UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Enabling OpenMPI workloads on bare-metal infrastructure using Kubernetes

Author:
Mykola BILIAIEV

Supervisor:
Mr. Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Enabling OpenMPI workloads on bare-metal infrastructure using Kubernetes

by Mykola BILIAIEV

Abstract

The purpose of this bachelor's thesis is to design and develop a software architecture for HPC clusters to support MPI-based workloads. We will demonstrate how many abstractions such a complex solution involves and how the cluster works from the software perspective. As a result, we will configure a two-node cluster, develop software architecture for it, and run workloads on top of it. We will use many technologies to develop our solution. To better understand the architecture, we will give a background on the most complex parts like Kubernetes and Docker.

§

Acknowledgements

I want to thank my supervisor Oleg Farenyuk for mentoring me and for all the great knowledge I gathered over the years. I would also like to thank my classmates who were back to back with me over the last four years. In particular, I want to thank the people who supported me during my work on this thesis - Hermann Yavorskyi, Volodymyr Chernetskyi, and Vladyslav Ursul; you did a great job. Last but not least, I would like to thank Ukrainian Catholic University for the opportunity to study on such an incredible program.

Contents

List of Figures

List of Tables

List of Abbreviations

CIDR	**C**lassless **i**nter-**d**omain **R**outing (CIDR)
CLI	**C**ommand **L**ine **I**nterface
CRD	**C**ustom **R**esource **D**efenition
HPC	**H**igh **P**erformance **C**omputing
JSON	**J**avaScript **O**bject **N**otation
K8S	Kuberenete**s**
MPI	**M**essage **P**assing **I**nterface
MPP	**M**assively **P**arallel **P**rocessing

Chapter 1

Introduction

At the very beginning, computers were primitive, it took hours to complete some simple arithmetic tasks, and one character mistake in code could cost hours of work. One of the first machines, ENIAC (1945), consumed up to 167 square feats – Council, 2005.

With time computers became more and more powerful. Engineers worked hard to build more efficient hardware and make computer parts as small as possible. New approaches to extract more performance appeared with each day.

As time went on, the idea of dividing tasks into separate pieces and running those at the same time arose. This approach would require writing more challenging code, but at the same time, it would give a considerable performance boost. The emergency of shared memory multiprocessors in the 60's changed the approach to writing efficient programs. Programs that supported the usage of multiple cores were much more efficient. Parallel computing became a standard.

Multiprocessors have their limitations. Whenever there is a need to scale the system, the solution is to buy new machines or upgrade hardware on nodes. It comes with a cost of a longs downtime or an unexpected budget. Mainframes (*What is a mainframe*) tried to solve this issue. Those machines could work non-stop for years. This solution also allows changing hardware parts without downtime; however, sooner or later, the system would still reach hardware limitations. In the early 1990s, the first clusters appeared. A cluster consists of many nodes with similar hardware. It connects in a fast local area network, with some software on top that allows running tasks across all nodes. This approach allows clusters to scale horizontally, utilizing all of compute nodes.

As the Massively Parallel Processors and clusters became popular, the need for some parallel computing standards arose. One of the most popular solutions is MPI (Message passing interface), a single API interface that simplifies the development of parallel programs – *Itro to mpi*.

Clusters started appearing in many countries and Fields. Competition for the best performance began. In 1993 the community decided to make this competition official and created *TOP 500 list*. This list shows the top 500 most performance clusters in the world.

As part of this thesis, we collaborated with the Institute Of Condensed Matter Physics who owns an HPC cluster. This cluster is far from TOP500, but it is still one of the biggest in Ukraine. This collaboration allows us to experiment with real cluster hardware and apply cluster architectures in practice.

1.1 Problem

The cluster creation process is long and tedious. It starts with building the physical environment for the cluster: cooling and ventilation, power supply, and security. The next step is to build nodes, choose proper hardware with a sufficient amount of resources. Good network performance is crucial; a dedicated solution like InfiniBand network can be a great choice. Then comes the tricky part, how to run actual workloads in this cluster.

The software part is as important as hardware. There are a variety of architectures to utilize maximum performance from the cluster. We want to explore this world and create an architecture that will be compatible with any HPC cluster and give ability run MPI workloads.

1.2 Goal

This thesis aims to build an architecture that allows running MPI workloads on the bare-metal Kubernetes cluster. The architecture objectives are the following:

- Fault tolerance – Cluster must be durable, and failure of any single node should not impact the overall functioning of the system
- Observability – Ability to see the state of MPI workloads, collect results and analyze historical data.
- Scalability – Whenever there is a lack of computation power, the cluster should support scale without downtime. Section 4.3.3 goes in details about implementation.
- Maintenance – The ability to update the cluster and keep all system components secure should be easy.

Institute Of Condensed Matter Physics lets us utilize a two-node cluster as part of our thesis – Fig.4.2

1.3 Thesis structure

We start with chapter 1 where we describe our goals and objectives. In chapter 2 we look over the main theory and technologies we utilize in our cluster. In chapter 3 we will explore different existing solutions. In chapter 4 we showcase our solution, describe how it works. In chapter 5 we run some workloads in the cluster and observe successful results.

Chapter 2

Background Information

2.1 HPC

High-Performance Computing is a way to process an immense amount of data and perform demanding calculations at an incredible speed. HPC cluster requires special software and hardware to maximize performance. A single cluster consists of tenths to thousands of nodes that utilize software to compute tasks in parallel.

HPC clusters play a significant role in today's world. They are used to calculate complex tasks in different fields. Many types of research rely on those compute powers. Those include molecular modeling, quantum mechanic, physics, Financial institutions, and many others.

2.2 Containerization

Containerization allows to wrap and isolate applications without significant overhead. This technology allows to easily manage application dependencies, manage software versions, seamlessly deploy and update environments.

Containerization is becoming increasingly popular, and there is a giant ecosystem surrounding it. The most popular ones are Docker (*Docker*) and Kubernetes (*Kubernetes*). Those two will be looked at in great detail, as they play a significant role in the architecture solution.

2.2.1 Docker history

The fact is that containers have been in computer science for a long time. The first parts of containerization appeared in UNIX V7 year 1979. This distribution introduced chroot, one of the three most essential parts of containerization, more on that later – *A Brief History of Containers: From the 1970s Till Now*.

In 2008 something very similar to modern containers appeared LXC (Linux Containers). It introduced the two missing parts, namespaces, and cgroups. The problem was that it was pretty hard to manage those containers, requiring some deep expertise in Linux – *A Brief History of Containers: From the 1970s Till Now*.

The most popular containerization software is Docker. Solomon Hykes created Docker in 2013. Docker allows the functionality of LXC with fantastic simplicity. Now anyone can use containers for their daily tasks – *A Brief History of Containers: From the 1970s Till Now*.

2.2.2 Docker implementation

The ecosystem of Docker consists of client-oriented parts and container implementation.

The actual containerization is provided by three main kernel components:

- Cgroups - this component allows isolating resources per container. It defines how much CPU/RAM should be available for the application. From the inside of the container, it will only access that predefined amount of computes power. Whenever a container tries to exceed those resources, it will either be throttling or killed by OOM killer[1] – *Cgroups*.
- Chroot - allows isolation of the file system for the container. It will change the root directory of a container and limit access only to this directory's scope – *Chroot*.
- Namespaces - namespaces allow to isolate many parts of the container. For example, network namespace isolates containers to its only network interface or PID[2] namespace, gives visibility of the root and children's process starting from number 1 – *Namespaces*.

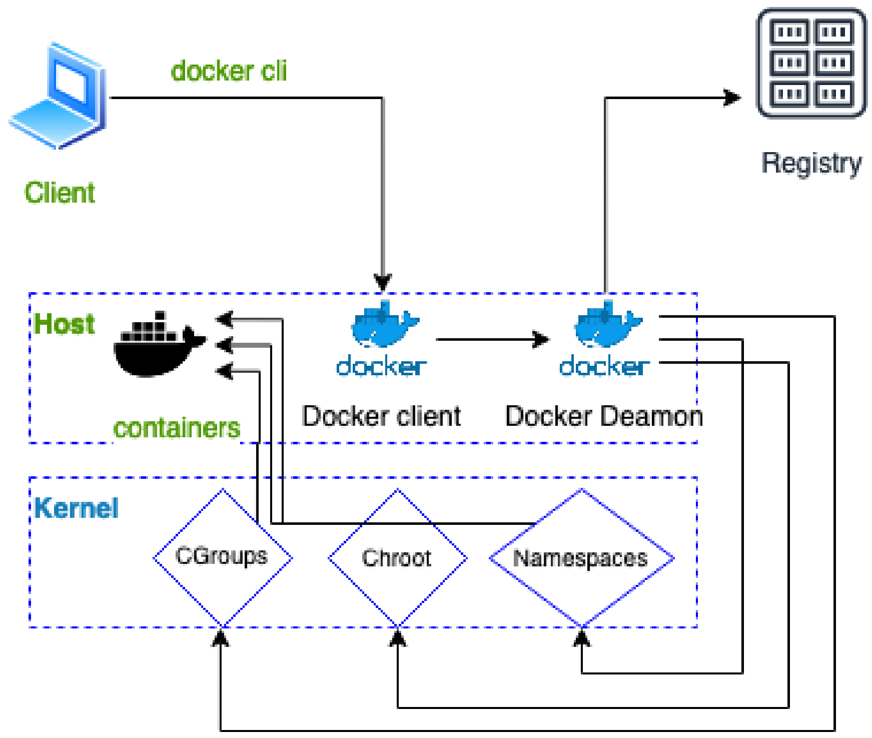


FIGURE 2.1: Docker diagram

Docker is a wrapper around those tools. Whenever we send commands through CLI, Docker will do all of the hard work - configure namespace, isolate file-system, provide CPU/RAM resources.

[1]Out of Memory Killer is a special process that runs in Linux kernel, it make sure that all critical system component has enough memory, and kills non-important ones is system is low on memory.
[2]PID is a unique process id

2.3 Kubernetes

Kubernetes is a container orchestrator. In simple words, k8s allows the management of container workloads without pain. All services are in YAML[3]formatted files with descriptions of how they should behave. For example, specify the number of replicas required, CPU/RAM resources needed, which code to execute, define execution parameters.

Kubernetes consists of many components that communicate with each other. This approach makes the system more reliable and faults tolerant. Single component failure will not impact overall cluster functionality. The (Luksa, 2018) book does a good job of introducing k8s.

Kubernetes consists of the following components:

- Master node – is the main node that runs all of the core k8s components. Those components are called the control plane. The control plane makes all decisions about the cluster state.
- Pod – is the smallest entity in k8s. It usually represents one running container. The developer can configure a multi-container pod, but there is usually no need for that.
- API Server – is the core component of the k8s Control Plane. Most of the communication goes through the API server. The API server is the only part that exposes Restful API.
- ETCD – is the persistence of the control plane, a key-value database, where k8s stores information about the current state, which resources should be deployed. The Control plane makes sure that all resources described in ETCD are running in the cluster.
- Scheduler – is the component responsible for scheduling pods. Scheduler looks into ETCD and finds new pods, and then it assigns a specific node to each pod.
- Kubelet – this component runs on each node in the cluster. It is responsible for reading information about pods from ETCD and deploying those containers on the node.
- Controller Manager – this component under the hood runs many controllers. Each of the controllers is responsible for some part of the system. For example Replication controller continuously checks if any of the replica sets needs to be increased or decreased.

2.3.1 CRD

Kubernetes has many native Resource Definitions. Those resources allow us to configure our k8s environment.

All Resource Definitions are native to the k8s API and can not be extended; for this purpose, CRDs are used. CRD stands for Custom Resource Definition. This feature allows us to extend the native Kubernetes API.

[3]YAML is a human readable language, it is usuallly used for configuration files.

2.3.2 Pod life cycle

To understand a bit deeper how k8s works we are going to explore what happens when user tries to create a new pod:

1. Create a YAML file with pod description.
2. Use a kubectl utility to send data to the API server.
3. API Server validates the request and puts information about this pod into ETCD.
4. Scheduler sees changes in ETCD and finds a proper node for this pod. Suppose a node with the required amount of resources is available. In that case, the scheduler assigns a node to this pod by putting this information in ETCD. If there are no resources for this pod, it will stay pending until more resources are available.
5. After scheduler assigns pods to a specific node, kubelet running on that node sees this information and deploys pod.

2.3.3 Operator and controller

Understanding an operator pattern and its implementation is essential in the scope of this thesis.

Controller is a single binary that endlessly runs in the control plane, watches for specific events, and reacts to them.

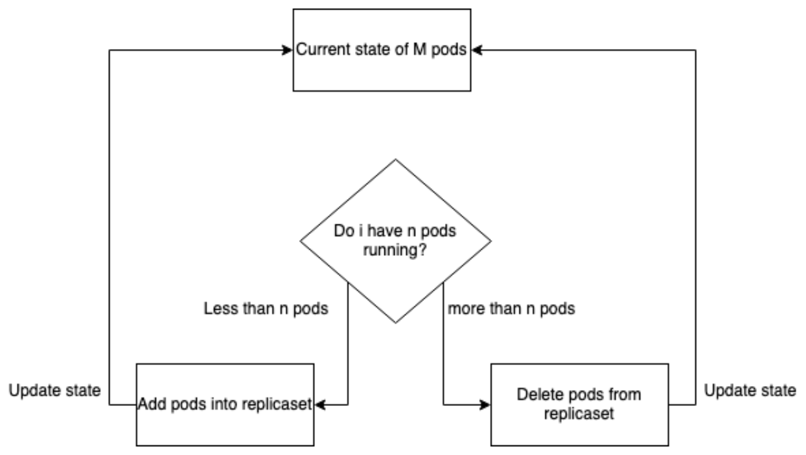


FIGURE 2.2: Replica set controller

Figure 2.2 represents a simplified flow of a replica set controller. One of its purposes is to ensure that the required amount of replicas is running at any moment.

An operator is a very similar concept. An operator is a controller, but the controller is not always an operator. The primary purpose of the operator is to solve some domain-specific logic. The operator also utilizes the concept of CRDS.

The combination of controller and CRDS is what we call an operator. As part of this thesis, we will deploy a few operators: Prometheus Operator, MPI operator. The second one is the crucial one.

2.4 MPI

Message Passing Interface (MPI) (*Itro to mpi*) is one of the standardized communication protocols for parallel computing. It is prevalent in HPC. Mpi can run on a single computer by spawning multiple processes or in the cluster by spawning processes on other nodes.

When we start a MPI program, we usually define the amount of process we require. The program starts with a single process. After MPI Initialization, it spawns the N-1 child process with a unique id from 0 to N.

Each child knows about its rank (child id), and based on this number, program knows how to behave.

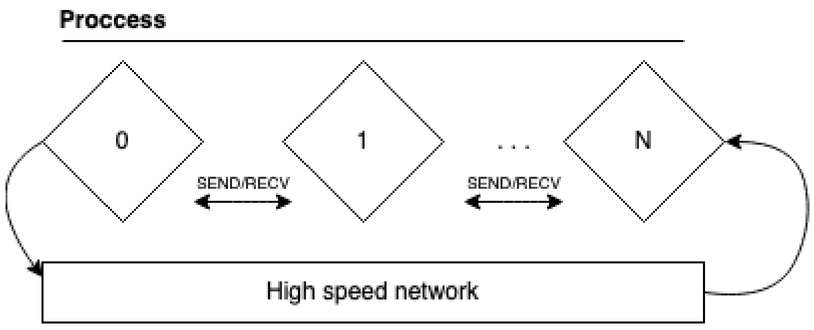


FIGURE 2.3: MPI proccess

2.4.1 Mpi in kubernetes

To deploy MPI, we will use *MPI Operator*. This operator will make the process of creating new MPI workloads easy. Figure 2.4 shows MPI operator architecture.

After the operator is deployed (Section 4.5) all of its components are available in the cluster.

How it works:

1. Config Map – controller creates a k8s resource ConfigMap. The ConfigMap is a JSON file that defines some parameters. It contains all information that MPI enalbed pods require for proper functioning; a script with a mpirun[4] command that each pod will execute and hostfile - file with DNS names of all other MPI pods.
2. RBAC – controller configures resources-based access so that pods have all necessary privileges.
3. Workers – conroller creates and waits for the successful creation of worker pods. After they are created, they go into infinite sleep, until launcher will start workloads on them.
4. Launcher – controller creates launcher which reads from the config created in step 1 and starts workload on all worker containers.
5. End – successful finish of launch container will indicate controller that it can delete all resources.

[4]mpirun command allows to execute parallel openmpi jobs

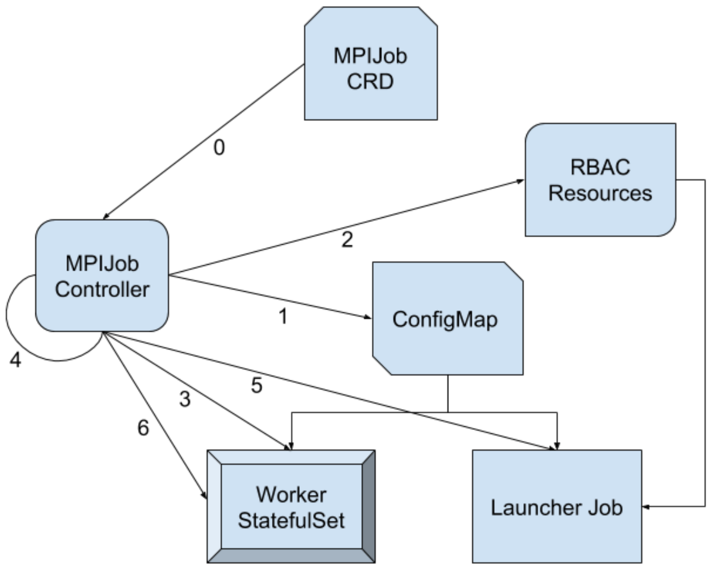


FIGURE 2.4: MPI operator – source *MPI Operator*

Chapter 3

Related Works

The cluster architecture designed in this thesis focuses on Kubernetes. We will go over two other solutions which have their market share and a different approach to cluster management.

Rocks (*Rocks Cluster Distribution*) at some point in time was a best practice solution for clusters. The popularity peaked in the 2000s. We will look into what features this tool has.

Then we will explore Apache Mesos (*Apache Mesos Architecture*) as it has a big chunk of the market in today's world. This solution has some exciting features and can compete in some places with Kubernetes.

3.1 Rocks Cluster Distribution

Rocks Cluster Distribution is one of the most popular Linux distribution for HPC clusters. Initially, it was running on top of Red Hat Linux. Modern cluster is based on CentOS. Rocks include many tools straight from the box, such as MPI and batch-queuing systems such as Sun Grid Engine (SGE), which users can utilize straight after installing the cluster.

Rocks can extend their features with such called rolls (Roll CDS). For example, this feature easily configures Lustra roll or Java roll, saving a lot of time and effort. Rolls and many other features made it widespread in the world of academia and government. According to *Rocks Cluster Distribution* in 2010, it was employed in 1,376 clusters.

Rocks did a great job in the 2000s solving problems for the academic world. In today's world, there are new approaches and designs that rocks do not support.

Here are a few drawbacks of the system:

- Bad container support – it is hard to imagine a sophisticated product without containerization. A more modern solution like Mesos or Kubernetes already has advanced container orchestration.
- Community – the popularity of the rocks has decade over the years. Update of the distribution is irregular, and its community is shrinking.

Architecture overview

The architecture of the cluster consists of four essential components – Fig.3.1:

- Worker Nodes – nodes that run the actual workload. They are all interconnected over a speed local area network that allows high throughput and low latency. Users can configure their Node types, such as a storage node with Lustre configured on it.
- Frontend Node – a node that serves as an entry point for workload configuration and gives access to cluster observability.
- Compute network – high-speed network usually utilizes infinite band Network interfaces. This network is primary used for workloads.
- Communication network – this is a regular network which we use for nodes communication or management. If the high-speed network fails, the communication network works as a safety measurement.

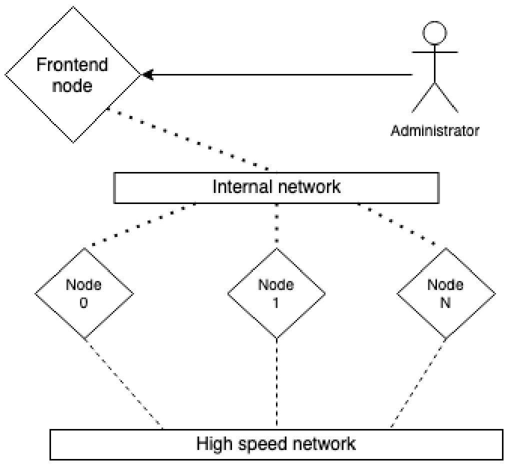


FIGURE 3.1: Rocks cluster architecture

3.2 Apache Mesos

Apache Mesos is an open-source cluster manager (*Apache Mesos Architecture*) that handles workloads in a distributed environment. It was initiated as a UC Berkeley project. The big difference in opposite to the k8s is that Mesos supports many workload types, not only containers.

Apache Mesos is similar to Kubernetes in many ways. It allows the management of container workloads across all nodes with out-of-the-box DNS resolution.

Architecture overview

The architecture of Mesos consists of many parts and requires some time to get a fluent understanding of it *Apache Mesos Architecture*. Figure 3.2 gives an overview of the components.

Core components of Apache Mesos:

- Framework – for a specific workload type, a framework is deployed. In case mpi workload is needed – the mpi framework should be present in the cluster. The framework itself has two main parts, scheduler, and executor. A scheduler is responsible for properly scheduling and assigning tasks to the nodes.

An executor is running on those nodes and processes the tasks given by the scheduler.

- Agent Daemons – each node runs a different agent daemon. It manages the node state, framework state.
- Master Daemon – the master daemon manages all agent daemons.

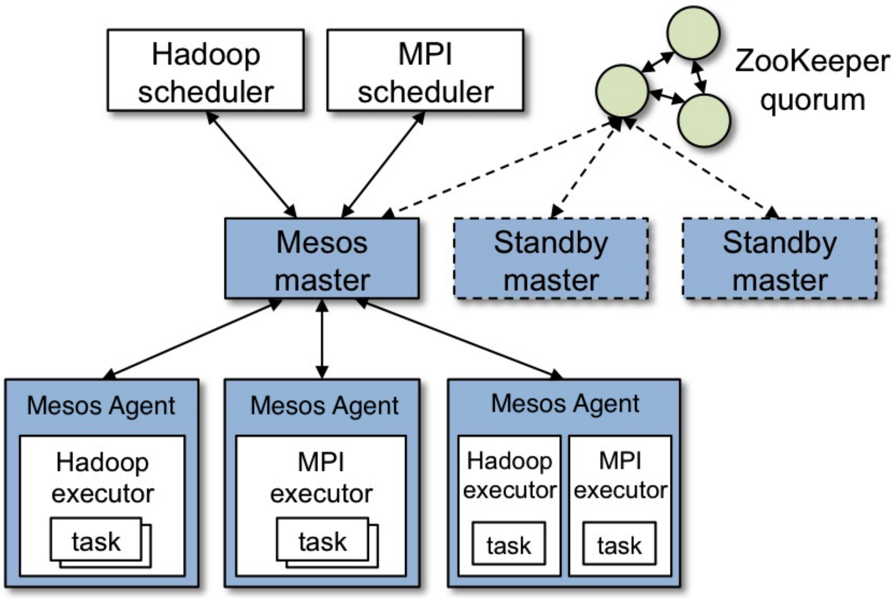


FIGURE 3.2: Mesos architecture

Execution process

The execution process is the same for any workload. Workloads will only differ in the implementation of executor and scheduler.

Execution process of the cluster looks as follows:

1. Agent sends information about how many resources they have available on each node. The master agent then goes through defined policies and informs specific frameworks how many resources they can utilize.
2. Each framework's scheduler decides which workloads should be deployed on which node and how much resources they will consume.
3. Master daemon collects data from frameworks. After that is sends information about tasks to all agent daemons. Agents call the specific executor on their nodes with supplied information, and the computing begins.

As a summary, Apache Mesos is a popular modern solution for many clusters. It is worth giving this technology a look when deciding how to configure a cluster.

Chapter 4

Solution Overview

4.1 General architecture

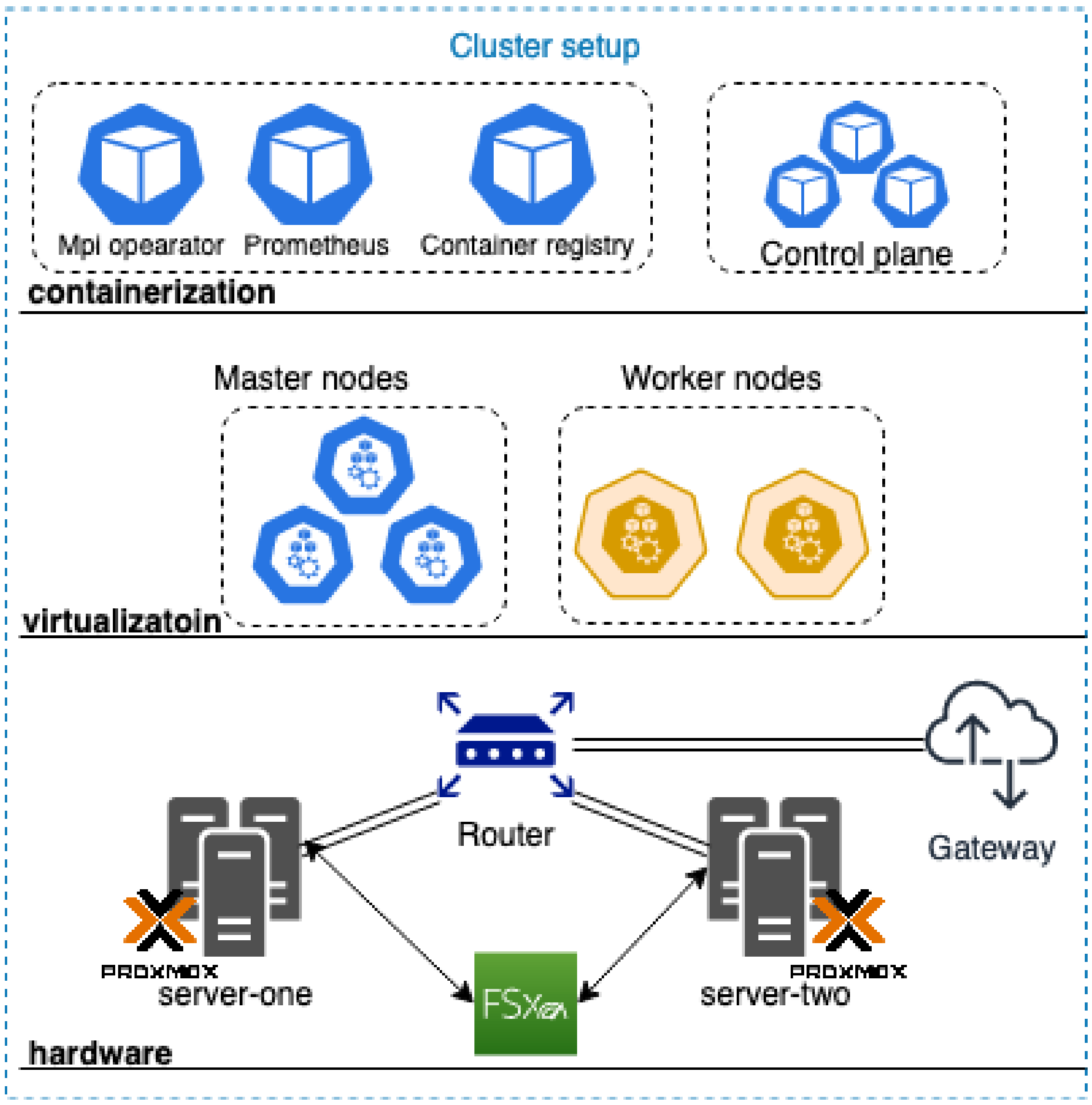


FIGURE 4.1: Cluster architecture

We split the cluster into three abstract parts:

- Hardware – configuration of physical nodes, power supply, network equipment. Configuration of automatic node connection into the cluster.
- Virtualization – configuration of virtualization managing tool. Setup of a virtual machine for Kubernetes master and workers nodes.
- Containerization – configuration of container-based resources. Control plane components, mpi operator, observability tools (Prometheus and grafana), and container registry.

4.2 Hardware

With help from the Institute for Condensed Matter Physics of the National Academy of Sciences of Ukraine, we were able to get two cluster nodes (Figure 4.2) for our experiments and architecture configuration.

	Node one	Node two
RAM	135Gb	8Gb
CPU Cores	Gb	8Gb
Processor	AMD Opteron(tm) 6128	Intel(R) Xeon(R) CPU E3-1230 V2
Storage	2Tb	0.5Tb

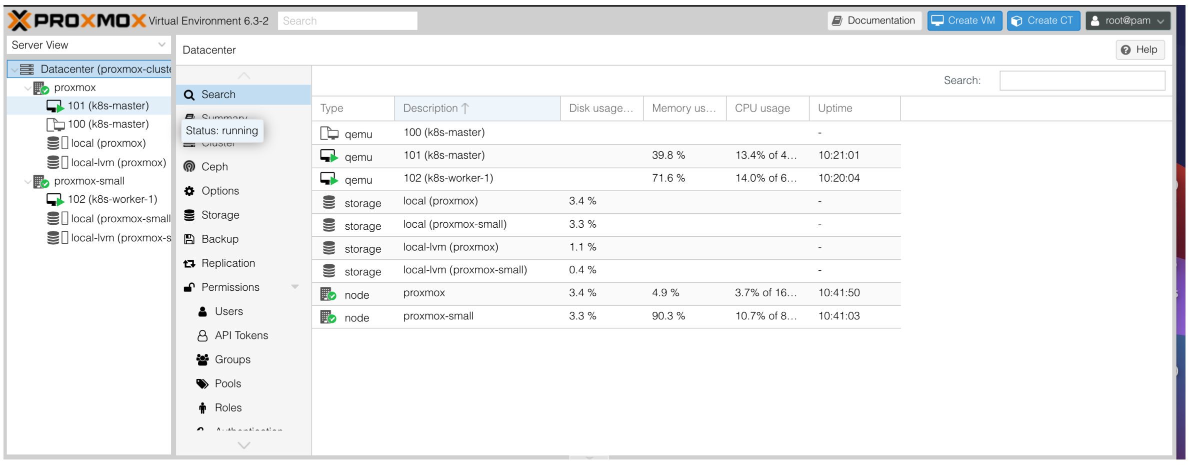
TABLE 4.1: Hardware specs

FIGURE 4.2: Bare metal cluster

4.3 Virtualization

4.3.1 Proxmox

Proxmox Virtual Environment (*Proxmox website*) is an open-source server virtualization management platform. It can manage containers and virtual machines. Proxmox stores all VM templates locally. The ability to quickly create configured nodes allows us to scale a cluster when it has insufficient resources. Proxmox also allows us to observe cluster states and get alerts when something is going wrong.
We configure the ISO image of proxmox on our two physical nodes. After we fully configure proxmox nodes, we can access GUI (Figure 4.3) through the exposed port.



Type	Description ↑	Disk usage…	Memory us…	CPU usage	Uptime
qemu	100 (k8s-master)				-
qemu	101 (k8s-master)		39.8 %	13.4% of 4…	10:21:01
qemu	102 (k8s-worker-1)		71.6 %	14.0% of 6…	10:20:04
storage	local (proxmox)	3.4 %			-
storage	local (proxmox-small)	3.3 %			-
storage	local-lvm (proxmox)	1.1 %			-
storage	local-lvm (proxmox-small)	0.4 %			-
node	proxmox	3.4 %	4.9 %	3.7% of 16…	10:41:50
node	proxmox-small	3.3 %	90.3 %	10.7% of 8…	10:41:03

FIGURE 4.3: Proxmox cluster

4.3.2 Network

We have two Network topologies, the overlay, and underlay (*Difference between Underlay Network and Overlay Network*).
All cluster nodes and virtual machines are in 192.168.0.1/24 private underlay network. All nodes can access the public internet for an update or external communication through a default gateway. Inbound traffic by default is fully closed, and only particular ports are forwarded through the router. For example, we expose port 8006 for proxmox and port 6443 for k8s API-server.
We configure one of the proxmox nodes to have a static IP. Any new node uses this IP to join into proxmox cluster. After we join a new node into a cluster, we will see it in proxmox GUI (Figure 4.3). When we create a virtual machine in proxmox, it will get a dynamic IP from the same CIDR block. See underlay network architecture – Figure 4.4.
Kubernetes has its overlay *Cluster Networking* over which all k8s components communicate. This overlay network has many implementations, and their overview is out of scope.

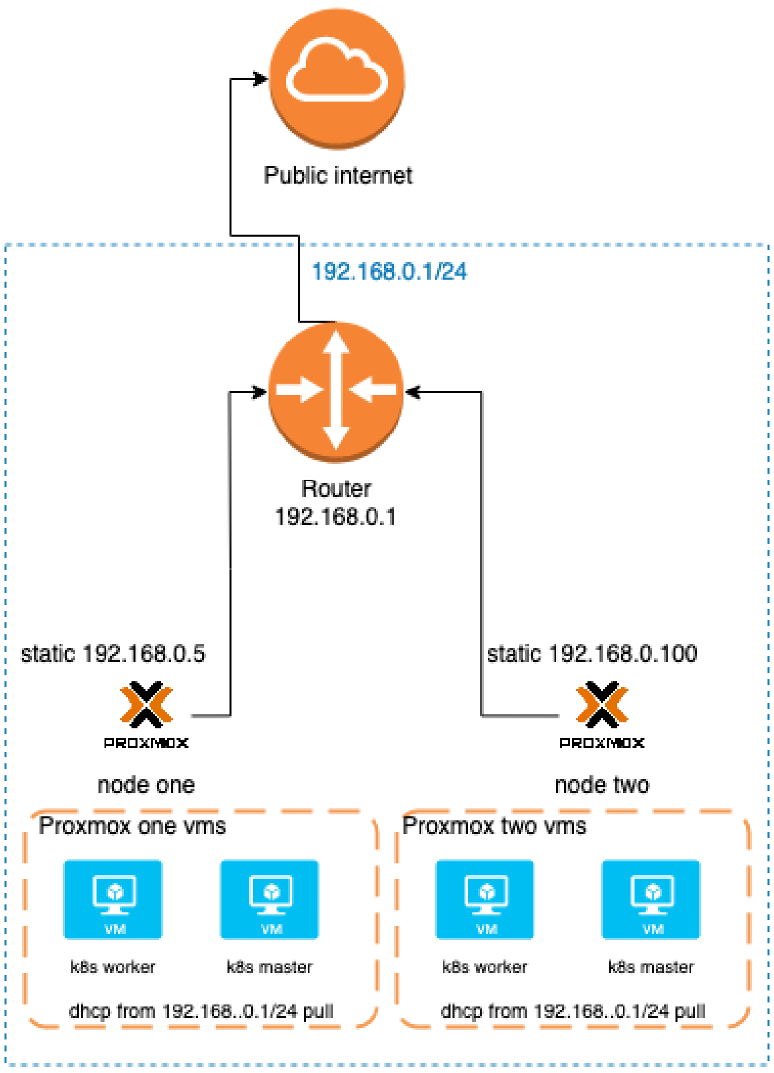


FIGURE 4.4: Network configuration

4.3.3 Cluster scaling

The ability to quickly add new hardware and virtual nodes is crucial. The Preboot eXecution Environment, PXE, is a specification that enables pxe-clients, nodes that have support for PXE, to boot from the network.
When we add new servers into the cluster, they go directly to the pxe server to start the booting process. To add a node into a cluster, we need to connect it to the same local area network. Through the TFTP server client will receive ISO image and start the installation of proxmox distribution – Figure 4.5.
We configure ISO images to connect into the cluster at boot time automatically. When we boot the node over the network for the first time, this machine executes a script that joins the node into the cluster.

```
$ pvecm add $ipv4_static_proxmox_address --fingerprint $auth_info
```

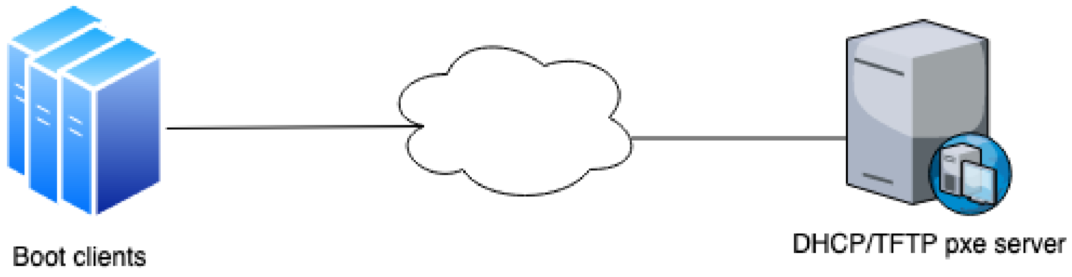


FIGURE 4.5: Pxe boot

4.4 K8S configuration

Kubernetes has two main types of nodes, masters, and workers. The only difference is in the workload types they host. Following best practice, we will configure all control plane components on master nodes and run workloads only on worker nodes. Users can still decide to run workloads on master nodes, but it can negatively impact cluster performance.

4.4.1 Node template

In proxmox, we can create VM templates. Our main template will use a base image ubuntu 20.04. Some of the important packages are:

- Docker – Each VM must have a container engine available so that k8s can create containers on our nodes.
- Kubeadm – Tool which simplifies the k8s installation process.
- kubectl – CLI utility for communication with k8s API-server.
  We will enable DHCP for all nodes as default. The master node will overwrite this configuration with a static IP so that worker nodes can communicate with API-server. We will generate a template only on one proxmox node, and it will be available for the cluster – Figure 4.3.

4.4.2 Installation

After all necessary tools are installed we can utilize kubeadm utility to install cluster. Cluster initialization commands:

- Initialize cluster by installing control plane on master nodes. Configured cluster on Figure 4.6

  ```
  $ kubeadm init --args
  ```

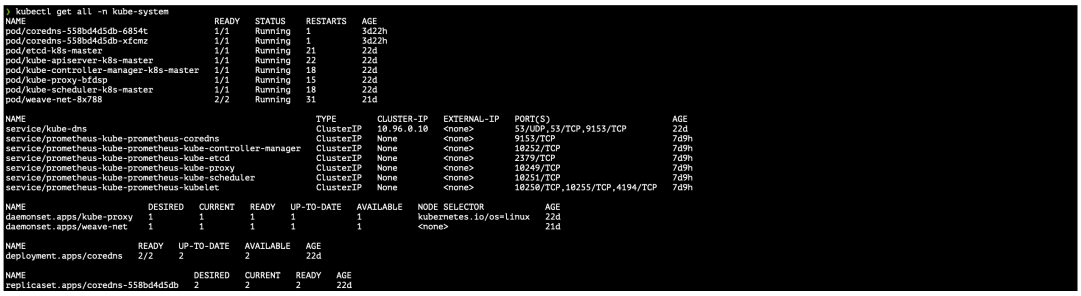
- Configure network plugin. We will install one of k8s *Cluster Networking* components called weave net [1].

  ```
  $ kubectl apply -f "https://cloud.weave.works/k8s/net
  ?k8s-version=$(kubectl␣version␣|␣base64␣|␣tr␣-d␣'\n')"
  ```

- From all worker nodes we join k8s cluster. We supply static IP of our master node, and an authentication token.

  ```
  $ kubeadm join <control-plane-host>:<control-plane-port>
   --token <token> --discovery-token-ca-cert-hash sha256:<hash>
  ```

[1] Weave net is a netowork toolkit that connects containers into a virtual network.



```
❯ kubectl get all -n kube-system
NAME                                     READY   STATUS    RESTARTS   AGE
pod/coredns-558bd4d5db-6854t             1/1     Running   1          3d22h
pod/coredns-558bd4d5db-xfcmz             1/1     Running   1          3d22h
pod/etcd-k8s-master                      1/1     Running   21         22d
pod/kube-apiserver-k8s-master            1/1     Running   22         22d
pod/kube-controller-manager-k8s-master   1/1     Running   18         22d
pod/kube-proxy-bfdsp                     1/1     Running   15         22d
pod/kube-scheduler-k8s-master            1/1     Running   18         22d
pod/weave-net-8x788                      2/2     Running   31         21d

NAME                                                  TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                        AGE
service/kube-dns                                      ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP         22d
service/prometheus-kube-prometheus-coredns            ClusterIP   None         <none>        9153/TCP                       7d9h
service/prometheus-kube-prometheus-kube-controller-manager   ClusterIP   None   <none>        10252/TCP                      7d9h
service/prometheus-kube-prometheus-kube-etcd          ClusterIP   None         <none>        2379/TCP                       7d9h
service/prometheus-kube-prometheus-kube-proxy         ClusterIP   None         <none>        10249/TCP                      7d9h
service/prometheus-kube-prometheus-kube-scheduler     ClusterIP   None         <none>        10251/TCP                      7d9h
service/prometheus-kube-prometheus-kubelet            ClusterIP   None         <none>        10250/TCP,10255/TCP,4194/TCP   7d9h

NAME                        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/kube-proxy   1         1         1       1            1           kubernetes.io/os=linux   22d
daemonset.apps/weave-net    1         1         1       1            1           <none>                   21d

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/coredns   2/2     2            2           22d

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/coredns-558bd4d5db   2         2         2       22d
```

FIGURE 4.6: Control plane

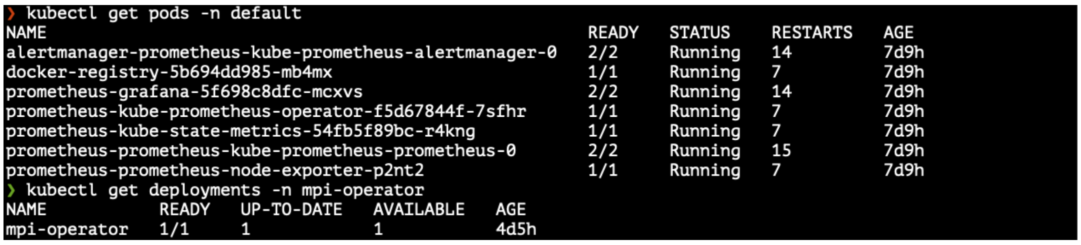
After successful control plane configuration, we need to deploy some extra components. We will deploy Prometheus[2] with Grafana[3] for observability and local docker registry, so that we can quickly pull and push images. Deployed components – Figure 4.7.

4.5 MPI opearator

To enable mpi workloads we need to deploy mpi-operator. Kubeflow has an open source solution for that - *MPI Operator*. mpi-operator.yaml describes which resources to deploy for operator.

```
$ git clone https://github.com/kubeflow/mpi-operator

$ kubectl create -f mpi-operatordeploy/v1alpha2/mpi-operator.yaml
```



```
❯ kubectl get pods -n default
NAME                                                      READY   STATUS    RESTARTS   AGE
alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running   14         7d9h
docker-registry-5b694dd985-mb4mx                          1/1     Running   7          7d9h
prometheus-grafana-5f698c8dfc-mcxvs                       2/2     Running   14         7d9h
prometheus-kube-prometheus-operator-f5d67844f-7sfhr       1/1     Running   7          7d9h
prometheus-kube-state-metrics-54fb5f89bc-r4kng            1/1     Running   7          7d9h
prometheus-prometheus-kube-prometheus-prometheus-0        2/2     Running   15         7d9h
prometheus-prometheus-node-exporter-p2nt2                 1/1     Running   7          7d9h
❯ kubectl get deployments -n mpi-operator
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
mpi-operator   1/1     1            1           4d5h
```

FIGURE 4.7: Custom k8s components

[2]Prometheus is a free software tool for monitoring. This tool records real-time metrics in a time series database.

[3]Grafana is a software that visualizes metrics to give observability of the system, it requires source of metrics, Prometheus is a popular choice.

Chapter 5

Running workloads

The final stage is to run mpi workloads in the configured cluster and observe successful results. This chapter aims to show how simple and intuitive it is to use our cluster solution and MPI operator in particular.

5.1 Configure workload container

In our MPIJob specification (Table.5.2), we need to define a container image, and this image must contain MPI-compatible code. For this workload, we are going to use an example code (Table.5.1) from *Itro to mpi*.

```
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
    int ierr, num_procs, my_id;

    ierr = MPI_Init(&argc, &argv);

    /* find out MY process ID, and how many processes were started. */

    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    printf("Hello␣world!␣I'm␣process␣%i␣out␣of␣%i␣processes\n",
       my_id, num_procs);

    ierr = MPI_Finalize();
}
```

We wrap this code into a container and push it into our private registry. To create the image, we will use a Dockerfile in which we define build specification – Listing.5.1. We will use this container in our MPIJob specification.

5.2 Configure MPI task

We describe workload in a YAML file; it contains information about how the workload should behave (Table 5.1). A few important fields:

- Kind - specify which resource type to deploy. Mpi controller filters by MPIJob kind to understand that it should parse this YAML specification.

- Launcher - configuration for the launcher container.
- Worker - configuration for worker containers.

We are going to send this YAML specification to the API-server, and it will create all of the necessary resources.

```
$ kubectl apply -f mpijob-workload.yaml
```

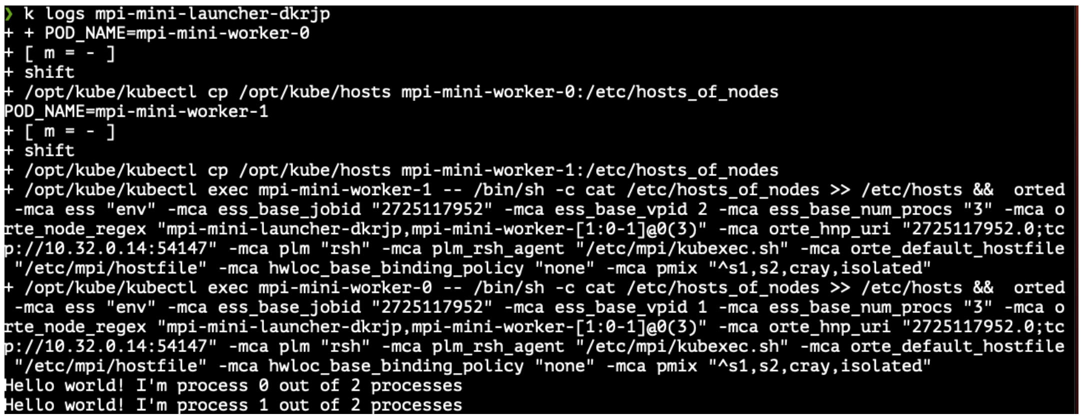
```
apiVersion: kubeflow.org/v1alpha2
kind: MPIJob
metadata:
   name: mpi-mini
spec:
    slotsPerWorker: 1
    cleanPodPolicy: Running
    mpiReplicaSpecs:
      Launcher:
        replicas: 1
        template:
           spec:
             containers:
             - image: \verb|$image_name|
               name: \verb|$container_name|
               command:
               - mpirun
               - --allow-run-as-root
               - -np
               - "2"
               - -bind-to
               - none
               - ./main
      Worker:
        replicas: 2
        template:
          spec:
            containers:
            - image: \verb|$image_name|
              name:  \verb|$container_name|
```

TABLE 5.1: MPIJob specification

5.3 Observe mpi results

Check workload results. Mpi-opearator collects and exposes all output in launcher. Kuberenetes has native way of scraping all logs from stdout. Kubectl utility allows us to check results - Fig.5.1:

```
$ kubectl logs <mpi-launcher-name>
```



```
❯ k logs mpi-mini-launcher-dkrjp
+ + POD_NAME=mpi-mini-worker-0
+ [ m = - ]
+ shift
+ /opt/kube/kubectl cp /opt/kube/hosts mpi-mini-worker-0:/etc/hosts_of_nodes
POD_NAME=mpi-mini-worker-1
+ [ m = - ]
+ shift
+ /opt/kube/kubectl cp /opt/kube/hosts mpi-mini-worker-1:/etc/hosts_of_nodes
+ /opt/kube/kubectl exec mpi-mini-worker-1 -- /bin/sh -c cat /etc/hosts_of_nodes >> /etc/hosts &&  orted
 -mca ess "env" -mca ess_base_jobid "2725117952" -mca ess_base_vpid 2 -mca ess_base_num_procs "3" -mca o
rte_node_regex "mpi-mini-launcher-dkrjp,mpi-mini-worker-[1:0-1]@0(3)" -mca orte_hnp_uri "2725117952.0;tc
p://10.32.0.14:54147" -mca plm "rsh" -mca plm_rsh_agent "/etc/mpi/kubexec.sh" -mca orte_default_hostfile
 "/etc/mpi/hostfile" -mca hwloc_base_binding_policy "none" -mca pmix "^s1,s2,cray,isolated"
+ /opt/kube/kubectl exec mpi-mini-worker-0 -- /bin/sh -c cat /etc/hosts_of_nodes >> /etc/hosts &&  orted
 -mca ess "env" -mca ess_base_jobid "2725117952" -mca ess_base_vpid 1 -mca ess_base_num_procs "3" -mca o
rte_node_regex "mpi-mini-launcher-dkrjp,mpi-mini-worker-[1:0-1]@0(3)" -mca orte_hnp_uri "2725117952.0;tc
p://10.32.0.14:54147" -mca plm "rsh" -mca plm_rsh_agent "/etc/mpi/kubexec.sh" -mca orte_default_hostfile
 "/etc/mpi/hostfile" -mca hwloc_base_binding_policy "none" -mca pmix "^s1,s2,cray,isolated"
Hello world! I'm process 0 out of 2 processes
Hello world! I'm process 1 out of 2 processes
```

FIGURE 5.1: Mpi-job results

```
FROM debian

RUN apt-get update && apt-get install -y \
    cmake \
    libgtest-dev \
    libboost-test-dev \
    openmpi-bin \
    openmpi-doc \
    libopenmpi-dev \
    libboost-all-dev \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

COPY ./ /data

WORKDIR /data

RUN mpicc -o main ./main.c

CMD ["/bin/bash"]
```

LISTING 5.1: Dockerfile for mpi workloads

Chapter 6

Summary

As part of this thesis, we designed a software architecture for HPC clusters. We tested our solution on top of bare-metal clusters. We showed both the technical and theoretical path that it takes to build an HPC cluster.
At the beginning of this thesis, we defined some objectives:

- Fault tolerance – this solution is fault-tolerant, and it does not have a single point of failure. If some system part breaks, it will not impact the overall functionality.
- Observability – Prometheus, in combination with Grafana, gives us visibility of the system. With this solution, we can monitor our cluster nodes and workloads.
- Scalability – a PXE based solution is configured to allow seamless scaling of the hardware nodes. We also configured the ability to scale the k8s cluster with new virtual machines.
- Maintenance – as our core technology, we really on k8s. Kubernetes requires maintains and expertise in it for proper functioning, but supporting k8s is far easier than most solutions on the market.

The future destination of this architecture is unknown, but we hope that it will be running in Ukrainian clusters shortly. Well, it is already running in one apartment!

6.1 Future works

Our solution has some places for improvement. Whenever we start new MPI workloads, we rely on the k8s default scheduler to schedule them. The behavior of this scheduler is not always the one we expect. We plan to write our scheduler, which is going to have different task queues depending on our needs. One more critical improvement will be to enable CUDA virtualization as many modoern HPC clusters require GPU support.

Bibliography

apache. *Apache Mesos Architecture*. URL: https://mesos.apache.org/documentation/latest/architecture/.

Bhardwaj, Rashmi. *Difference between Underlay Network and Overlay Network*. URL: https://ipwithease.com/difference-between-underlay-network-and-overlay-network/.

Council, National Research (2005). *Getting Up to Speed: The Future of Supercomputing*. National Academies Press. ISBN: 0309095026.

edu, Condor. *Itro to mpi*. URL: http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml.

Foundation, Cloud Native Computing. *Kubernetes*. URL: https://kubernetes.io.

Hykes, Solomon. *Docker*. URL: https://www.docker.com/.

IBM. *What is a mainframe*. URL: https://www.ibm.com/it-infrastructure/z/education/what-is-a-mainframe.

Kernel.org. *Cgroups*. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

Kubeflow. *MPI Operator*. URL: https://github.com/kubeflow/mpi-operator.

Kubernetes. *Cluster Networking*. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/.

Luksa, Marko (2018). *Kubernetes in action*. Manning Publications. ISBN: 9781617293726.

manual, Linux. *Chroot*. URL: https://man7.org/linux/man-pages/man2/chroot.2.html.

– *Namespaces*. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html.

Osna, Rani. *A Brief History of Containers: From the 1970s Till Now*. URL: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016.

Solutions, Proxmox Server. *Proxmox website*. URL: https://www.proxmox.com/en/.

TOP500. *TOP 500 list*. URL: https://www.top500.org/.

wikivisually. *Rocks Cluster Distribution*. URL: https://wikivisually.com/wiki/Rocks_Cluster_Distribution.