they all run on Node8 6.3. Therefore the next is behaviour to test what happens when Node8 is removed from cluster. Disable the node by draining it:

```
kubectl drain node8 --ignore-daemonsets --delete-local-data --force
```

After short while,`SchedulingDisabled` is set as `node8` status 6.4.
And the pods from node8 are being created on other available nodes 6.5.

```
NAME                        READY   STATUS    RESTARTS   AGE     IP              NODE
frontend-848d88c7c-4tz7q    1/1     Running   0          8s      10.233.69.17    node8
frontend-848d88c7c-52xwn    1/1     Running   0          13m     10.233.69.14    node8
frontend-848d88c7c-7mw6f    1/1     Running   0          8s      10.233.69.21    node8
frontend-848d88c7c-ft7jp    1/1     Running   0          13m     10.233.69.15    node8
frontend-848d88c7c-g42z5    1/1     Running   0          8s      10.233.69.18    node8
frontend-848d88c7c-j6lkx    1/1     Running   0          8s      10.233.69.20    node8
frontend-848d88c7c-t8hb8    1/1     Running   0          8s      10.233.69.19    node8
frontend-848d88c7c-wg2xz    1/1     Running   0          13m     10.233.69.13    node8
mongo-75f59d57f4-25d5s      1/1     Running   0          8m57s   10.233.69.16    node8
```

Figure 6.3: Application scaled

```
NAME    STATUS                     ROLES                  AGE   VERSION
node0   Ready                      control-plane,master   12h   v1.20.4
node1   Ready                      control-plane,master   12h   v1.20.4
node2   Ready                      control-plane,master   12h   v1.20.4
node3   Ready                      <none>                 12h   v1.20.4
node4   Ready                      <none>                 12h   v1.20.4
node6   Ready                      <none>                 12h   v1.20.4
node7   Ready                      <none>                 12h   v1.20.4
node8   Ready,SchedulingDisabled   <none>                 12h   v1.20.4
```

Figure 6.4: Node8 has been drained.

```
NAME                        READY   STATUS    RESTARTS   AGE     IP              NODE
frontend-848d88c7c-2cjp7    1/1     Running   0          6m23s   10.233.108.4    node6
frontend-848d88c7c-c4khl    1/1     Running   0          6m23s   10.233.92.3     node3
frontend-848d88c7c-gfmvp    1/1     Running   0          6m23s   10.233.105.5    node4
frontend-848d88c7c-l8s58    1/1     Running   0          6m23s   10.233.95.2     node7
frontend-848d88c7c-mf9l8    1/1     Running   0          6m23s   10.233.92.5     node3
frontend-848d88c7c-mgv92    1/1     Running   0          6m23s   10.233.95.3     node7
frontend-848d88c7c-tt2j5    1/1     Running   0          6m23s   10.233.108.5    node6
frontend-848d88c7c-xm5qb    1/1     Running   0          6m23s   10.233.105.4    node4
```

Figure 6.5: Pods redeployed to available nodes

To be able to use the drained node again, run command:

```
kubectl uncordon node8
```

`Node8` status is back to `Ready`.

6.4 Use node affinity to schedule pods on certain node

Kubernetes allows manipulating where the pods should be scheduled. For example user might want to run certain pods on node with more RAM. In the case `frontend` pods should be run on `node8` since it is the only one with 256GB RAM . To achieve this it is possible to specify label on the node. For this case it can be random label such as `nodeRAM`

```
kubectl label nodes node8 nodeRAM=256GB
```

Check if the label was applied:

```
kubectl get node/node8 --show-labels
```

Next step is to add node-affinity into the `frontend` deployment. There are 2 options:

requiredDuringSchedulingIgnoredDuringExecution 6.6 After draining node8, pods will remain in `Pending` state, since they require set label on the node to be able to be scheduled there 6.8.

preferredDuringSchedulingIgnoredDuringExecution 6.7 After `node8` is drained, pods will schedule on any other node. `Weight` argument is required with this option, however it is not very significant in this example since there is only one preference set. In case of more, `weight` plays a role in calculated which node scheduler schedules the pod to.

```yaml
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: nodeRAM
          operator: In
          values:
          - 256GB
```

Figure 6.6: Required affinity example.

```yaml
affinity:
  nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      preference:
        matchExpressions:
        - key: nodeRAM
          operator: In
          values:
          - 256GB
```

Figure 6.7: Preferred affinity example.

```
kubectl edit deploy/frontend
```

Add node affinity into `spec:template:spec:`.

After affinity rule applied to deployment, nodes are trying to run but get stuck on Pending state 6.8. And after getting node8 back up, any additional pods will be run on `node8` 6.9.

```
NAME                        READY   STATUS    RESTARTS   AGE   IP          NODE
frontend-85dfdc56d4-6lx4m   0/1     Pending   0          7s    <none>      <none>
frontend-85dfdc56d4-grbbf   0/1     Pending   0          7s    <none>      <none>
frontend-85dfdc56d4-vklll   0/1     Pending   0          7s    <none>      <none>
```

Figure 6.8: Pending pods.

```
NAME                        READY   STATUS    RESTARTS   AGE   IP             NODE
frontend-85dfdc56d4-6lx4m   1/1     Running   0          76s   10.233.69.24   node8
frontend-85dfdc56d4-grbbf   1/1     Running   0          76s   10.233.69.23   node8
frontend-85dfdc56d4-vklll   1/1     Running   0          76s   10.233.69.22   node8
```

Figure 6.9: Pods deployed on with label.

Chapter 7

Conclusion

The goal of the first part of the thesis was to explore containers, container orchestration, Kubernetes, its architecture and components, as well as Ansible automation tool and then design an optimal architecture for kubernetes-based on-premise mini-cloud. All mentioned topics were described in the thesis and after that cluster architecture was designed. This design was then presented in form of a diagram and further described.

The second part of this Bachelor thesis focused on implementation of prepared design upon machines that form a kubernetes cluster. Kubernetes cluster installation tool called Kubespray was used to configure and install kubernetes onto university servers. Bash scripts and Ansible playbooks were written for the purpose of preparing the servers for kubernetes cluster installation, as well as some post installation tasks, such as retrieving `kubeconfig`. After successful kubernetes cluster setup, cluster monitoring has been configured. Prometheus monitoring toolkit was used for monitoring setup, combined with other required tools such as `kube-state-metrics`. Chosen storage solution for kuberentes cluster was `Ceph`. Ceph cluster has been configured and set up with the help of storage orchastrator called `Rook`. The last part of the implementation part of this thesis was focused on testing configured running kubernetes cluster functionality by deploying an example application on the cluster and experimenting with scaling, scheduling manipulation, and recovery of the application workload. Additionally a short documentation guide was created for users of the kuberentes cluster, describing basic usage of the cluster and application deployment.

Thesis goal of creating an application-computation on-premise mini-cloud based on kubernetes was fulfilled, however there are still certain components such as implementation of user authentication system that did not fit into this thesis, but can be implemented in the future as another feature of the cluster.

Bibliography

[1] *About the Open Container Initiative* [online]. The Linux Foundation [cit. 2021-01-18]. Available at: https://opencontainers.org/about/overview/.

[2] *Docker Architecture* [online]. Aqua Security Software Ltd. [cit. 2021-01-18]. Available at: https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/.

[3] *Docker overview* [online]. [cit. 2021-01-18]. Available at: https://docs.docker.com/get-started/overview/.

[4] *How Ansible Works* [online]. [cit. 2021-01-18]. Available at: https://www.ansible.com/overview/how-ansible-works.

[5] *Introduction to Kubernetes architecture* [online]. Red Hat [cit. 2021-01-18]. Available at: https://www.redhat.com/en/topics/containers/kubernetes-architecture.

[6] *Kube-state-metrics* [online]. [cit. 2021-01-18]. Available at: https://github.com/kubernetes/kube-state-metrics.

[7] *Mesos Architecture* [online]. The Apache Software Foundation [cit. 2021-01-18]. Available at: http://mesos.apache.org/documentation/latest/architecture/.

[8] *Podman* [online]. [cit. 2021-01-18]. Available at: https://podman.io/.

[9] *Prometheus overview* [online]. [cit. 2021-01-18]. Available at: https://prometheus.io/docs/.

[10] *What is a Container?* [online]. [cit. 2021-01-18]. Available at: https://www.docker.com/resources/what-container.

[11] *What is container orchestration?* [online]. Red Hat [cit. 2021-01-18]. Available at: https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

[12] *What is CRI-O?* [online]. [cit. 2021-01-18]. Available at: https://cri-o.io/.

[13] *What is Podman?* [online]. [cit. 2021-01-18]. Available at: http://docs.podman.io/en/latest/.

[14] *Kubernetes vs Mesos vs Swarm* [online]. Sumo Logic, may 2019 [cit. 2021-01-18]. Available at: https://www.sumologic.com/insight/kubernetes-vs-mesos-vs-swarm/.

[15] *Affinity and anti-affinity* [online]. December 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity.

[16] *ConfigMaps* [online]. December 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/concepts/configuration/configmap/.

[17] *Kubernetes Components* [online]. June 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/concepts/overview/components/.

[18] *Operating etcd clusters for Kubernetes* [online]. November 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/.

[19] *Set up High-Availability Kubernetes Masters* [online]. November 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/.

[20] *Stacked etcd topology* [online]. May 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/.

[21] *Storage Classes* [online]. November 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/concepts/storage/storage-classes/.

[22] *Taints and Tolerations* [online]. November 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/.

[23] *What is Kubernetes?* [online]. October 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[24] *Ceph Storage* [online]. The Linux Foundation, 2021 [cit. 2021-05-12]. Available at: https://rook.io/docs/rook/v1.6/ceph-storage.html.

[25] *Example: Deploying PHP Guestbook application with MongoDB* [online]. April 2021 [cit. 2021-05-12]. Available at: https://kubernetes.io/docs/tutorials/stateless-application/guestbook/.

[26] *Persistent Volumes* [online]. January 2021 [cit. 2021-01-18]. Available at: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[27] Basyildiz, B. *A Brief History of Container Technology* [online]. Section, august 2019 [cit. 2021-01-18]. Available at: https://www.section.io/engineering-education/history-of-container-technology/.

[28] Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*. 2014, vol. 1, no. 3, p. 81–84. DOI: 10.1109/MCC.2014.51.

[29] Castro, J. et al. *Don't Panic: Kubernetes and Docker* [online]. December 2020 [cit. 2021-01-18]. Available at: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/.

[30] Christoph Blecker, P. S. *Metrics Server* [online]. [cit. 2021-01-18]. Available at: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/metrics-server.md.

[31] Codegiant, T. *How to Deploy a Ceph Cluster on Kubernetes With Rook* [online]. codegiant, october 2020 [cit. 2021-05-12]. Available at: https://blog.codegiant.io/how-to-deploy-a-ceph-cluster-on-kubernetes-with-rook-daf24e6db914.

[32] Diego Ongaro, J. O. *In Search of an Understandable Consensus Algorithm (Extended Version)* [online]. [cit. 2021-05-12]. Available at: https://raft.github.io/raft.pdf.

[33] Eldridge, I. *What Is Container Orchestration?* [online]. New Relic, july 2018 [cit. 2021-01-18]. Available at: https://blog.newrelic.com/engineering/container-orchestration-explained/.

[34] Estes, P. *RunC: The little container engine that could* [online]. opensource.com, august 2016 [cit. 2021-01-18]. Available at: https://opensource.com/life/16/8/runc-little-container-engine-could.

[35] Hong, Y.-J. *Introducing Container Runtime Interface (CRI) in Kubernetes* [online]. December 2016 [cit. 2021-01-18]. Available at: https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/.

[36] Jean Mathieu Saponaro, J. M. *Collecting metrics with built-in Kubernetes monitoring tools* [online]. Datadog, march 2020 [cit. 2021-01-18]. Available at: https://www.datadoghq.com/blog/how-to-collect-and-graph-kubernetes-metrics/.

[37] Keecheril, J. *Ansible Architecture* [online]. January 2020 [cit. 2021-01-18]. Available at: https://dev.to/keecheriljobin/ansible-architecture-working-co9.

[38] Kienzler, S. *Welcome To The Container Jungle: Docker vs. containerd vs. Nabla vs. Kata vs. Firecracker and more!* [online]. April 2020 [cit. 2021-01-18]. Available at: https://www.inovex.de/blog/containers-docker-containerd-nabla-kata-firecracker/.

[39] Ladha, P. *Demystifying High Availability in Kubernetes Using Kubeadm* [online]. Velotio Technologies, 2019 [cit. 2021-01-18]. Available at: https://www.velotio.com/engineering-blog/demystifying-high-availability-in-kubernetes-using-kubeadm.

[40] Sayfan, G. *Mastering Kubernetes*. 2nd ed. Packt, 2016. ISBN 9781788999786.

[41] sigs kubernetes. *Kubespray* [https://github.com/kubernetes-sigs/kubespray]. GitHub, 2021.

[42] Watt, S. *Kubernetes Architectural Overview* [online]. Red Hat [cit. 2021-01-18]. Available at: https://www.linux.com/news/what-makes-kubernetes-cluster/.