UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Emil Koemets

Kubernetes cluster automated setup and management with Ansible in the University of Tartu High Performance Computing center

Bachelor's Thesis (9 ECTS)

Supervisors: Ivar Koppel (PhD)
Sander Kuusemets (BsC)

Tartu 2022

Kubernetes cluster automated setup and management with Ansible in the University of Tartu High Performance Computing center

Abstract:

Kubernetes is one of the most popular container orchestration solutions. It eases the process of deploying and managing containerized workloads. University of Tartu High Performance Computing center uses Kubernetes to provide the university's research groups with the capabilities required for running containerized workloads. The result of this thesis is an automation solution using the Ansible automation tool, which allows the cluster operators to set up and manage the Kubernetes cluster with minimal effort.

Keywords:

Kubernetes, Ansible, automation

CERCS:

P170 Computer science, numerical analysis, systems, control

Kuberenetese klastri automatiseeritud ülesseadmine ning haldus Ansible abil Tartu Ülikooli teadusarvutuste keskuse näitel

Lühikokkuvõte:

Kubernetes on üks populaarsemaid konteinerite orkestreerimise lahendusi, mis lihtsustab konteinerite kasutamist. Tartu Ülikooli teadusarvutuste keskus rakendab Kubernetest ülikooli uurimisgruppidele konteinerite kasutamiseks vajaliku võimekuse pakkumiseks. Käesoleva töö tulemusena on Ansible automatiseerimise tööriista abil valminud lahendus, mis võimaldab minimaalse vaevaga vajaliku Kubernetese klastri ülesseadmist ning haldust.

Võtmesõnad:

Kubernetes, Ansible, automatiseerimine

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of contents

1 Introduction

Containers provide a reliable way to package and deploy applications in different environments. This is achieved by packaging all of the system libraries and dependencies alongside the application and running it inside an isolated environment, which ensures the same conditions for the application, whether it is run on a developer's machine or in a complex cloud environment. Containers have small system resource usage and fast startup time, which promotes creating a separate one for each application. This often leads to environments running hundreds or even thousands of containers that each require setup and management. Container orchestration tools are often used to solve these problems. They provide a wide variety of built-in functionality to manage container deployments successfully. Kubernetes is one of the most popular container orchestration solutions. It allows running containers on a cluster of machines by providing a declarative language for configuring the state of containers and related components.

University of Tartu HPC (High Performance Computing) center offers infrastructure and services for scientific computing to the university's research groups. One of the services they provide is a managed Kubernetes cluster service. The client is provided with access to the cluster that they can utilize to run containerized workloads. Providing this service requires setup, configuration and management of the Kubernetes cluster and its applications. These actions are currently carried out manually, which not only requires valuable system administrators' time but also makes it difficult to replicate the cluster for testing purposes. This thesis aims to provide an automated setup and management solution using the Ansible automation tool for the cluster control plane, worker nodes and networking.

This thesis is organized as follows: Chapter 2 gives an overview of containers and container orchestration solutions. Chapter 3 describes Kubernetes architecture and operating principles. Chapters 4-7 describe the automated tasks for setting up and managing the cluster in the following order: preparing the machines, setting up nodes for running Kubernetes components, creating a Kubernetes cluster from individual machines and setting up cluster networking. Chapter 8 describes the cluster management process using the created automation solution. Chapter 9 briefly summarizes the created automation solution.

2 Containers

A container is a standard software unit that contains application code and includes all required dependencies and libraries for running the application [1]. Containers have been gaining popularity with the shift towards cloud computing. According to IBM's survey [2] conducted in 2020, over 56% of participants were already highly familiar with containers. Containers allow the creation of packages known as container images that can be used in a wide variety of environments as all of the application dependencies are self-contained. Containers also ease the process of running the application. Using containers the application lifecycle is managed by starting or stopping the container. The exact command to actually run the application is included inside the container image.

Containers are often compared to virtual machines as both provide an isolated environment for running applications. One of the main differences is the virtualization level at which these solutions operate. Virtual machines virtualize the hardware while containers virtualize the operating system using the features provided by the host operating system. This results in the containers having a smaller system resource usage and faster startup times than virtual machines [3].

2.1 Container architecture

Containers are created by providing a suitable container image to a container runtime [1]. A container image is a lightweight immutable executable containing application code alongside its dependencies and libraries required to run the application [4]. The container image does not include the kernel as containers share the host kernel. Container images are also layered, meaning that one image can be used as the base to create a new image with extended functionality [4]. Such layered architecture allows the reuse of container images and reduces the complexity of building new container images. Depending on the application's language, there are often container images available from public registries with a preconfigured environment that can be used as a base for creating the image for the application.

Container runtime is a program responsible for creating the actual container from the container image. To ensure the compatibility of a container image with different container runtimes, OCI

(Open Container Initiative) open governance structure was founded in 2015 by the container industry leaders to create standards for container images and runtimes [5]. OCI maintains the following specifications: container image specification named Image Specification (image-spec) and runtime specification called Runtime Specification (runtime-spec) [5]. These full specification names are rarely used in practice, and container images and runtimes adhering to the specification are called OCI container images and OCI runtimes accordingly. These specifications guarantee that an OCI container image will work on any compliant runtime.

2.2 Container orchestration

The lightweight nature and fast startup time of containers promote creating a container for each application, which leads to environments running hundreds of containers. Running each container in production environments requires setup, management and monitoring, which can become quite complicated and repetitive, especially when done at scale. As an example, some of the problems that arise using containers are starting and stopping, setting up networking and providing persistent storage. These problems are commonly solved using container orchestration solutions. According to IBM's 2020 survey, 61% of the participants used a container orchestration solution for managing containers [2]. Container orchestration solutions provide an abstraction over the physical resources which the containers actually use. They allow the grouping of independent machines to form a cluster on which the containers are run. This approach also scales well as machines can be added or removed from the cluster depending on the resources actually required. The containers and additional components that the container orchestration solution manages are usually configured in a declarative way. Instead of providing the exact steps of how the container should be managed, it only requires specifying the desired state and the container orchestration solution performs the required actions to ensure the container is in the wanted state. Some of the many tasks that container orchestration solutions help automate are deployment, configuration, resource allocation, scaling, monitoring and load balancing [6].

3 Kubernetes

Kubernetes is one of the most popular open-source container orchestration solutions developed and maintained by CNCF (Cloud Native Computing Foundation). Google initially started the development on Kubernetes, but after reaching the first stable version, the project was donated to the CNCF in 2015 [7]. According to a report published in 2020 by Datadog cloud monitoring platform, based on their customer's data, approximately half of the organizations used Kubernetes to run containers [8]. Another indication of its popularity can be seen by the adoption of Kubernetes by cloud service providers. Most of the major cloud service providers like Google, Amazon and Azure have an offering for managed Kubernetes service.

3.1 Kubernetes architecture

Kubernetes cluster can be divided into two main parts: the control plane and worker nodes. The independent machines that form the Kubernetes cluster are referred to as nodes, which are represented in the state as Node objects. In this thesis, the Author uses the uncapitalized form of the object's name for generally describing the concept and capitalized form for specifically referring to the object. This style is also suggested by the Kuberenetes documentation style guide [9]. Figure 1 provides a visual overview of the Kubernetes cluster architecture. Control plane is the generic term for describing the following set of components: API (Application Programming Interface) server, controllers, scheduler and *etcd* datastore. API is an interface allowing programmatic interaction with the application providing the interface. The control plane components allow configuring the cluster using the API server and are responsible for ensuring the desired state in the cluster. A node running such a set of components is called a control plane node. The actual architecture of the control plane can vary depending on the requirements for the cluster. Often multiple control plane nodes are run simultaneously allowing the control plane to remain functional in case some of the nodes fail. By best practice, the control plane nodes only run containerized workloads that are required to run the control plane. All of the other workloads are usually run by the worker nodes. The workload that the control plane schedules to run on a worker node is a pod, which is the smallest managed deployable unit in Kubernetes that groups one or more containers that share storage and networking resources [10]. The following sections list and describe the default implementations of the control plane and worker node components.

The control plane component cloud-controller-manager is not described since the exact functionality depends on the specific cloud provider.

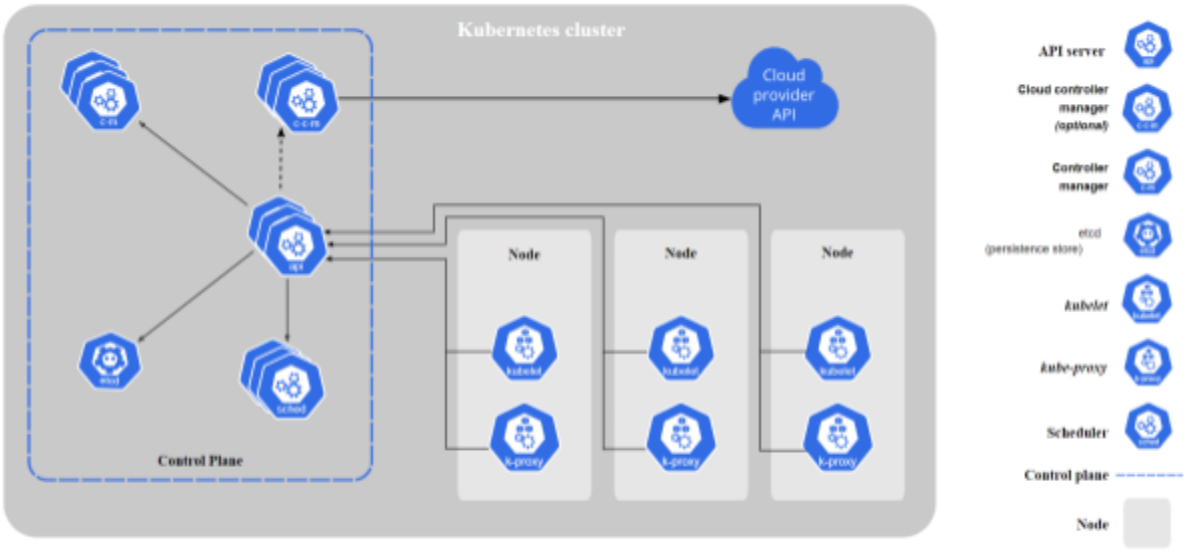

Figure 1. Kubernetes cluster architecture [11]

Kubernetes control plane is made up of the following components:

- **kube-apiserver** - Component that exposes HTTP (Hypertext Transfer Protocol) API, which provides communication with the cluster for users and internal components by querying and modifying the cluster objects [12]. HTTP is a communication protocol allowing to send data over the network in a standardized way. The main data format used to represent the API resources is JSON (Javascript Object Notation), which is a human-readable data format often used for exchanging data using HTTP. The objects are persisted using the *etcd* key-value datastore. In production environments, the communications with the API are often secured using TLS (Transport Layer Security). TLS is a cryptographic protocol allowing secure network communications between two applications.
- **kube-scheduler** - Component responsible for watching newly created Pod objects and determining the most suitable node for the pod, by taking into account the available resources for each node and the constraints defined for the Pod [13]. It is important to

note that the scheduler is only responsible for assigning a node to a pod and does not enforce it to run on that node. Kubernetes provides the following ways to configure constraints for the selected node: affinity and taints and tolerations. Affinity is used to target specific nodes on, which the pod should be scheduled. This is usually done by assigning labels to nodes and explicitly targeting those in the Pod objects specification. One common use case for it is to target nodes with a specific operating system. Taints and tolerations are used to exclude pods from being scheduled on certain nodes. A taint can be applied to the Node object and Pods without corresponding toleration are excluded from being scheduled on that node. On the other hand, Pod objects with tolerations can be still scheduled on nodes without corresponding taints. A widespread use case for this is to exclude pods from running on control plane nodes.

- **kube-controller-manager** - Set of built-in controllers bundled into a single binary for easier deployment. Each controller is responsible for tracking and enforcing the desired state of at least one Kubernetes resource as the state of the cluster is constantly changing [14]. Controllers read and modify the resources through the *kube-apiserver* instances.
- **etcd** - Strongly consistent and distributed key-value store allowing to access stored data from a cluster of machines [15]. All Kubernetes objects are persisted to *etcd* and accessed through the API server. At the time of writing, April 2022, *etcd* was the only supported Kubernetes key-value store.

Kubernetes worker nodes are made up of the following components:

- **kubelet -** Agent that runs on each node in the cluster and ensures containers specified by the control plane are running and healthy [16]. Kubelet exposes HTTP API that listens to requests which contain the Pod object's specification and fulfills the specification by managing containers using the container runtime.
- **kube-proxy -** The component responsible for providing routes to services defined in the cluster by Service objects. It is done by constantly monitoring changes made to the Service and updating the networking rules accordingly. In Kubernetes a service allows exposing a set of pods as a single network service. A more thorough explanation for services is provided in the networking section.

- **Container runtime -** Container runtime is the program responsible for creating the container from the container image. Any container runtime can be chosen for the node as long as it implements the Kubernetes CRI (Container Runtime Interface). This interface allows *kubelet* to communicate with the container runtime without coupling it to any specific runtime implementation.

3.2 Kubernetes objects

Kubernetes objects are persistent entities that describe the state of the cluster [17]. Each object contains the desired state and current state that are represented by fields *spec* and *status* accordingly. The desired state is configured by submitting a Kubernetes object with the *spec* field to the Kubernetes API. In addition to the *spec* field, the following fields have to be provided for an object:

- **apiVersion** - Defines the API version used for specifying the *spec* field. The schema and fields supported by the Kubernetes API for objects may differ depending on the version.
- **kind -** Specifies the object's type.
- **name** - Part of the object's metadata that defines its name.

Kubernetes object that only defines the required fields and the specification is also known as manifest. Once the object is created through Kubernetes API, controllers will continuously ensure that the object's current state matches the desired one until the object is removed, which is done by continuously monitoring the current state of the object and performing required actions. After performing any actions, the controllers update the *status* field to reflect the actual state of the object.

In Kubernetes, it is also common for controllers to create and manage other objects to provide the desired state. This is the case for workload objects for which controllers usually create and manage a set of Pod objects. The following workload objects are more commonly used to run containers:

- Deployment - Runs a set number of pods and also provides features for scaling and rolling updates.
- ReplicaSet - Runs a set number of pods at all times, but does not provide any additional features.

- DaemonSet - Runs a pod on every node in the cluster.

Kubernetes objects are queried and modified using the API server. While it is possible to interact with the API server directly, it is more common for users and cluster operators to use the *kubectl*[1] command-line utility [12]. It provides many easy-to-use commands for interacting with objects. Kubectl allows objects to be represented in either JSON or YAML (Yet Another Markup Language). YAML is a data language often used for defining configuration files. Figure 2 shows a simple Pod manifest written in YAML that uses the Ubuntu container image. The YAML representation is often preferred for creating manifests as it is considered to be more readable and allows comments. It is to be noted that the YAML objects are still converted to JSON by *kubectl* before they are sent to the API server.

```
1
2   apiVersion: v1
3   kind: Pod
4   metadata:
5     name: ubuntu-pod
6   spec:
7     containers:
8     - name: ubuntu-container
9       image: ubuntu:latest
10
11
```

Figure 2. Ubuntu container Pod manifest

Kubectl allows the cluster objects to be managed either imperatively or declaratively. Imperative management requires to always specify the operation that is carried out with the object. This is often used in situations that require one-off modifications to the cluster state. Declarative management requires specifying the desired state using manifests and *kubectl* will automatically perform required modifications to satisfy the desired state, if possible. This allows representing the cluster's desired state using a set of manifest files, which can be easily changed and version controlled.

[1] https://kubectl.docs.kubernetes.io/

3.3 Kubernetes networking

Kubernetes does not provide a networking implementation but rather only specifies a networking model that has to be fulfilled. The actual networking inside the cluster is provided by a network plugin that has to satisfy the following requirements in the cluster:

- IP address is assigned to every pod.
- Containers inside a pod share the same IP.
- All pods in the cluster can communicate with each other using pod IP addresses.

One of the main benefits of plugin-based networking is flexibility, allowing Kubernetes to be deployed into various environments with different networking requirements. The networking is usually provided by a plugin that implements the CNI (Container Networking Interface), which defines a standardized interface for managing container networking. In the case of Kubernetes, the networking is set up for a pod instead of a single container. The CNI plugin is installed on each node by the networking plugin and called by *kubelet* as part of the pod's management lifecycle. It is to be noted that there also exists a non-CNI based solution *kubenet*, but it is less used in production clusters due to its limited functionality.

3.3.1 Service

In a Kubernetes cluster, it is common to run an application on a set of non-permanent pods, which may be created or destroyed as needed. A problem arises when trying to access this application over the network as each pod is assigned an IP address at random. In Kuberenetes, this problem is usually solved using services. A Service object is created by defining the Pod objects to target and how these pods should be accessed. Targeted pods are usually defined by selectors, which allow selecting pods using the labels attached to them. The way to access those pods is specified by providing at least one of the following Service object types:

- ClusterIP - Exposes the service on a single IP address that is only accessible inside the cluster network.
- NodePort - Exposes the service on a static port on all nodes in the cluster, which makes it accessible inside the host network. By default, the ports that can be used are in the range 30000-32767.
- LoadBalancer - Exposes the service on an external load balancer. This is often used to make the service accessible from public networks.

The Service object only specifies how the application should be exposed and the actual work to provide the service is done by controllers. The services of ClusterIP and NodePort type are provided inside the cluster by *kube-proxy*. The services of the LoadBalancer type are not provided by default and require a separate controller, which configures the external load balancer according to the Service objects.

3.3.2 Ingress

Kubernetes cluster is often used for running multiple applications, which users can access from the public network. While services could be used to expose the applications, a more common approach is to use the Ingress object, which allows exposing HTTP routes to the services inside the cluster. An Ingress object is created by defining a set of rules that specify to which services the incoming traffic is sent. Each rule defines an optional host for which traffic is matched, a list of paths and a backend for each path. The backend specifies the service to which traffic is directed. Kubernetes does not provide a default controller for the Ingress objects, which means the installation of an ingress controller is required.

4 Automation setup

University of Tartu HPC center provides university's research groups with infrastructure and services used for scientific computing. One of the services they provide is a managed Kubernetes cluster. The client is provided access to their own Kubernetes namespace in which they can run their applications using containers. The application can also be exposed to the public network if needed using the UT HPC center reverse proxy servers. To provide this service they have to set up and manage the Kubernetes cluster to keep it up-to-date with the latest updates and security patches. These actions are currently carried out manually, which takes up valuable time as it requires manually connecting to each individual machine and performing the required actions. It also makes the cluster setup hard to replicate for testing purposes. This thesis aims to provide an automated solution for setting up and managing the following cluster components: control plane, worker nodes, networking and ingress.

One of the essential parts of automating the Kubernetes cluster setup and management is choosing an automation tool. Automation tools make the automation process easier by providing a declarative language used for defining host configuration and a way to apply the defined configuration to multiple hosts. For example, the simple task of creating a file on multiple hosts would first require manually connecting to each host. Then executing a command to determine if the file does not already exist and only then executing a command to actually create the file. Automation tools streamline this process by only requiring the user to define the desired state of the target hosts, in this case, the location of the file and automatically perform the required actions to ensure the desired state is satisfied. Another benefit these tools provide is easier collaboration since they require the automation process to be structured in a certain way specified by the tool.

4.1 Ansible

Ansible[2] was chosen as the automation tool for automating the deployment and configuration of the Kubernetes cluster. It was chosen over other automation tools mainly due to the fact that UT HPC center already uses Ansible for their existing automation processes and therefore system administrators have the required expertise for using it. Ansible uses an agentless architecture,

[2] https://www.ansible.com/

which utilizes SSH (Secure Shell) to deploy and configure software on target hosts. SSH is a network protocol often used for securely accessing hosts over the network. Target hosts are usually defined using their hostname inside an inventory file. Inventory also allows grouping multiple hosts by assigning them to a group, which can be used in playbooks to apply a similar configuration to multiple hosts. Ansible also provides two default groups: all, which includes all hosts defined in the inventory, and ungrouped, which includes all hosts not assigned to any group except all [18].

Using Ansible the desired state is defined using YAML in the form of Ansible playbooks. A playbook is created by specifying the following components:

- **hosts** - Defines the hosts on which the tasks are run. This is usually a hostname or group name. More complicated patterns can also be used. For example, multiple hostnames or groups could be specified.
- **tasks** - Specifies a list of tasks that are run on the target hosts. Each task defines a name and the module to be called. The task name is optional, but it is good practice to always give the task a descriptive name as it is shown in the execution process and helps to understand what is executed.
- **module** - A task specifies the module and its arguments to be called. Modules are scripts that are deployed to target hosts and actually perform the specified actions. Ansible modules are often idempotent, meaning that executing the module multiple times using the same arguments has the same result.

Figure 3 shows a simple example of an Ansible playbook that runs tasks on all hosts specified in the inventory and creates a file named *test.txt* using the file module with arguments path and state.

```
1
2   - hosts: all
3     tasks:
4       - name: Create a test file
5         file:
6           path: "test.txt"
7           state: file
8
9
```

Figure 3. Ansible playbook that creates a file on all hosts

As Ansible is usually used to manage multiple hosts, it is common to still have differences between those hosts. For example, hosts requiring a specific software package may need different versions of it. Instead of creating separate playbooks to manage these hosts, Ansible allows creating variables to handle dynamic values. Inside the task, the variable name is used instead of the exact value and the variable itself is often defined in a separate file. Variables can also be used inside templates, which are files that allow using special placeholder syntax for including variables inside the file. Using Ansible, the templates are converted to regular files using the Jinja2[3] templating engine, which reads the template file, replaces variables with corresponding values and outputs a regular text file. It is often used to create configuration files, which require using dynamic values.

4.1.1 Roles

Ansible roles provide a way to group related content into an independent reusable component. Roles are created by grouping the content into a defined directory structure. In this thesis some of the more commonly used directories were:

- defaults - includes default values for variables used by the role.
- tasks - includes tasks performed by the role.
- templates - includes template files required by the role.

Ansible looks for a file named *main.yaml* by default in each of those directories. Roles cannot be executed on their own but are included as a part of a playbook. They allow breaking a complex

[3] https://palletsprojects.com/p/jinja/

playbook into smaller logical components. For example, a playbook deploying a web application and a database could be split into two separate roles, where one deploys the web application and another the database. This also improves the readability of the written automation code by forcing the use of a logical file structure.

4.2 Ansible setup

The Ansible project created for automating the setup and management of the UT HPC center Kubernetes cluster consisted of the following components:

- Variables - The variables are defined inside files and are used mainly to specify the Kubernetes and its applications versions and modifications to the application manifests.
- Inventory - The inventory was structured into the following three groups: cluster, control plane and worker nodes. The cluster group includes all hosts used in the Kubernetes cluster. The control plane group includes hosts used for running the control plane components. Lastly, the worker node group includes hosts used as worker nodes.
- Playbooks - The project includes playbooks *cluster.yaml* and *reset.yaml*. The *cluster.yaml* is the main playbook used for setting up and managing the cluster. The *reset.yaml* playbook is included for testing purposes and resets all nodes using the *kubeadm reset* command.
- Roles - The automation tasks were grouped into separate Ansible roles, which are used inside the *cluster.yaml* playbook. The created roles and actions they perform are described in detail in the following chapters.

The Ansible project can be used by first defining the hosts inside the inventory and assigning them to the previously mentioned groups. Then the variables need to be specified according to the desired cluster configuration. Finally, the automation process is run using the *ansible-playbook* command with the *cluster.yaml* playbook as its argument.

5 Preparing the machines

The HPC center provides already provisioned physical machines on which the Kubernetes cluster will be deployed. These machines provide the default installation of the CentOS[4] operating system and SSH access. In order to deploy the Kubernetes cluster, these machines have to be prepared which includes configuring the operating system and installing a suitable container runtime.

5.1 Configuring the operating system

All required tasks for configuring the operating system are separated into a role named *centos*, which allows applying the same system configuration to all machines in the cluster. The exact configuration depends on the chosen deployment tool and cluster components. This role performs the following configurations:

- Disables memory swapping as *kubelet* does not support it.
- Configures SELinux to run in permissive mode allowing containers access to the host filesystem.
- Disables firewalld as it may interfere with the Calico networking plugin. Calico's documentation suggests disabling the firewall and using the network policy feature to manage hosts' security [19].

5.2 Installing the container runtime

In this cluster deployment, the cri-o[5] container runtime was used, which is optimized for Kubernetes workloads. It is one out of many container runtimes that implements the Kuberenetes CRI. The process of installing the cri-o container runtime is performed by a role named *cri-o*. Separating the installation of the container runtime into its own role also allows swapping it out for another runtime in the future if needed. This role performs the installation using the following actions:

1. Loads and configures the *overlay* and *br_netfilter* kernel modules required for container networking.
2. Adds cri-o package repositories.

[4] https://www.centos.org/
[5] https://cri-o.io/

3. Installs cri-o using the systems package manager. The exact package version is configured by an Ansible variable.

The container runtime is upgraded by changing the package version variable's value, which then upgrades the cri-o package.

6 Setting up nodes

After the machines are successfully prepared for running Kubernetes, they first need to be set up before being used as a control or worker nodes in the cluster. This mainly involves installing Kubernetes management and deployment tools and *kubelet* on each machine. Kubernetes deployment tools make it easier to deploy and manage all of the distributed components conforming to Kubernetes' best practices. The functionalities of these tools can range from only deploying a minimum viable Kubernetes cluster to a fully production ready cluster. The following sections provide an overview of the tools and created automation processes for setting up each individual Kubernetes node.

6.1 Kubeadm

Kubeadm[6] is a command-line tool chosen to deploy and manage a minimum viable Kubernetes cluster that follows best practices. It was chosen for its minimal nature, which provides greater flexibility and more control over selecting and configuring the additional components. While more fully featured deployment tools can initially make it easier to deploy and manage the cluster, they can be problematic in case of errors for which solving requires paid support or an intricate understanding of the tool. Kubeadm automates the deployment process by combining all actions required for creating the cluster into two commands, *init* and *join*. The *init* command creates a new cluster from scratch by setting up and configuring the required control plane components on a single node. The example usage and partial output of the *init* command are shown in Figure 4. The *join* command is used to add a new control plane or worker node into an already initialized cluster. The example usage and output are shown in Figure 5. Both commands also incorporate multiple checks before and after performing any actions, ensuring the actions are performed in a safe manner.

[6] https://kubernetes.io/docs/reference/setup-tools/kubeadm

```
[vagrant@master1 ~]$ sudo kubeadm init --control-plane-endpoint 192.168.56.5:6443
--pod-network-cidr "10.10.44.0/20"
[init] Using Kubernetes version: v1.24.0
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet
connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images
pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default
kubernetes.default.svc kubernetes.
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost master1] and IPs
[192.168.56.11 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost master1] and IPs
[192.168.56.11 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
```

Figure 4. Usage and partial output of *kubeadm init* command

```
[vagrant@worker1 ~]$ sudo kubeadm join 192.168.56.5:6443 --token gk7tvy.x5yfx2diu6klowu6
--discovery-token-ca-cert-hash
sha256:6d8e76ceab42cfba2d06b8144d2c0b9d4fa7cabf851f011687b7de9bc6e2a4ea  a4ea
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm
kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file
"/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
```

Figure 5. Usage and output of *kubeadm join* command

6.2 Kubectl

The interactions with the Kubernetes API were performed using *kubectl*, which included the following tasks:

- determine versions of applications and cluster components.
- check the status of applications and nodes.
- declaratively manage state.

The declarative state management is extended by a tool called *kustomize*[7], which is provided as a *kubectl* native feature. Kustomize is used to customize Kubernetes manifests in a template-free way. In a basic scenario, it uses a special file named *kustomization.yaml* which defines manifest files and patches to those manifests. The patches are partial Kubernetes objects that only define fields to be modified. Kustomize uses the kind and name of the object to determine which patches should be applied to each manifest and merges them into the initial manifests.

6.3 Setting up the node

The process of setting up the node is separated into an Ansible role named *node/common*. In addition to setup, this role also includes the upgrade process of nodes. This role is applied to all control and worker nodes in the cluster. It can be logically split into these distinct processes: facts gathering, installing and upgrading. The fact gathering process determines information about the installed packages and also the status of the current node in the cluster. This information is needed to make further decisions about installing or upgrading the packages on the current node. If the fact gathering process does not find all the required packages the role performs the following actions for the installation process:

- Loads and configures the *br_netfilter* kernel module required by Kubernetes.
- Adds Kubernetes package repositories.
- Installs the *kubeadm*, *kubectl* and *kubelet* packages using the systems package manager if these are not present on the node.

It is important not to perform any upgrades on the *kubelet* package in the install process as it could interrupt the cluster's operations. If any deviation of the desired Kubernetes version is found, the role performs the following actions to upgrade the node:

[7] https://kustomize.io/

- Upgrades the *kubeadm* package to the desired version.
- Runs *kubeadm upgrade* command, which first verifies that the node can be upgraded safely and then performs actions for upgrading the node.
- Drains the node using *kubectl*, which allows Kuberenetes to move the scheduled workloads to other nodes without causing any interruptions.
- Upgrades *kubelet* and *kubectl* packages to the desired version and restarts the *kubelet* service.
- Uncordons the node using *kubectl*, which allows Kubernetes to schedule workloads on this node again.

7 Creating a Kubernetes cluster

After individual nodes are set up and ready for running Kubernetes components, they can be used to form a Kubernetes cluster. This involves setting up and configuring the control plane, worker nodes and networking for the cluster. The following sections provide an overview of the control plane and network architecture, which is visualized in Figure 6, and automation roles created for forming the cluster from individual nodes and deploying the networking.

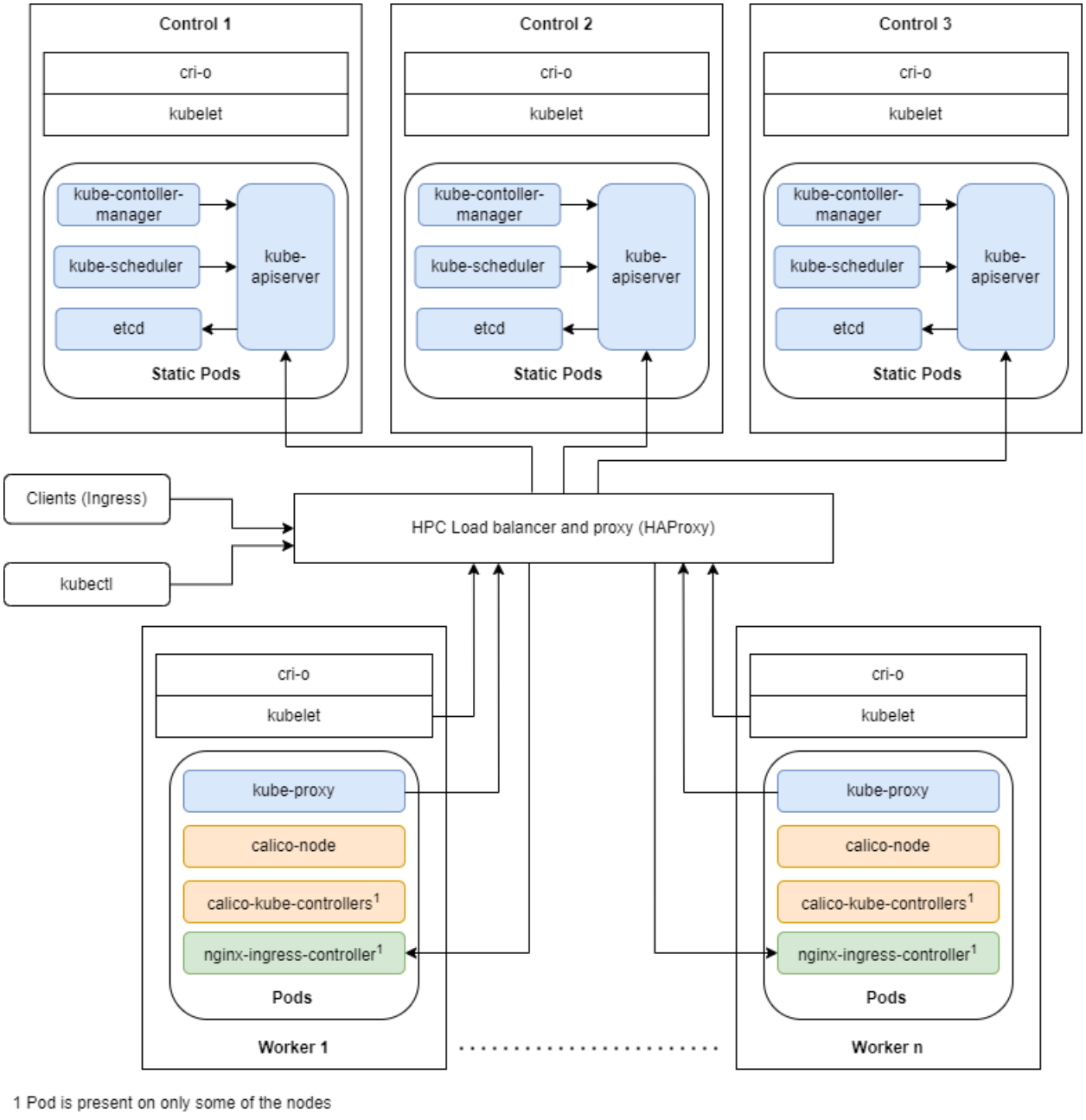

Figure 6 The cluster architecture

7.1 Control plane

The UT HPC center cluster control plane consists of three nodes, each running the following services: *kube-apiserver. kube-scheduler, kube-controller-manager* and *etcd* datastore. This is known as stacked *etcd* configuration, as the *etcd* service is deployed on the same node alongside other control plane services. The stacked configuration is easier to deploy and requires fewer physical resources, but offers worse redundancy than managing an external *etcd* datastore. On each node, the control plane services are run as static pods. Static pods are managed directly by *kubelet* and cannot be controlled by the control plane. These three nodes provide a high availability cluster allowing for a single node failure without impacting regular operation. Load balancing to each *kube-apiserver* instance is provided by an external HAProxy[8] load balancer managed by the HPC center and its deployment and configuration are left to the cluster administrators.

7.2 Networking

The networking solution consists of a network plugin and ingress controller, which both are deployed as Kubernetes applications. In this context, Kubernetes application refers to application deployed and managed only using Kubernetes objects. The UT HPC center cluster uses the Calico networking plugin. Calico[9] is a container networking and security solution. It is deployed using the default configuration, which uses an overlay network to allow routing packets between pods on different nodes. Overlay networks create a new network layer on top of an already existing one and are easier to deploy as the underlying network does not have to be made aware of the cluster network. The Calico networking plugin is configured to use the Kubernetes datastore for storing the configuration and operational state. Its deployment consists of *calico-node* and *calico-kube-controllers* workload objects. The *calico-node* is a DaemonSet object, which is responsible for setting up the network on each individual node. The *calico-kube-controllers* is a Deployment object, which runs a collection of different controllers required for setting up and managing the network.

[8] http://www.haproxy.org/
[9] https://www.tigera.io/project-calico/

7.2.1 Ingress

The cluster uses Ingress-NGINX[10], which is an ingress controller maintained by the Kubernetes project, as the ingress controller implementation, which uses NGINX[11] for providing the traffic routing specified by Ingress objects. NGINX is a web server that also provides load balancing and reverse proxying capabilities. The deployment consists of the *ingress-nginx-controller* Deployment object, which monitors cluster Ingress objects and performs the required configuration of the NGINX instance. The ingress controller is exposed using a NodePort service, which makes it only accessible inside the host network. Access from the public network is provided by an external HAProxy reverse proxy, which is managed by the UT HPC center.

7.3 Setting up control plane nodes

The tasks required for setting up a control plane node are included inside an Ansible role named *node/control.* This role requires the first control plane node to be specified, as it will initialize the cluster on that node using the *kubeadm init* command. Secondary control plane nodes are joined to the cluster using the *kubeadm join* command. This role also makes modifications to the static Pod manifests created by *kubeadm* to allow further configuration of control plane components. These modifications are done by copying the manifest patches defined by a variable to a directory on the node and then providing the directory as the value to the patches argument for *kubeadm* commands. The process is similar to patching with *kustomize*, but instead of having a *kustomization.yaml*, it uses a special file naming convention that includes information about the component to be patched, patching strategy and order. Finally, the role ensures that the node is healthy by checking that all Pods for this node inside the *kube-system* namespace are in the ready state. This is done using the *kubectl wait* command, which allows specifying the completion condition and waits until the condition is fulfilled. This health check is especially important as in the case of misconfiguration it would propagate also to other control plane nodes bringing down the whole cluster.

[10] https://kubernetes.github.io/ingress-nginx/
[11] https://www.nginx.com/

7.4 Setting up networking

The tasks for automating the setup and configuration of the Calico networking plugin and Ingress-NGINX ingress controller are provided using Ansible roles *calico* and *nginx*. The created roles are very similar as both are deployed as Kubernetes applications using the default manifests provided by the official sources. The roles download the default manifests onto one of the control plane nodes, using the version provided by a variable. Customizations to the default manifests are provided using *kustomize*, which requires creating the *kustomization* directory by performing the following steps:

- Creates a directory for patch and manifest files.
- Copies the default manifests to the created directory.
- Copies patches defined by a variable to the created directory.
- Creates a *kustomization.yaml* file that references the default manifests and copied patches.

After the correct directory structure and files are created, the desired state is applied to the cluster using the *kubectl apply -k* command which processes the *kustomization* directory. The *-k* parameter specifies to process the provided directory using *kustomize*. Finally, the roles ensure that the deployment was successful using the *kubectl wait* command to ensure the created workload objects are in the ready state.

7.5 Setting up worker nodes

The tasks for setting up a worker node are included in an Ansible role named *node/worker*. This role is run after the *common* role on all worker nodes. It joins the worker node to the cluster using the *kubeadm join* command. Similar to the *node/control* role it ensures the worker node is healthy by checking that all Pods for this node inside the *kube-system* namespace are in a ready state using the *kubectl wait* command.

8 Automated cluster management

The usage of the Ansible automation setup created in this thesis for setting up and managing a Kubernetes cluster has the following prerequisites:

- Machines used for Kubernetes nodes have been prepared by installing the CentOS operating system and setting up SSH access.
- A load balancer has been set up and configured, which provides access to the *kube-apiserver* instances running on control plane nodes.
- A reverse proxy has been set up and configured, which provides access from the public network to the clusters' ingress controller.

8.1 Management process

Kubernetes cluster administrators often need to carry out various actions to manage the cluster. The automation solution created in this thesis allows performing many of these actions automatically by using the provided Ansible playbook.

Kubernetes cluster resources can be increased by adding additional nodes to the cluster. Manually performing this action requires obtaining a join token by executing the *kubeadm token* command on a control plane node. After obtaining the token, the new node is added to the cluster by executing *kubeadm join*. Using the automation solution this process is performed by adding the hostname of the new node to the inventory file and assigning it to either worker or control plane group. Then running the main playbook adds the node to the cluster.

Keeping the cluster up-to-date with the latest updates requires upgrading the components on every node. It is manually done by connecting to each node in the cluster and performing the required actions to upgrade the components to the desired version. Using the automation solution this is done by updating the Kubernetes cluster version variable and running the provided playbook, which performs the upgrade process automatically on each node.

As the requirements for the cluster change over time, the control plane components configuration may need to be adjusted accordingly. The components are deployed as static pods, for which

modifications require connecting to each control plane node and manually changing the Pod manifest files. Using the automation solution, the modifications are supplied as patch files in the Ansible project and defined using a variable. After configuring the wanted modifications, running the playbook applies these patches to the manifest files on each control plane node.

8.2 Limitations

The automation solution created in this thesis was designed for small clusters and only sets up and manages the control plane, worker nodes and networking. This imposes some limitations to the usability of the cluster, which need to be accounted for.

The cluster size is limited to a maximum of 50 nodes. This limitation mainly comes from the chosen Calico networking plugin configuration. Calico's documentation suggests using a special scaling daemon for clusters bigger than 50 nodes [20]. As the UT HPC center Kubernetes cluster is under 50 nodes it was deployed without it for simplicity. In addition, the provided Ansible playbook is configured to finish executing all tasks on a single host before moving on to the next one, which may take too long to be suitable for use with a large number of nodes.

The automation solution does not deploy and configure any highly available persistent storage solution. Only local storage is available in the cluster by default, which is backed by each individual node. This means the pod's persistent storage is provided by a specific node and in case of a failure, Kubernetes is unable to reschedule this pod to run on another node. Any highly available persistent storage solution deployed as a Kubernetes application could be added by creating an Ansible role similar to the ones created in this thesis for setting up and managing the Calico networking plugin and Ingress-NGINX.

8.3 Potential cluster use cases

The Kubernetes cluster deployed by the automation solution is suitable for most containerized workloads that do not require highly available persistent storage. The cluster provides high availability and resources from multiple physical machines for running containers. Some of the potential use cases include:

- **Running server applications -** The server application can be deployed using the Deployment object, which allows the application to be scaled and upgraded without downtime. Access to the application from the public network can be provided using the cluster ingress.
- **Performing distributed computations -** The computations can be performed by containers deployed in the cluster, which may be scaled as needed. The work done by the containers could be coordinated by a separate application.
- **Serving static web pages -** The web page can be packaged into a container image alongside the web server, which serves it. The container can then be deployed in the cluster and access from the public network can be provided using the cluster ingress.

9 Summary

In this thesis automation solution was created for automating the setup and management of the University of Tartu HPC center Kubernetes cluster's control plane, worker nodes and networking. The cluster operators can use this solution to set up the Kubernetes cluster and also perform administrative tasks, which include adding a node to the cluster, upgrading and modifying the configuration of the control plane and networking components.

Creating the automation solution required to automate the tasks for preparing the machines, setting up Kubernetes nodes, forming the cluster from individual nodes and installing networking. These processes were automated by combining the required actions into an Ansible playbook. Before the playbook can be used, the cluster operators first define the hosts for the control plane and worker nodes and then the desired cluster components' versions and configurations. After that, running the playbook tries to automatically perform the required actions to satisfy the desired state. Upgrades to the cluster are performed by updating the defined versions and rerunning the playbook. The created playbook also incorporates various health checks to ensure the actions are carried out successfully.

The automation solution has limitations on the size of the cluster and the usage of persistent storage. The automation solution could be further extended by improving the network plugin configuration and providing a persistent storage solution to overcome these limitations.

References

[1] “What is a container?” https://www.docker.com/resources/what-container (accessed Mar. 13, 2022).
[2] “Containers in the enterprise.” https://www.ibm.com/downloads/cas/VG8KRPRM (accessed Mar. 13, 2022).
[3] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, “A new container scheduling algorithm based on multi-objective optimization,” *Soft Comput*, vol. 22, no. 23, pp. 7741–7752, Dec. 2018, doi: 10.1007/s00500-018-3403-7.
[4] “Container Images: Architecture and Best Practices.” https://www.aquasec.com/cloud-native-academy/container-security/container-images/ (accessed Mar. 14, 2022).
[5] “About the Open Container Initiative.” https://opencontainers.org/about/overview/ (accessed Mar. 14, 2022).
[6] “What is container orchestration?” https://www.redhat.com/en/topics/containers/what-is-container-orchestration (accessed Mar. 14, 2022).
[7] “TechCrunch: As Kubernetes hits 1.0, Google donates technology to newly formed Cloud Native Computing Foundation.” https://www.cncf.io/news/2015/07/21/techcrunch-as-kubernetes-hits-1-0-google-donates-technology-to-newly-formed-cloud-native-computing-foundation/ (accessed Mar. 14, 2022).
[8] “11 Facts about real-world container use.” https://www.datadoghq.com/container-report-2020/ (accessed May 10, 2022).
[9] “Kubernetes Documentation Style Guide.” https://kubernetes.io/docs/contribute/style/style-guide/ (accessed May 10, 2022).
[10] “Kubernetes Pods.” https://kubernetes.io/docs/concepts/workloads/pods/ (accessed Mar. 22, 2022).
[11] “Kubernetes Components.” https://kubernetes.io/docs/concepts/overview/components/ (accessed May 02, 2022).
[12] “Kubernetes API.” https://kubernetes.io/docs/concepts/overview/kubernetes-api/ (accessed Mar. 22, 2022).
[13] “Kubernetes Scheduler.” https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (accessed Apr. 10, 2022).
[14] “Controllers.” https://kubernetes.io/docs/concepts/architecture/controller/ (accessed Apr. 10, 2022).
[15] “etcd.” https://etcd.io/ (accessed Apr. 10, 2022).
[16] “kubelet.” https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (accessed Apr. 10, 2022).
[17] “Understanding Kubernetes Objects.” https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (accessed Apr. 11, 2022).
[18] “How to build your inventory.” https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#default-groups (accessed May 08, 2022).

[19] “Calico System Requirements.” https://projectcalico.docs.tigera.io/getting-started/kubernetes/requirements (accessed Apr. 30, 2022).
[20] “Install Calico for on-premises deployments.” https://projectcalico.docs.tigera.io/getting-started/kubernetes/self-managed-onprem/onpremises (accessed May 08, 2022).

Appendix

I Source code

The Ansible playbooks and roles created for the automation process are available on GitHub in this repository https://github.com/ekoemets/ansible-kubernetes.

II License

Non-exclusive license to reproduce the thesis and make the thesis public

I, Emil Koemets,

1. grant the University of Tartu a free permit (non-exclusive license) to: reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis Kubernetes cluster automated deployment with Ansible in the University of Tartu High Performance Computing center, supervised by Ivar Koppel and Sander Kuusemets.
2. I grant the University of Tartu the permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright,
3. I am aware that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Emil Koemets

10/05/2022