POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Automation of Edge Datacenter Infrastructure

Supervisors

Prof. Fulvio RISSO

Dott. Raffaele Giuseppe TRANI

Candidate

Teodoro CORBO

July 2023

Summary

Cloud-native technologies, marked by their inherent flexibility, scalability, and cost-effectiveness, have revolutionized the IT infrastructure landscape of large-scale organizations, especially major telecommunications companies (telcos). Among these technologies, Kubernetes (K8s) has emerged as a leading platform for managing containerized workloads and services. This shift to cloud-native solutions becomes crucial for telcos seeking to efficiently manage growing data traffic and provide a wide array of services.

One of the critical obstacles in this digital transformation journey involves the complexity associated with managing bare metal infrastructure, particularly in the context of multi-cluster environments at the network edge. This challenge underscores the importance of the work presented in this thesis.

Edge networks, situated closer to end-users and crucial for minimizing latency and quick data processing, often include multiple bare metal nodes interconnected by network switches. The configuration of these switches, however, demands meticulous manual work, such as setting up network interfaces, VLANs, routing protocols, and security settings. This process is not only labor-intensive but also highly susceptible to human error, thereby leading to inefficiencies and potential system vulnerabilities.

Addressing these concerns, this thesis presents a novel architecture that leverages the principles of Sylva, an initiative by Linux Foundation Europe, and the extensibility of Kubernetes. **The architecture seeks to automate the traditionally manual process of network configuration in edge sites with bare metal nodes**, reducing human intervention, increasing accuracy, and significantly enhancing operational efficiency.

The proposed architecture pivots around two core elements: the Network Operator and the Actuator Operators. The Network Operator utilizes Custom Resource Definitions (CRDs) to outline the intended network topology and configuration, while the Actuator Operators serve as the communicative conduit between the Network Operator and the physical network devices.

The architecture's implementation utilizes Kubebuilder, an open-source Software Development Kit (SDK).

A significant component of this implementation is the development of an Actuator Operator compatible with Cisco Network Services Orchestrator (NSO), a model-driven software used to automate the provisioning and configuration of network devices. This operator communicates with NSO using RESTCONF APIs, facilitating real-time execution of configuration modifications and maintaining the alignment between the desired and current states of network configurations.

In summary, this thesis addresses a significant challenge in the telecommunications networking domain, providing a cloud-native and Kubernetes-based automated solution for managing and configuring network resources at the edge sites. By resolving the complexities of bare metal nodes' setup, it contributes a meaningful stride towards the digital transformation of network infrastructure management.

Ringraziamenti

Ringrazio i miei genitori, Chiara, la mia famiglia, senza di voi tutto questo non sarebbe stato possibile, mi avete sostenuto sempre, in ogni momento, credendo in me fino alla fine.

Un ringraziamento speciale va alle nonne Lucia e Giovanna, ai nonni Michele e Teodoro che ho sempre nel cuore, a tutti gli zii e i cugini per l'affetto immenso che mi avete sempre dimostrato.

Ringrazio Lisa, tu che ci sei sempre stata, nel bene e nel male spronandomi a fare sempre meglio, ad impegnarmi, ad andare avanti anche quando magari avrei voluto mollare tutto.

Ringrazio gli amici veri, Antonio, Francesco, Pier, Andrea, l'ormai francese Giulio, gli amici della "5/6 fila", i "Terroni a Torino", gli "amici di giù", il "gruppo Taranto", avete reso il tutto più bello, senza di voi il Poli avrebbe preso il sopravvento.

Ringrazio Claudia, sei entrata nella mia vita in un periodo difficilissimo prendendomi per mano ed insegnandomi il bello che si nasconde anche dietro i piccoli gesti.

Esprimo la mia gratitudine al prof. Risso per la passione che ha saputo trasmettermi in questi anni e la disponibilità dimostrata.

Un grazie di cuore va a Tim e ai futuri colleghi Carlo, Federico, Alessandro, Roberto e Raffaele che mi hanno accolto in azienda e guidato con grande professionalità durante lo svolgimento del lavoro di tesi.

Desidero infine ringraziare tutte le persone che non ho nominato direttamente, ma che hanno contribuito a questo percorso. Ogni passo compiuto, ogni conoscenza acquisita, ogni esperienza vissuta, sia positiva che negativa, ha lasciato un'impronta indelebile rendendo tutto ciò possibile.

Table of Contents

List of Figures IX

Acronyms XI

1 Introduction 1

1.1 Goal of the thesis 1

1.2 Structure of the work 1

2 Kubernetes 3

2.1 Kubernetes History 3

2.2 Applications Deployment Evolution 4

2.3 Architecture 5

2.3.1 Control Plane 6

2.3.2 Nodes 7

2.4 Kubernetes Objects 7

2.5 Networking in Kubernetes 11

2.6 Security and Roles Access 12

2.6.1 Role-Based Access Control 13

2.7 Extending Kubernetes: Operators and Custom Resources 13

2.7.1 Custom Resources 13

2.7.2 Controllers 14

2.7.3 Operators 14

2.7.4 Kubebuilder 15

3 Cisco Network Services Orchestrator 16

3.1 Introduction 17

3.2 Features 17

3.3 Architecture 19

3.4 Workflow 19

3.5 Communication with Network Devices 20

3.5.1 NETCONF 21

3.5.2 CLI . . . 21
3.6 Interaction with NSO using RESTConf . . . 21
3.7 Service Invocation through RESTConf . . . 21
3.8 Conclusion . . . 22

4 Project Sylva 23
4.1 Introduction to Sylva . . . 23
4.2 Main Objectives and Benefits . . . 23
4.3 Technical Aspects of Project Sylva . . . 24
4.4 Hybrid Deployment and Bare Metal Automation . . . 27

5 Network Edge Automation for Bare Metal Infrastructure 29
5.1 Challenges . . . 30
5.1.1 The Need for Automation at the Edge . . . 30
5.2 Multi-distributed Cloud Ecosystem . . . 31
5.2.1 Management Cluster . . . 31
5.3 Architecture Proposal . . . 32
5.3.1 Network Operator . . . 33
5.3.2 Actuator Operators . . . 34

6 Implementation of the Proposed Architecture 36
6.1 Building Blocks: Operators and CRDs . . . 36
6.1.1 Kubernetes Operators . . . 36
6.1.2 Custom Resource Definitions (CRDs) . . . 38
6.2 Implementing them with Kubebuilder . . . 42
6.2.1 Creating the CRDs and controller . . . 43
6.2.2 Defining the CRD Spec and Status . . . 43
6.2.3 Implementing the Controllers . . . 44
6.3 Actuator Operator for NSO . . . 45
6.3.1 Resource Management . . . 45
6.3.2 Communication with NSO . . . 46
6.3.3 RESTCONF API Calls . . . 47
6.3.4 Role-Bases Access Control integration . . . 49
6.4 Implementation Workflow . . . 51
6.4.1 Creating a New VlanService . . . 51
6.4.2 Modifying a VlanService . . . 52
6.4.3 Deleting a VlanService . . . 53

7 Simulation and Result 54
7.1 Benchmark Hardware Specifications . . . 54
7.2 Benchmark Results . . . 55
7.3 Conclusion . . . 56

8 Future Perspectives 58
8.1 Scenario Overview . . . 58
8.2 ClusterAPI . . . 58
8.2.1 Key Components . . . 58
8.2.2 Deploying a cluster . . . 59
8.3 Metal3 . . . 60
8.3.1 Major Components of Metal3 . . . 61
8.3.2 Provisioning Process . . . 62
8.3.3 Retrieving Information about the Physical Host . . . 63
8.3.4 Conclusion . . . 64
8.4 Integration of ClusterAPI and Metal3 . . . 65
8.4.1 IP Address Management (IPAM) . . . 65
8.5 Conclusion . . . 67
8.5.1 Next Steps . . . 67
8.5.2 Final Remarks . . . 68

9 Conclusion 69

Bibliography 71

List of Figures

2.1 Evolution of applications deployments . . . 4
2.2 Kubernetes architecture . . . 6

4.1 Sylva's collaborators . . . 23
4.2 Five pillars of Sylva . . . 24
4.3 Distrubuted Cloud . . . 25

5.1 Multi-distributed cloud ecosystem . . . 31
5.2 General architecture . . . 32
5.3 Proposed architecture . . . 33

6.1 CRDs implementation . . . 38
6.2 NSO interactions . . . 46
6.3 Workflow . . . 51

7.1 NSO API calls response time . . . 56

8.1 BareMetalHost operator . . . 61
8.2 metal3 inspection workflow . . . 64
8.3 IPAM workflow . . . 65
8.4 Smart Network Operator . . . 67

Acronyms

telco
: telecommunications company

k8s
: kubernetes

NSO
: Network Services Orchestrator

CRD
: Custom Resource Definition

RBAC
: Role-based access control

VM
: Virtual machine

CNF
: Cloud-native Network Function

CaaS
: Container-as-a-Service

Maas
: Metal-as-a-Service

VLAN
: Virtual Local Area Network

HPC

High-Performance computing

Chapter 1

Introduction

Cloud computing has become increasingly crucial in recent years for its capacity to provide scalable and readily available services, fostering innovation and offering flexibility. This holds true for telecommunications companies as cloud-native technologies allow for the rapid deployment of new services, ultimately leading to a competitive edge in the industry. In this context, a crucial aspect for telecommunication companies (telcos) is the automation of the entire network infrastructure, where starting from a simple configuration, the entire cluster infrastructure is built. Managing and configuring nodes composed of virtual machines is relatively easy with current technologies. However, when it comes to physical nodes (bare metal servers), there is a significant gap in how to configure the network for these devices.

1.1 Goal of the thesis

The goal was to design an architecture and develop an implementation to automate the configuration of devices at the cluster's edge. Specifically, in a Kubernetes (k8s) cluster scenario with bare metal nodes connected to one or multiple switches. The goal is to develop a controller and Custom Resource Definition (CRDs) system that seamlessly integrates within the cluster, possesses knowledge of the network topology, and establishes communication with switches for accurate network configuration.

1.2 Structure of the work

The thesis will unfold with the following structure:

- **Chapter 2** provides an overview of Kubernetes, the technology that enables the orchestration and deployment of cloud-native applications.

- **Chapter 3** provides an overview of Cisco Network Services Orchestrator, a tool that allows for the management, connectivity, and configuration of switches.
- **Chapter 4** introduces the project Sylva, main objective and why it's so important for telcos.
- **Chapter 5** introduces the challenges and describes the proposed solution.
- **Chapter 6** provides an overview of the programming language used, the implementation of Custom Resource Definition (CRDs) and controllers, and the management of security permissions and roles (RBAC).
- **Chapter 7** provides a description of the testing environment used, the achieved results, and the performance obtained.
- **Chapter 8** discusses the future objectives of this thesis, including the potential integration with popular modern tools like Metal3 and ClusterAPI for optimal management of the entire network topology.
- **Chapter 9** presents the conclusions drawn from the accomplishments made throughout the thesis.

Chapter 2

Kubernetes

In this chapter, we delve into the fascinating world of Kubernetes, tracing its historical progression and continued evolution over time. Known commonly by its abbreviated form, K8s, Kubernetes represents an expansive framework with layers of complexity and a broad set of capabilities. While a comprehensive examination would require a more extended exploration, we concentrate here on its pivotal concepts and components, particularly focusing on its extensibility.

We underscore the versatility of Kubernetes by exploring Custom Resource Definitions (CRDs) and Operators, elements at the heart of Kubernetes' extensibility. These powerful tools allow Kubernetes to evolve beyond its out-of-the-box functionality, catering to diverse and specialized application requirements.

Further, we spotlight Kubebuilder, a tool essential for building these custom resources, providing an accessible framework for leveraging the extensibility features of Kubernetes.

2.1 Kubernetes History

Kubernetes, also known as K8s, is an open-source platform designed to automate deploying, scaling, and managing containerized applications [1]. The system was born at Google, based on the company's experience running services at large scale, and it borrows concepts from the Google Borg system .

Google open-sourced the Kubernetes project in 2014. Since then, it has become one of the largest and most active projects on GitHub [2]. Kubernetes is now maintained by the Cloud Native Computing Foundation (CNCF), which was founded in 2015.

The rapid adoption of Kubernetes can be attributed to the growing need for cloud-native technologies, which offer greater flexibility and scalability compared to traditional IT infrastructure. In particular, telecommunication companies

are increasingly adopting cloud-native approaches to manage their large-scale, distributed networks [3].

2.2 Applications Deployment Evolution

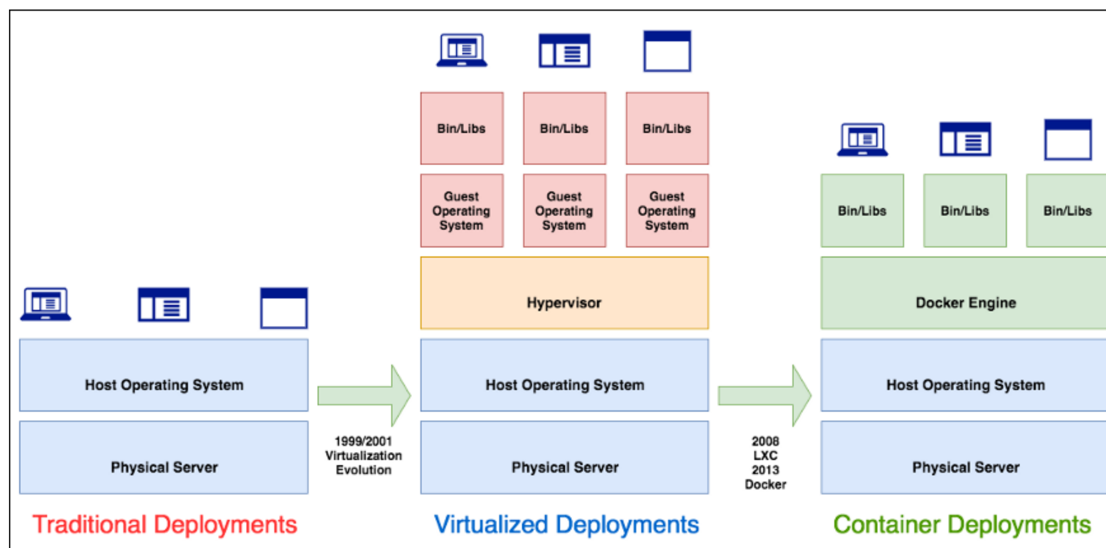


Figure 2.1: Evolution of applications deployments

The evolution of application deployment strategies is a story of relentless innovation and adaptation, reflecting the persistent shifts in technological capabilities and business requirements. It tracks the transition from static, monolithic deployments on physical servers to adaptable, decoupled applications running on container orchestration platforms like Kubernetes.

The earliest era of software application deployment was dominated by **physical servers.** In this so-called "bare-metal" approach, each application was intimately tied to the physical server it operated on. The server's hardware resources, such as its CPU, memory, and storage, were dedicated to a single application. This straightforward approach was plagued with significant drawbacks concerning resource utilization, scalability, and isolation between different applications operating on the same machine [4].

Suppose two applications were deployed on the same server; they would compete for the same pool of resources, often leading to performance bottlenecks. Scalability was another substantial challenge; augmenting more servers was a costly proposition and required considerable time and effort. The lack of isolation was another issue; an error in one application could compromise other applications running on the same server.

Virtualization technology emerged as a solution to these challenges. It enabled the concurrent running of multiple virtual machines (VMs) on a single physical server, with each VM operating as an independent system complete with its own operating system and dedicated resources. This breakthrough ushered in a new era of scalability and efficiency as creating a new VM to scale an application became a simple task. Additionally, isolation between VMs improved the stability and security of the overall system. However, VMs weren't without their limitations. They introduced significant overhead in terms of resource usage and management complexity due to running multiple full-fledged operating systems and managing these disparate systems.

In light of these constraints, the technology industry developed a more lightweight and manageable alternative to VMs, leading to the birth of **containerization**. Containers encapsulate an application along with its dependencies into a standalone unit that can operate anywhere, offering a highly portable, environment-agnostic solution for software deployment [5]. **Docker** emerged as the industry standard for containerization, providing tools and standards that significantly simplified the process of building, shipping, and running containerized applications.

Despite the advantages of containers, managing a large number of them across different servers presented new challenges, particularly around scheduling, networking, and scalability. **Kubernetes** was introduced to address these issues, providing a robust platform for orchestrating containers. K8s automates the deployment, scaling, and management of containerized applications, offering built-in solutions for service discovery, secret management, and storage orchestration [1].

It manages and coordinates containers across multiple physical or virtual machines, ensuring resource optimization and high availability. With its extensible architecture and vibrant community, K8s has become an indispensable part of modern cloud-native application deployment strategies.

This journey from bare-metal to Kubernetes encapsulates the ongoing evolution of software deployment, embodying the industry's pursuit of efficiency, portability, and scalability. Looking ahead, the advent of **serverless computing** and **edge computing** promise to spur further innovation in this space, setting the stage for the next chapter in the narrative of application deployment evolution.

2.3 Architecture

Kubernetes is constructed around a modular and highly extensible architecture. One can conceptualize its structure as divided into two core components: the control plane, which is the brain of the cluster that dictates its state, and the nodes, the workers where the actual applications are run [6].

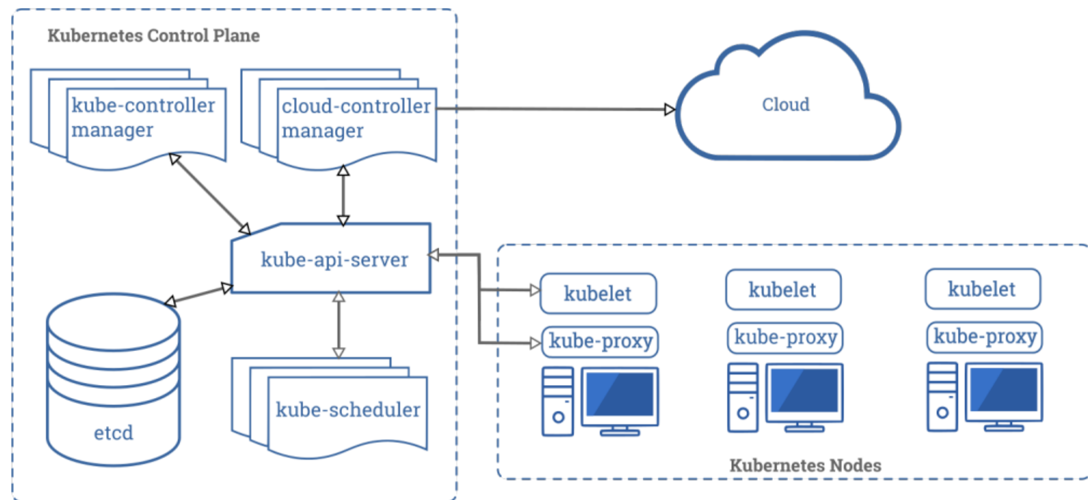


Figure 2.2: Kubernetes architecture

2.3.1 Control Plane

The control plane operates as the central management entity of a Kubernetes cluster. Its main role involves maintaining the desired state of the cluster, including which applications are running, which nodes those applications inhabit, and other operational particulars. To accomplish this, the control plane utilizes several key components:

- **Kubernetes API Server:** As the gateway to the Kubernetes control plane, the API Server exposes the Kubernetes API and represents the primary interface for cluster interaction. It serves as a communication hub that external entities (like end-users or external services) and the cluster's internal components use to perform various operations .
- **etcd:** This is a reliable, distributed key-value store that Kubernetes employs to manage the complete data landscape of the cluster. It stores all the data related to the cluster's state, configuration, and metadata. The integrity, reliability, and availability of etcd are vital to the functioning of a Kubernetes cluster .
- **Scheduler:** The scheduler is a critical component that takes charge of scheduling pods (the smallest deployable units of computing in Kubernetes) onto nodes. It continuously monitors for unscheduled pods and, once detected, assesses the cluster's state to decide the best node for the pod to run on. This decision-making takes into account numerous factors, including resource availability, policy constraints, data locality, and user specifications

- **Controller Manager:** The Controller Manager runs various controllers, which are essentially background processes tasked with maintaining the cluster's desired state. A noteworthy example is the Replication Controller, which ensures that the number of running replicas of a service matches the number defined in the specification. Controllers constantly reconcile the observed state of the cluster with the desired state

2.3.2 Nodes

Nodes are the workhorses of a Kubernetes cluster. They are the machines where the actual applications are run, each acting as a member of the cluster. Each node is a self-contained runtime environment equipped with the necessary services to support the containers running on it. The essential components of a node include:

- **Kubelet:** This is an agent that resides on each node in the Kubernetes cluster. The Kubelet is responsible for maintaining the state of the node and ensuring that all containers within the node's pods are running as expected. It communicates with the control plane to receive commands and report back the status of its operations

- **Kube-proxy:** Kube-proxy serves as a network proxy and load balancer for a node, facilitating network communication to the Pods from network sessions inside or outside of the cluster. It is responsible for managing network routing, IP addressing, and enforcing the network rules on the node .

- **Container Runtime:** This is the underlying software that is in charge of running containers. Container Runtime abstracts the application from the machine's specific hardware characteristics, providing a consistent environment for the application to run, regardless of the underlying infrastructure. Examples include Docker, containerd and others.

Taken together, these components form the basic building blocks of a Kubernetes cluster, each playing a distinct role in delivering a robust, extensible, and highly available platform for managing containerized applications at scale.

2.4 Kubernetes Objects

Kubernetes uses a set of objects to represent the state of a cluster, define configurations, and set policies. These objects are essentially records of intent–once created, the Kubernetes system works to ensure that the object exists and matches the provided specifications. A typical Kubernetes resource object has the following key elements:

- *APIVersion:* the versioned schema of this representation of an object.
- *Kind:* a string value representing the REST resource this object represents.
- *ObjectMeta:* metadata about the object, such as its name, annotations, labels etc.
- *ResourceSpec:* defined by the user, it describes the desired state of the object.
- *ResourceStatus:* filled in by the server, it reports the current state of the resource.

The operations allowed on these resources are the typical CRUD (Create, Read, Update, Delete) actions. Kubernetes system continually checks and ensures that the current state matches the desired state as defined in the specification.

Pods

Pods are the smallest and simplest objects in Kubernetes. A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that dictate how the container(s) should run.

Labels and Selectors

Labels are key-value pairs attached to a Kubernetes object and are used to organize and identify a subset of objects. Selectors allow grouping and selecting sets of objects with the same label.

Services

In Kubernetes, a Service is an abstract representation of a network service that exposes an application running on a set of Pods. It provides a stable endpoint for accessing the application within the cluster or externally, depending on its ServiceType.

There are different ServiceTypes available in Kubernetes:

- **ClusterIP**: This is the default ServiceType and is accessible only from within the cluster. It assigns a virtual IP address (ClusterIP) to the Service, allowing other components within the cluster to communicate with it.
- **NodePort**: A NodePort Service exposes the Service on a static port of each Node's IP address. This allows external access to the Service by contacting <NodeIP>:<NodePort>. It is commonly used for development and testing purposes.

- **LoadBalancer**: A LoadBalancer Service exposes the Service externally using a cloud provider's load balancer. This ServiceType automatically provisions a load balancer that distributes incoming traffic to the Service across multiple Pods, ensuring high availability and scalability.

- **ExternalName**: An ExternalName Service maps the Service to an external service by specifying its DNS name. This allows local applications to access the external service seamlessly, without exposing the details of the underlying implementation.

By using Services in Kubernetes, you can decouple the application from its underlying infrastructure and provide a consistent and reliable endpoint for accessing your services. Services play a crucial role in enabling communication between different components within the cluster and enabling external access to your applications.

Volumes

A Volume in Kubernetes is essentially a directory, possibly with some data in it, which is accessible to the containers in a Pod. Kubernetes Volumes enables data to survive container restarts and shares the data between containers within the Pod.

Namespaces

Namespaces are virtual partitions of a Kubernetes cluster providing scope for names, useful in environments with many users spread across multiple teams or projects. Kubernetes creates four namespaces by default:

- *kube-system:* This namespace houses objects created by the Kubernetes system, predominantly those associated with the control-plane agents.

- *default:* This namespace contains objects and resources created by users and is utilized as the default namespace for operations if no other namespace is explicitly specified.

- *kube-public:* As the name suggests, this namespace is accessible to all users, including those who are not authenticated. It serves special purposes such as exposing certain cluster-wide information to the public.

- *kube-node-lease:* This namespace is responsible for maintaining objects that hold heartbeat data from nodes, contributing to the efficient functioning of Node lifecycle events and performance.

ReplicaSet and Deployment

ReplicaSets control a set of pods, allowing for scaling the number of pods currently in execution. Deployments, on the other hand, manage the creation, update, and deletion of pods, automatically creating a ReplicaSet, which then creates the desired number of pods.

Example: Deployment in Kubernetes

Let's take a look at an example of a Deployment in Kubernetes. The following YAML code defines a Deployment for an Nginx web server:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 5
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to or removed from the cluster, Pods are added to or deleted from them accordingly.

EndpointSlice

An EndpointSlice is an abstraction that contains references to a set of network endpoints of a service. It provides a more scalable and extensible alternative to the original and deprecated Endpoint resource.

2.5 Networking in Kubernetes

Kubernetes networking plays a critical role in the overall functioning of a Kubernetes cluster. It enables communication between different components, ensuring that the system operates efficiently and coherently.

Pod Networking

Each Pod in a Kubernetes cluster is assigned its own IP address. This allows Pods to communicate with each other, across nodes in the cluster, without needing to implement NAT (Network Address Translation). This design aligns with Kubernetes' philosophy of treating Pods as the primary building blocks of applications.

Service Networking

While Pod networking ensures communication between Pods, Service networking allows for communication with external clients or between different components of an application within the cluster. Services, by design, are stable and do not change their IP addresses, unlike Pods that could be frequently created or destroyed. Thus, Services provide a reliable way to access applications.

Ingress and Egress Traffic

Ingress refers to the incoming traffic directed towards the Pods in the cluster, while Egress represents the outgoing traffic from the Pods to other services within the cluster or to external resources. Kubernetes provides several resources, like the Ingress Controller and Network Policies, to manage and control these traffic flows.

Ingress Controller

An Ingress Controller is a Kubernetes resource that manages incoming traffic based on the rules defined in Ingress Resources. It is typically responsible for load balancing traffic, SSL termination, and name-based virtual hosting.

Network Policies

Network Policies provide a way to control the traffic to and from Pods based on specified rules. By default, Kubernetes allows all incoming and outgoing traffic to Pods. Network Policies can be used to restrict this and allow only the desired traffic.

Container Network Interface (CNI)

Kubernetes uses the Container Network Interface (CNI) for Pod networking. The CNI is a set of specifications that define how network interfaces should be configured within containers and how they should interact with each other. Several CNI plugins, such as Calico, Weave, and Flannel, provide different methods of implementing network connectivity and different sets of features.

Network Architecture Models

There are different network models that Kubernetes can use. These models are defined by the CNI plugins used:

- **Flat Network Model:** In this model, all Pods can communicate with each other without NAT across nodes. Every Pod gets a unique IP within the cluster network, and there is no need to map container ports to host ports. This model simplifies communication and reduces latency.
- **Overlay Network Model:** In an overlay network, a virtual network is created on top of the existing network. This virtual network has its own IP addresses and subnet, isolated from the underlying network. The overlay network encapsulates the network packets into network packets that the underlying network can transmit.

Overall, networking in Kubernetes is a complex but critical aspect of the platform. The networking features ensure efficient communication and manageability within the cluster, supporting the scale and dynamism of containerized applications in Kubernetes. It plays a crucial role in maintaining the performance, security, and resilience of applications running on Kubernetes.

2.6 Security and Roles Access

As a distributed system, Kubernetes needs to address a variety of security concerns. The complexity arises from different aspects of security, including but not limited to container security, network security, API security, and the security of the

applications running on top of Kubernetes. Among these, an aspect that requires special attention is access control, for which Kubernetes primarily uses Role-Based Access Control (RBAC) .

2.6.1 Role-Based Access Control

RBAC is one of the critical aspects of Kubernetes security. It is an approach to restrict system access to authorized users. In Kubernetes, RBAC allows fine-grained control of what operations a user can perform on a given resource, such as pods, services, and volumes.

Roles in Kubernetes define the permissions to perform operations, such as read, write, and execute, on a set of resources. RoleBindings, then, link those roles to specific users or groups. Kubernetes also provides ClusterRoles and ClusterRoleBindings for defining roles and role bindings that apply to the entire cluster.

These RBAC mechanisms are vital for defining and enforcing access control in Kubernetes, ensuring that users have the necessary permissions they need to perform their tasks, while preventing unauthorized access to sensitive parts of the system.

2.7 Extending Kubernetes: Operators and Custom Resources

The capability to extend the Kubernetes API lies at the core of the Kubernetes design, facilitating robust customization and integration. Central to this extensibility are Kubernetes Operators and Custom Resources .

2.7.1 Custom Resources

At its core, the Kubernetes API is an endpoint that accepts and handles JSON objects. A **Custom Resource** is an extension of the Kubernetes API that allows you to create your own API objects. Custom Resources work like any other native Kubernetes object, like pods, deployments, or services.

The **Custom Resource Definition** (CRD) is used to define Custom Resources. A CRD provides a schema for the Custom Resource, enabling the API server to handle its lifecycle. This allows developers to leverage the Kubernetes platform's scalability, robustness, and other aspects for their custom applications .

2.7.2 Controllers

A **controller** in Kubernetes is a control loop that observes the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. They are the brains behind Kubernetes, continuously monitoring the state of the cluster and making adjustments to bring the system to the desired state .

Controllers work based on the principles of reconciliation and level-driven design. The **reconciliation principle** ensures that, regardless of the current state of the system, the controller will try to make changes to bring the system to the desired state. The level-driven design, on the other hand, allows controllers to be idempotent, ensuring that each execution, given the same inputs, yields the same results.

Examples of built-in controllers that ship with Kubernetes include the *replication controller, endpoints controller, namespace controller, and the serviceaccount controller*. However, Kubernetes also allows users to create custom controllers, similar to custom resources, extending the system's behavior to suit specific needs.

2.7.3 Operators

Operators build upon the foundational concept of Custom Resources. They are essentially custom controllers for Custom Resources, providing domain-specific knowledge of how an application should run and behave in different situations.

An Operator is composed of the following:

- *A Custom Resource Definition*: This defines the desired state of the application. Users can modify this desired state at any time, and the Operator will react to changes accordingly.

- *A Custom Controller*: This component watches the state of the application and makes adjustments to the application to bring it to the desired state. The Controller uses the Kubernetes API to create, update, and delete resources.

- *A set of application-specific permissions*: These permissions, defined via (RBAC), allow the Operator to carry out actions on behalf of the user .

Through the combination of these components, Operators offer a powerful and flexible way to manage complex applications in a Kubernetes-native way. They simplify application deployment, scaling, and management, allowing developers to focus more on the application logic and less on the operational details.

2.7.4 Kubebuilder

Kubebuilder, a significant open-source project maintained by the Kubernetes community, serves as a fast and effective framework for constructing and developing k8s APIs. It employs a wealth of libraries and tools from the Kubernetes ecosystem, thereby simplifying the creation process of custom resources and their corresponding controllers. This eases the management and control of applications [7].

Guided by the principle of declarative application management, Kubebuilder utilizes the power of APIs to manage application configurations, deployments, and runtime, mirroring Kubernetes' philosophy of declarative configuration and automation.

Adopting the principle of domain-driven design, it houses the logic associated with a particular API into an individual package. This approach enhances maintainability as each package encases both the API definition and the related controller.

Kubebuilder has several command-line interfaces (CLI) to create custom resource definitions (CRDs), controllers, and their related items. The following are some key commands:

- **Initialize a new project:** `kubebuilder init -domain example.com -license apache2 -owner "The Kubernetes authors"`
- **Create APIs and controllers:** `kubebuilder create api -group batch -version v1 -kind CronJob`
- **Generate CRD manifests:** `make manifests`
- **Generate and install CRDs into the Kubernetes cluster:** `make install`

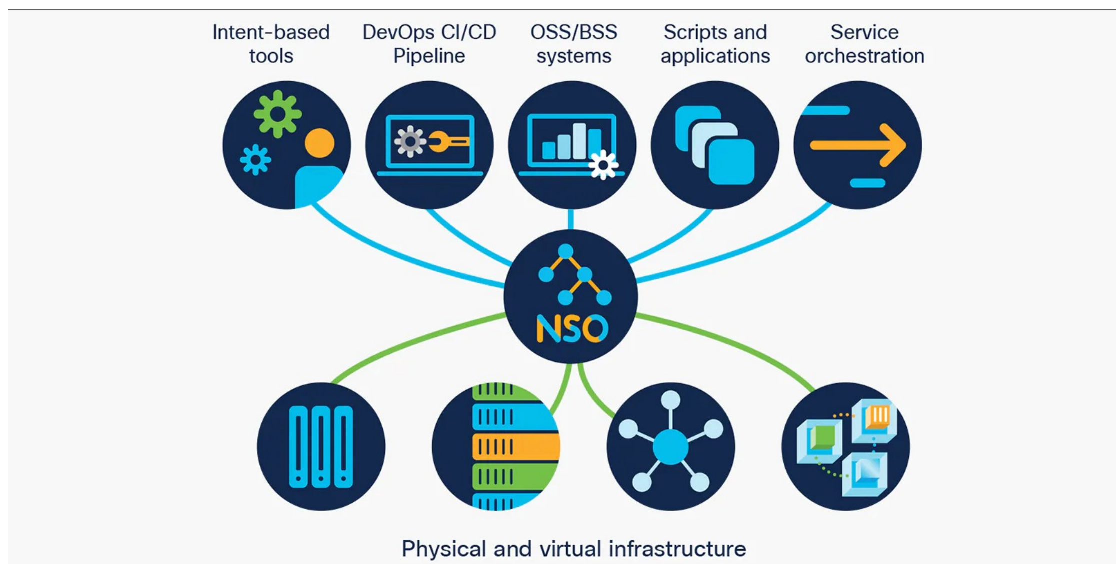
In the context of this thesis, Kubebuilder commands have been utilized to define and create CRDs, generate CRD manifests, and implement the controllers.

Chapter 3

Cisco Network Services Orchestrator

Cisco Network Services Orchestrator (**NSO**), developed by Tail-f Systems, is a *multi-vendor network orchestration platform* designed to provide comprehensive lifecycle service automation to transform and digitize networks. It is extensively used in service provider networks and larger enterprises to automate services across traditional and virtualized networks [8].

- NSO overview



3.1 Introduction

As networks grow in complexity and scale, the need for automation becomes increasingly critical. The requirement for faster service delivery, error-free configurations, and efficient resource utilization has led to the emergence of orchestration platforms like Cisco NSO.

NSO aims to deliver high-quality services faster and more easily through network automation. It uses network-wide, transaction-based processes to deliver services, ensuring error-free configurations and service consistency. By **providing a single, network-wide interface** for all network devices, irrespective of the vendor, it simplifies the entire process of network configuration and service delivery.

3.2 Features

Cisco NSO offers a wide range of features that enable efficient network orchestration and automation. Some key features include:

Network Abstraction and Modeling

NSO leverages YANG data modeling language to define the structure and semantics of network data. YANG provides a standardized way to describe data models and their relationships, making it easier to define and understand the configuration and operational data exchanged between NSO and network devices. This abstraction allows administrators to define services and network configurations in a vendor-agnostic manner, focusing on intent and desired behavior rather than device-specific configurations.

Multi-Vendor Device Support

NSO supports a wide variety of network devices from different vendors. It utilizes standard protocols like NETCONF and RESTConf to communicate with network devices and configure them. NSO's multi-vendor support enables organizations to manage and orchestrate heterogeneous networks with ease, eliminating vendor lock-in and streamlining network operations.

Service Orchestration

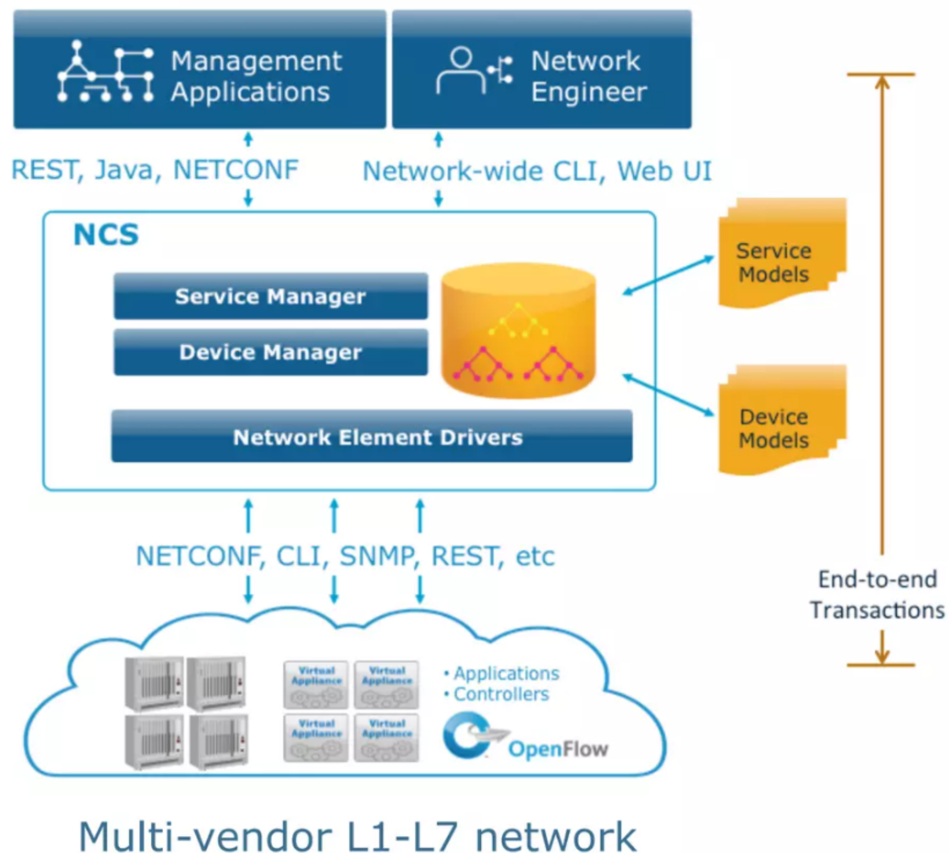
One of the key features of Cisco NSO is its ability to orchestrate services across multiple network devices. It enables the creation and management of complex service chains that span multiple network elements, such as routers, switches, firewalls, and load balancers. NSO provides a high-level abstraction of the **network**

service using YANG data models, allowing administrators to define services in a vendor-agnostic manner. NSO then translates this high-level intent into device-specific configurations and orchestrates the necessary changes across the network devices to provision and activate the service.

Built-in Fault Management

NSO includes built-in fault management capabilities that enable **proactive monitoring of network devices** and services. It can detect faults and deviations from desired configurations, triggering appropriate remedial actions. NSO's fault management capabilities help ensure the availability and reliability of network services by detecting and addressing issues in a timely manner.

- NSO Key Features



3.3 Architecture

Cisco NSO follows a distributed **client-server architecture**. The main components of the NSO architecture are: [9]

NSO Server

The NSO server is the core component responsible for managing and orchestrating network services. It stores the network configurations, data models, and service definitions. The NSO server communicates with network devices using protocols like NETCONF and RESTConf, and it processes service requests and configuration changes.

NSO Clients

NSO clients are the interfaces through which administrators interact with the NSO server. They provide a user-friendly interface, such as a CLI or a web-based GUI, to manage and configure network services. NSO clients allow administrators to define service specifications, perform service provisioning, and monitor the status and health of the network.

Datastore

The NSO datastore is a centralized repository that stores the network configurations, data models, and service definitions. It provides a consistent view of the network state and allows for easy retrieval and modification of network data. The datastore ensures that all components within the NSO architecture have access to the latest and synchronized network information.

Transaction Manager

The transaction manager handles the transactional operations in NSO. It ensures that all changes made to the network configurations are consistent and atomic. The transaction manager keeps track of configuration changes, allows for rollbacks in case of failures, and maintains a history of configuration revisions.

3.4 Workflow

A typical workflow in Cisco NSO involves several steps to deliver network services and manage network configurations. The workflow can be summarized as follows:

Service Definition

The administrator defines the network service using YANG data models. The YANG data models describe the desired behavior, configuration parameters, and relationships between different network elements. The service definition focuses on intent and abstracts away the device-specific configurations.

Service Provisioning

Once the service is defined, the administrator provisions the service using the NSO client interface. The NSO client translates the high-level service definition into device-specific configurations using the appropriate YANG models. NSO then pushes the configurations to the respective network devices using NETCONF or RESTConf protocols.

Configuration Deployment

NSO deploys the configurations to the network devices in a transactional manner. It ensures that all configurations are error-free, consistent, and in line with the desired service behavior. NSO communicates with network devices, validates the configurations, and handles any conflicts or errors that may arise during the deployment process.

Service Activation

Once the configurations are successfully deployed, NSO activates the service on the network devices. It coordinates the activation process across multiple devices, ensuring the service is provisioned and activated consistently across the network.

Service Monitoring and Management

NSO continuously monitors the network devices and services to ensure their proper functioning. It collects operational data, performs health checks, and alerts administrators in case of any faults or deviations from the desired state. NSO provides a comprehensive set of management tools to monitor, troubleshoot, and analyze the network performance.

3.5 Communication with Network Devices

Cisco NSO can communicate with network devices using various protocols, such as NETCONF and CLI. These communication mechanisms allow NSO to configure and manage the network devices in a vendor-agnostic manner.

3.5.1 NETCONF

NETCONF (Network Configuration Protocol) is a standardized network management protocol developed by the IETF. It provides mechanisms to install, manipulate, and delete the configuration of network devices. NSO utilizes NETCONF to communicate with network devices and exchange configuration information.

NETCONF operates over SSH (Secure Shell) and uses XML-based messages for data exchange. NSO communicates with network devices using the NETCONF protocol, allowing it to retrieve device configurations, push configuration changes, and retrieve operational data.

3.5.2 CLI

In addition to NETCONF, NSO also supports the Command Line Interface (**CLI**) for communication with network devices. CLI is a text-based interface used to configure and manage network devices. NSO provides built-in adaptors for various network device vendors, allowing it to convert high-level data models into device-specific CLI commands.

Using the CLI mechanism, NSO can interact with network devices that do not support NETCONF natively. This ensures compatibility with a wide range of network devices in the ecosystem.

3.6 Interaction with NSO using RESTConf

Cisco NSO also supports **RESTConf**, a RESTful API based on the principles of Representational State Transfer (REST). RESTConf provides a lightweight, HTTP-based interface for configuring network devices and accessing operational data.

RESTConf uses the YANG data modeling language for data representation and supports the basic CRUD (Create, Read, Update, Delete) operations through HTTP verbs. This allows applications to programmatically interact with NSO using standard HTTP methods.

By supporting RESTConf, NSO provides a modern, web-friendly interface that enables developers to integrate their applications with NSO and perform operations such as service provisioning, configuration management, and data retrieval.

3.7 Service Invocation through RESTConf

To invoke services in NSO using RESTConf, clients send **HTTP requests** to the NSO server with the appropriate RESTful API endpoints and payload. The payload typically contains the necessary data and parameters for the service invocation.

For example, to create a new network service, a client would send an HTTP POST request to the corresponding RESTConf endpoint, providing the service configuration data in the request body. NSO receives the request, validates the data, and performs the necessary operations to create and activate the service across the network devices.

Similarly, clients can use RESTConf to retrieve service information, update configurations, delete services, and perform other operations supported by NSO.

3.8 Conclusion

Cisco Network Services Orchestrator (NSO) is a powerful network orchestration platform that enables comprehensive lifecycle service automation. **It simplifies the process of network configuration** and service delivery by providing a single, vendor-agnostic interface. NSO's features, architecture, and workflow empower organizations to automate services, manage complex network infrastructures, and ensure efficient resource utilization. With its multi-vendor support and built-in fault management capabilities, NSO plays a crucial role in transforming and digitizing networks for modern-day requirements.

Chapter 4

Project Sylva

Project Sylva, launched by Linux Foundation Europe in November 2022, represents a monumental step towards accelerating the cloud adoption of network infrastructures within the European Union's privacy, security, and energy efficiency requirements [10].

4.1 Introduction to Sylva

European carriers (Telefonica, Telecom Italia, Orange, Vodafone, Deutsche Telekom) and two vendors (Ericsson and Nokia) collaborate to form the Sylva project. The partners aim to confront the myriad challenges prevalent in the deployment of telco and edge use cases both within the EU and on a global scale.

Figure 4.1: Sylva's collaborators

Sylva was conceived in response to the increasing fragmentation in the telco Cloud ecosystem that had been impeding operational model transformation. By introducing a **homogeneous telco cloud framework**, Sylva aspires to foster interoperability, flexibility, and ease of operation across the industry .

4.2 Main Objectives and Benefits

At the heart of Sylva is the commitment to constructing a cloud-native infrastructure stack. This solution is designed to support various telco applications (including 5G, OpenRAN, CDN, and others) and edge use cases. By facilitating the development of

such digital services, Sylva paves the way for new business opportunities, benefiting telco operators, vendors, and application developers alike

4.3 Technical Aspects of Project Sylva

Project **Sylva** seeks to address multiple significant technical hurdles, encompassing aspects such as network performance, the concept of distributed cloud (which involves multi-cluster Kubernetes and bare metal automation), energy efficiency, and security (including hardening and compliance). By prioritizing 'openness,' Sylva promotes a transparent, inclusive approach to technological advancements [11].

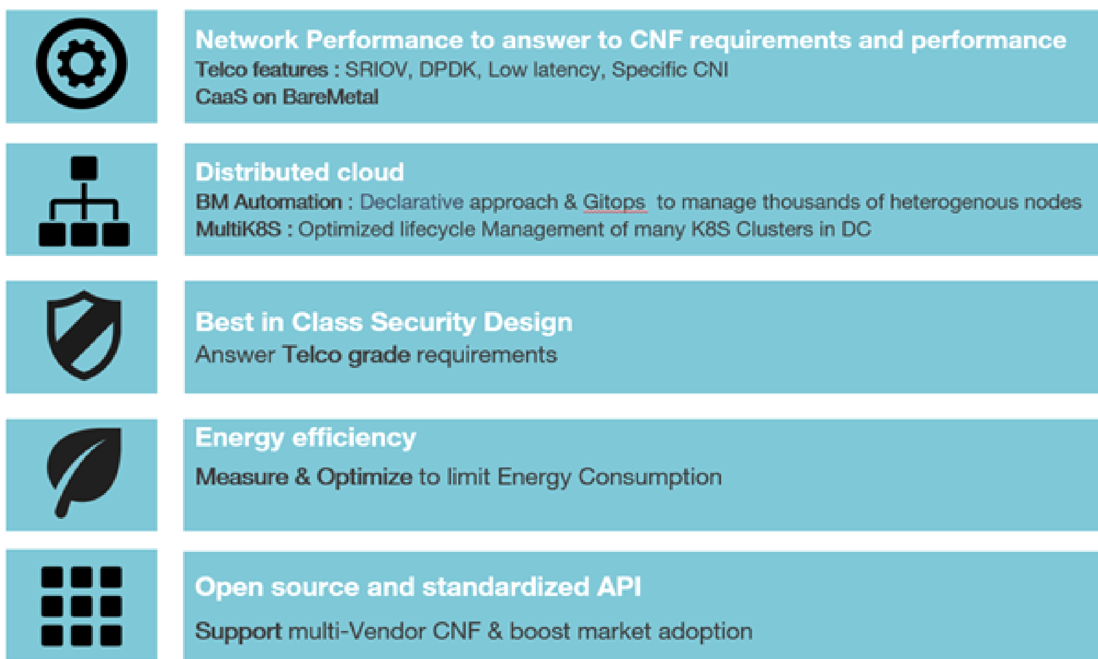


Figure 4.2: Five pillars of Sylva

Network Performance

Sylva implements performance requirements in line with the expectations of 5G Core and Open RAN Cloud Native Network Functions (CNF). These requirements include features like SRIOV or Near-Real time OS.

Distributed Cloud

Sylva proposes an architecture that can manage cloud infrastructures ranging from central locations to far-edge sites. This architecture allows organizations to deploy and manage applications across multiple geographic locations, bringing the benefits of cloud computing closer to end-users and enabling low-latency and high-performance services.

The **distributed cloud** architecture encompasses a combination of centralized cloud resources and distributed edge computing nodes. This distributed infrastructure enables organizations to leverage the scalability, flexibility, and cost-efficiency of cloud computing while catering to the specific requirements of edge computing scenarios, such as reduced latency, data sovereignty, and local data processing.

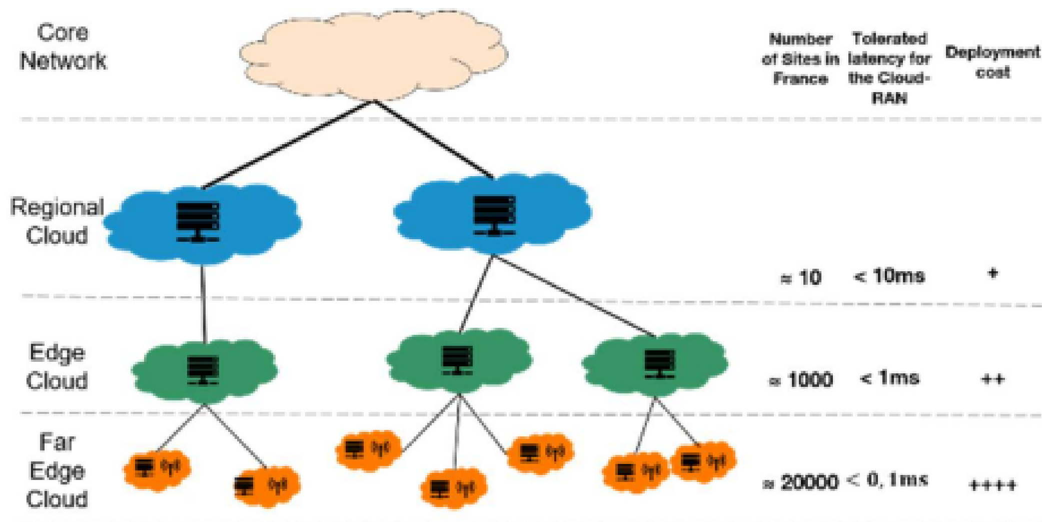


Figure 4.3: Distrubuted Cloud

Declarative Approach

The declarative approach is instrumental in managing a vast number of physical nodes and Kubernetes clusters. This method involves describing the desired infrastructure in a declarative way and using the Kubernetes reconciliation framework and controllers to manage the lifecycle of the infrastructure components based on the difference between the current and the declared state. It marks a new phase in the automation journey of the network management, from the build (day 0), run (day 1), and operate (day 2) stages, with deployment process, fault management,

and upgrade being natively automated.

Kubernetes Clusters Life Cycle Management

Sylva seeks to define an effective method for managing multiple k8s clusters. The need for managing a high volume of clusters arises from the following reasons:

- In a distributed cloud, an edge site (for example, a radio site) can face connectivity issues. In such scenarios, a multi-Kubernetes approach is more robust than a stretched Kubernetes cluster.
- Due to the lack of hard multitenancy in Kubernetes, some critical CNFs require a dedicated Kubernetes cluster for high security isolation, and possibly on isolated hardware.

Security

Given the rise in cyber-attacks against Telco (as evidenced by the GSMA cyber threat report 2021), state and EU regulations have tightened their expectations. In response, the Sylva project continuously integrates security requirements

Energy Efficiency

Sylva's design is mindful of energy consumption control, considering the large volume of nodes it aims to manage. The project is set to integrate mechanisms to measure consumption with advanced analysis per microservice, with the goal of identifying optimized or inefficient CNF under real conditions. Additionally, Sylva plans to integrate optimization mechanisms based on Kubernetes scaling capabilities.

Open Source and Standardized API

Adhering to the principles of openness and collaboration, Sylva commits to the use of open-source and standardized APIs, contributing to the larger ecosystem of network development.

Modular Design

Sylva's architecture is intentionally modular. It integrates numerous open-source components, which can be replaced with alternative projects as necessary in the future. This flexibility allows Sylva to remain on the cutting edge of technology, continuously integrating innovative components as they become available.

4.4 Hybrid Deployment and Bare Metal Automation

Sylva's **hybrid deployment strategy**, embedded within the framework of the distributed cloud, accommodates a wide spectrum of use cases, ranging from leveraging existing cloud infrastructure, such as VMWare or OpenStack, to implementing Container-as-a-Service (CaaS) on Bare Metal. This versatility in deployment architecture renders Sylva highly adaptable, catering to the specific needs and constraints of different scenarios.

The **deployment model selection** is crucial and is typically contingent on several factors, including the nature of the workload, the desired level of isolation, security concerns, cost-effectiveness, performance requirements, and the presence (or absence) of an existing cloud infrastructure.

Virtualized Environments

In scenarios where a robust cloud infrastructure is already in place, Sylva can readily integrate with prevalent **virtualized environments** such as VMware or OpenStack. This integration offers several benefits. For one, it leverages the abstraction and isolation capabilities inherent in virtualization, which can be beneficial for multi-tenant environments or applications with stringent security requirements. Additionally, the elasticity of such environments can be a boon for workloads with significant variability, enabling dynamic resource scaling to match the application's demand.

Bare Metal Environments

On the other hand, for use cases that demand maximum performance, such as high-performance computing (HPC) or big data applications, Sylva supports Container-as-a-Service (CaaS) on Bare Metal deployments. A **bare metal environment** eschews the overhead of virtualization, instead running applications directly on the server hardware. This approach can yield substantial performance gains, particularly for I/O intensive or latency-sensitive applications. Moreover, with the advent of containerization and orchestration platforms like Kubernetes, many of the benefits of virtualization, such as resource isolation and management, can be achieved in a bare metal context.

Moreover, Sylva utilizes **automation** to streamline the deployment and management of applications across these diverse environments. Automation in the context of bare metal refers to the process of automatically provisioning and configuring physical servers, eliminating the manual intervention traditionally associated with these tasks. The automation tooling then works to bring the actual state of the

infrastructure in line with this desired state, creating a highly repeatable and reliable deployment process.

In summary, Sylva's hybrid deployment strategy for distributed cloud environments embraces the flexibility and benefits of different models. Its integration capabilities with existing cloud infrastructures, the performance advantages offered by bare metal deployments, and the efficiency of automation together contribute to a versatile, efficient, and powerful platform.

Chapter 5

Network Edge Automation for Bare Metal Infrastructure

In the realm of **distributed multi-cluster environments**, managing virtualized devices has become a routine task. This model combines the benefits of both virtualized and physical infrastructure, allowing organizations to leverage the scalability and flexibility of virtual machines (VMs) while also incorporating the performance advantages and direct hardware access provided by bare metal nodes. In such environments, the configuration of VMs can be relatively straightforward, as they can be easily provisioned, migrated, and managed across different clusters. However, when it comes to bare metal nodes, which are physical servers tightly coupled to their hardware, the complexity increases.

Unlike virtual machines, physical servers lack the same level of flexibility and portability. They are inherently tied to their specific hardware components, including network interfaces, storage devices, and other physical resources. This divergence necessitates special attention to factors such as server location, connectivity, and hardware configuration when dealing with bare metal infrastructure.

When managing a distributed multi-cluster environment, the focus often lies on efficiently configuring and orchestrating virtualized devices within the clusters. Virtual machines can be created, cloned, and scaled easily, benefiting from the abstraction layer provided by the hypervisor. The management plane can effectively handle these tasks, abstracting away the underlying hardware details.

However, the challenge arises when attempting to automate the configuration and management of **bare metal devices** that are part of the network edge. These devices, such as network switches, routers, and load balancers, play a critical role in connecting the cluster to the external network and providing network services.

Unlike virtual machines, these devices are not as easily provisioned or manipulated. They require manual configuration and physical access, making them a bottleneck in the automation process.

To address this challenge, an architecture is needed that can automate the configuration and management of bare metal devices at the network edge. This architecture should seamlessly integrate with Kubernetes clusters, leveraging the existing automation and orchestration capabilities while extending them to the physical network infrastructure. By automating the provisioning, configuration, and monitoring of bare metal devices, organizations can achieve consistent and efficient network management across the entire hybrid environment.

In the following sections, we will explore an architecture specifically designed to tackle the manual configuration challenges of network edge devices connected to bare metal nodes within Kubernetes clusters. We will delve into the key components and mechanisms that enable network edge automation, highlighting the benefits and use cases of this approach.

5.1 Challenges

In the world of distributed cloud computing, orchestration of virtual machines and containers is a common task, thanks to the abstraction layer that these technologies provide. The task becomes more complex when we have to deal with physical resources, i.e., the bare metal servers, in an edge computing environment. Unlike virtual machines and containers, these servers are inseparable from their hardware, which introduces challenges related to server location, connectivity, and hardware configuration.

5.1.1 The Need for Automation at the Edge

One particular area that has been problematic is the configuration of network switches connecting to the bare metal nodes in an edge network. These switches require detailed setup for aspects like network interfaces, VLANs, routing protocols, and security settings. The configuration process is complex, labor-intensive, and prone to human error. It often involves repetitive tasks, requiring a high level of expertise in network administration and a detailed understanding of the network architecture.

The Role of Edge Networks

Edge networks play a crucial role in distributed cloud computing, primarily in reducing latency and ensuring quick data processing. They are located closer to

the end-users and often consist of multiple bare metal nodes. Each of these nodes is connected to a network switch, requiring individual configurations.

5.2 Multi-distributed Cloud Ecosystem

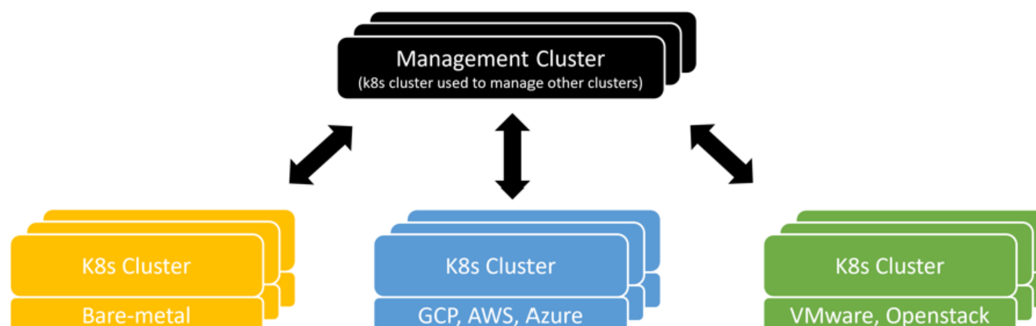


Figure 5.1: Multi-distributed cloud ecosystem

Given the complexity and diversity of resources in an edge computing environment, managing such systems effectively poses a significant challenge. These environments often contain multiple clusters, which could be distributed across different geographic locations and various infrastructures. These clusters could be cloud-based or run on bare metal environments, and each comes with its unique configuration and management requirements.

5.2.1 Management Cluster

To handle such complexity, we often employ a management cluster that acts as a control center. It coordinates and oversees the operations of all the workload clusters, including those in the edge sites. Each of these workload clusters, regardless of its underlying infrastructure, is under the control of the overarching management cluster.

In the context of the edge network scenario, a single edge site composed of bare metal nodes and network switches becomes a single workload cluster within the larger ecosystem. Therefore, we need a system within the management cluster that allows for remote and autonomous management and configuration of the edge devices. This leads us to the concept of Kubernetes operators, a solution that integrates seamlessly with the existing workflows within the management cluster.

5.3 Architecture Proposal

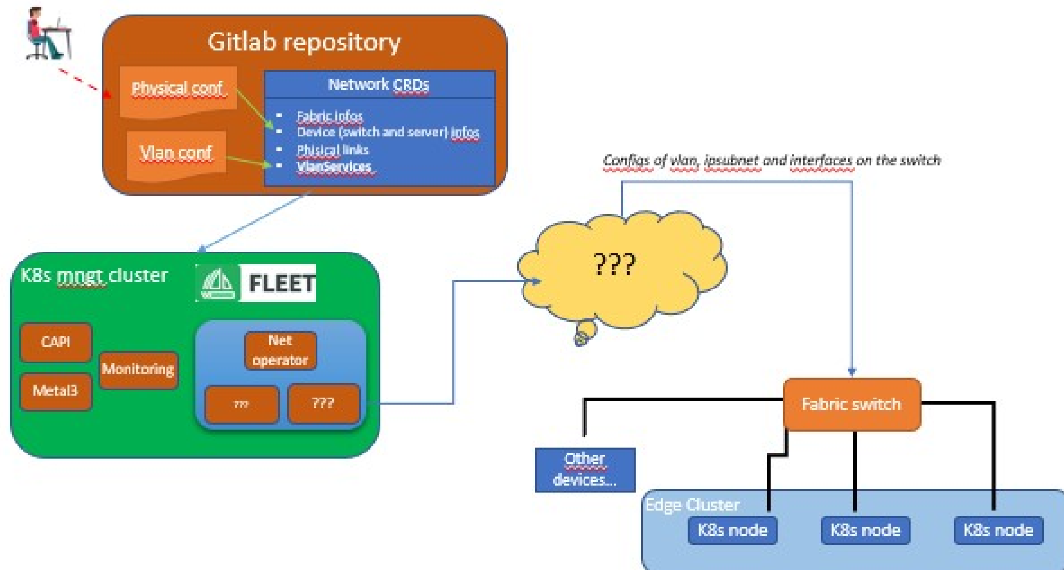


Figure 5.2: General architecture

In order to tackle the complexities inherent to managing the network configuration of bare metal infrastructure at the edge, a solution has been designed that leverages the capabilities of Kubernetes' Custom Resource Definitions (CRDs) and Kubernetes Operators.

CRDs provide a mechanism for defining custom resources, enabling the integration of arbitrary configurations and states into Kubernetes' declarative management approach. On the other hand, Kubernetes Operators, which are custom software extensions to Kubernetes, encapsulate operational knowledge about specific applications or infrastructure components, and can automate complex tasks related to the management of their associated resources. Together, these two mechanisms form the cornerstone of the proposed architecture, allowing a comprehensive and flexible solution for network configuration management in line with Sylva's principles.

The architecture encompasses two pivotal components: the Network Operator and the Actuator Operators.

The Network Operator, housed within the management cluster, is charged with comprehending the complex fabric of the network topology. It observes, maintains, and communicates the network's desired state to the broader Kubernetes ecosystem, inclusive of the necessary configurations for the network switches attached to bare metal nodes. This operator is envisaged as a constant in the ecosystem, serving as a consistent source of truth for the current network topology.

Conversely, the Actuator Operators are designed to be flexible and interchangeable, catering to the need for varying operational capabilities across different devices. Each Actuator Operator is responsible for actualizing the configurations on the physical devices. It does this by interpreting the desired state relayed by the Network Operator and converting it into specific actions on the network devices.

An example of such an Actuator Operator is one capable of interfacing with Cisco's Network Services Orchestrator (NSO). This particular operator would interpret the desired network state from the Network Operator, transcribe it into a format recognized by NSO, and then communicate this information to NSO, prompting the necessary changes on the relevant devices.

This amalgamation of a standardized, constant Network Operator along with a diverse set of interchangeable Actuator Operators constitutes a robust, scalable, and adaptable solution for network configuration automation in hybrid deployment scenarios.

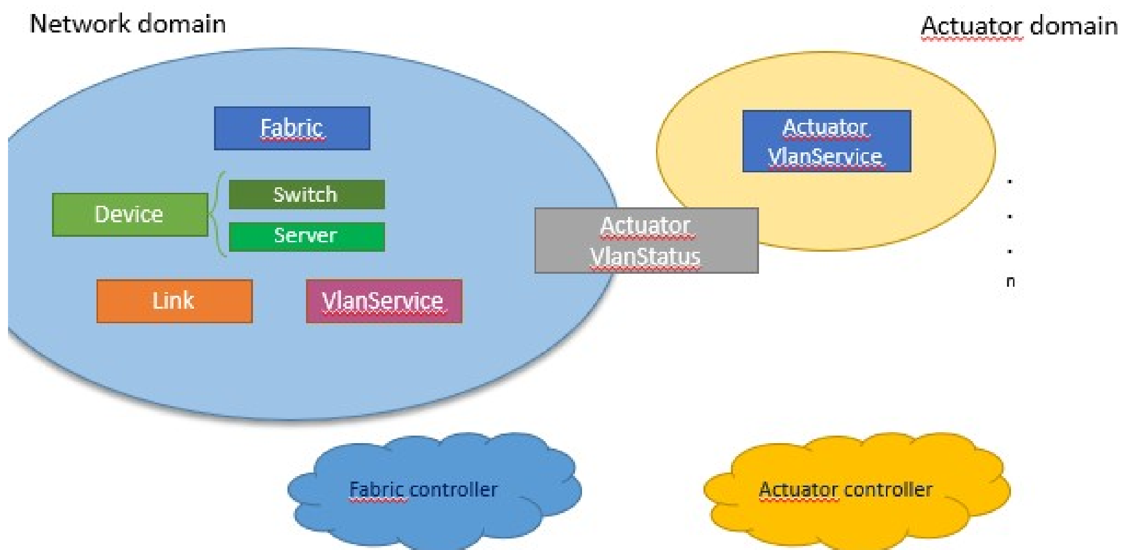


Figure 5.3: Proposed architecture

5.3.1 Network Operator

The *Network Operator* serves as the foundational pillar of the proposed architecture, assuming the responsibility of defining and maintaining the network topology's desired state. To achieve this, it leverages an array of specifically tailored CRDs, each focusing on an essential aspect of the network configuration:

- **Fabric:** The fabric, in the context of network topology, encapsulates the collective entities associated with a specific site. Fabric CRD captures these elements, identifying the actuator in control, and detailing the devices under its management, including the types and numbers of switches and servers.
- **Device:** Distinguishing the devices in the fabric is crucial for effective configuration management. The Device CRD, which is subdivided into Switch and Server CRDs, facilitates this identification. Each of these sub-CRDs delineates individual devices, describing their available interfaces and their unique MAC addresses.
- **Link:** This CRD captures the physical point-to-point connections between the devices. It outlines the device-port pairs, providing a map of how different devices are interconnected.
- **VlanService:** As the name suggests, this CRD outlines the intended VLAN configurations, which include the VLAN identifier, IP subnet, and the device-port where the VLAN should be set up. A status field is embedded in this CRD, providing real-time information about the VLAN configuration's current state.

These CRDs collectively provide the Network Operator with a detailed and dynamic understanding of the network. This knowledge is invaluable as it allows the Network Operator to consistently maintain the desired network state and promptly respond to any changes or disruptions.

5.3.2 Actuator Operators

While the Network Operator handles the design of the network topology, the *Actuator Operators* step in to translate this design into actionable configurations on the physical devices.

The proposed architecture envisions the development of various Actuator Operators, each customized to communicate with different network orchestrators, like Cisco NSO, thus establishing a flexible interface to the physical network devices. To achieve this, a specialized set of CRDs and corresponding controllers are developed for each Actuator Operator.

In the case of an NSO-focused operator, for instance, the system encompasses a custom resource for capturing the current configuration state of the network devices. In tandem with this, a controller supervises the interactions with NSO, overseeing tasks like configuration management, status monitoring, and synchronization between the desired and actual states.

The integration of multiple Actuator Operators augments the adaptability of the system, facilitating the support of various orchestrators and thereby catering to

a broader range of networking environments. This arrangement nurtures an open, scalable system, reducing reliance on a single orchestrator and hence diminishing the risk of a single point of failure.

Actuator Vlan Status

In order to ensure smooth communication between the Network and Actuator domains, a shared resource '*ActuatorVlanStatus*' has been introduced. This resource allows Actuator Operators to communicate the configuration status back to the Network Operator, indicating whether the intended changes have been correctly implemented, or if there were any issues. This ensures that the Network Operator can maintain an accurate view of the network's state, even when configuration tasks are delegated to the Actuator Operators.

Chapter 6

Implementation of the Proposed Architecture

This chapter delves into the practical implementation of the previously described architectural blueprint. We will discuss the programming languages used, the approach adopted for constructing Kubernetes Operators and CRDs, and how we have leveraged the toolset provided by Kubernetes to bring our design into fruition.

6.1 Building Blocks: Operators and CRDs

At the heart of our design are Kubernetes Operators and Custom Resource Definitions (CRDs). These two elements form the foundation upon which we have built our architecture.

6.1.1 Kubernetes Operators

Kubernetes Operators are software extensions to the Kubernetes API that enable automated creation, configuration, and management of complex applications. They build upon the fundamental Kubernetes concept of a controller, adding domain-specific knowledge to create a custom control loop that can manage a specific application or service. Operators effectively encode operational knowledge into software, automating tasks that traditionally required manual intervention by a human operator.

In our system, we have utilized this powerful concept to create two operators: the Network Operator and the Actuator Operator.

The Network Operator

The Network Operator presently takes charge of managing resources that depict the desired network topology and interconnectivity. This process leans heavily on the GitOps approach, where the network topology information is supplied by the network administrator.

In the GitOps approach, infrastructure management takes place through a version control system, predominantly Git. The network administrator leverages this system to create and meticulously describe various resources, which encapsulate the necessary information about the connections between servers and switches.

This method provides a clear, version-controlled history of changes, enhancing predictability and accountability. Moreover, the it enables easy rollbacks to previous configurations if necessary, supporting error mitigation and system stability.

The operator watches these resources—Fabric, Device, Link, and VlanService—for any changes, reacting accordingly. It orchestrates the necessary actions to reconcile the current network state with the desired state defined by these resources. One of its key tasks is to update the status of the VlanService resource based on the outcome of network configuration operations, which is reported by the corresponding Actuator Operator through the ActuatorVlanStatus resource.

In the future, as we will discuss in Chapter 8, the Network Operator will have an enhanced intelligence. It will be able to construct the entire network topology by interfacing with other components within the management cluster, eliminating the need for manual description of resources by the network administrator. This capability will further streamline the management of network services and resources, making the operator an even more central component in our proposed architecture.

The Actuator Operator

The Actuator Operator, on the other hand, is primarily responsible for overseeing the process of network configuration execution on a specific fabric. For each fabric, a specific actuator is defined (for example Cisco NSO). The operator watches for changes on VlanService resources related to its actuator and acts to reconcile the actual network configuration with the desired one, communicating directly with the desired network orchestrator.

When the Actuator Operator recognizes a change in a VlanService resource, it creates an internal resource representing the desired VLAN configuration and sends this configuration to the orchestrator that have to enacts the requested changes on the relevant network devices. The Actuator Operator waits for the outcome of this operation and updates the ActuatorVlanStatus resource with the result. This status resource provides a shared state that can be watched by other components of the system, such as the Network Operator, allowing it to update the status of the original VlanService resource.

This structure and behavior of the Actuator Operator illustrate how Kubernetes Operators can offload complex operational tasks, freeing network administrators to focus on defining the desired network services and topology. The Operator takes care of ensuring that the actual network state matches this desired state, handling any errors or issues that may arise in the process.

6.1.2 Custom Resource Definitions (CRDs)

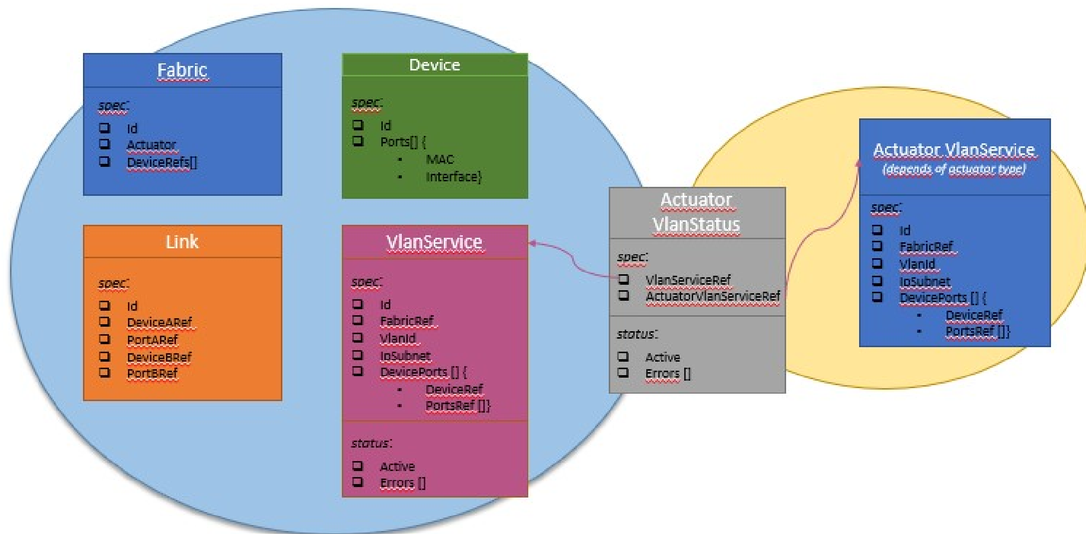


Figure 6.1: CRDs implementation

In Kubernetes, a Custom Resource Definition (CRD) is a way of defining our own object kinds, letting the Kubernetes API server handle the entire lifecycle. These CRDs essentially extend the Kubernetes API, providing new endpoints that the Kubernetes API server will handle.

The primary advantage of using CRDs is that they allow us to define our own, domain-specific objects that behave much like the built-in Kubernetes objects. This means that we can use kubectl and other Kubernetes tools to manage these resources, and we can implement custom controllers to automate their behavior.

Network operator CRDs

In the network operator, we defined four principal CRDs:

- **Fabric**: This CRD is used to represent a network fabric. It includes specifications about the network devices and the actuator configured to manage these devices. It carries information like the name of the fabric, the actuator that's used to manage it, and a list of devices that form the fabric.

```yaml
---
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.10.0
  creationTimestamp: null
  name: fabrics.fabric-operator.tim.k8s.teo
spec:
  group: fabric-operator.tim.k8s.teo
  names:
    kind: Fabric
    listKind: FabricList
    plural: fabrics
    singular: fabric
  scope: Namespaced
  versions:
  - name: v1
    schema:
      openAPIV3Schema:
        description: Fabric is the Schema for the fabrics API
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned
   schema of this representation
              of an object.
            type: string
          kind:
            type: string
          metadata:
            type: object
          spec:
            description: FabricSpec defines the desired state
   of Fabric
            properties:
              actuator:
                type: string
              devices:
                items:
                  properties:
                    name:
                      type: string
                  type: object
                type: array
              id
                type: string
            type: object
          status:
```

```
            description: FabricStatus defines the observed
  state of Fabric
            type: object
        type: object
    served: true
    storage: true
    subresources:
      status: {}
```

Listing 6.1: Fabric base CRD

- **Device**: This CRD is used to represent a network device within a Fabric. The Device resource allows us to specify detailed device information, such as device interfaces and their MAC addresses. It can be a switch or a physical server also

```yaml
apiVersion: fabric-operator.tim.k8s.teo/v1
kind: Device
metadata:
  labels:
    app.kubernetes.io/name: device
    app.kubernetes.io/instance: device-sample
    app.kubernetes.io/part-of: fabric-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: fabric-operator
  name: clarinet4-leaf2
  namespace : clarinet4
spec:
  id : Clarinet4-leaf2
  fabric :
    - name :  clarinet4
  ports:
    - mac : "00:1b:63:84:45:e6"
      interface: "ethernet1/1/16"
    - mac: "00:1b:63:84:45:e7"
      interface: "ethernet1/1/10:1"
    - mac: "b0:4f:13:7c:38:b4"
      interface: "ethernet1/1/5:4"
    - mac: "b0:4f:13:7c:38:c1"
      interface: "ethernet1/1/9:1"
    - mac: "b0:4f:13:7c:38:c2"
      interface: "ethernet1/1/9:2"
```

Listing 6.2: Device example resource

- **Link**: This CRD represents a link within the network. The resource describes the network connectivity between devices, providing detailed information about the

link endpoints. It contains information such as the name of the link, the devices it connects, and the specific interfaces involved in the link.

- **VlanService**: This CRD is used to represent a VLAN service within the network. It provides a declarative interface to describe the desired network configuration for a particular VLAN service. The VlanService resource includes the VLAN ID, the fabric on which it operates, and a list of devices and their interfaces involved in the VLAN.

```yaml
apiVersion: fabric-operator.tim.k8s.teo/v1
kind: VlanService
metadata:
  labels:
    app.kubernetes.io/name: vlanservice
    app.kubernetes.io/instance: vlanservice-sample
    app.kubernetes.io/part-of: fabric-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: fabric-operator
  name: vlan100
  namespace : clarinet4
spec:
  id : vlan100
  fabric : clarinet4
  vlanid : "100"
  ipsubnet: "163.162.196.33/29"
  devices-ports:
    - device : clarinet4-leaf2
      ports:
        - "b0:4f:13:7c:38:b4"
        - "b0:4f:13:7c:38:c1"
        - "b0:4f:13:7c:38:c2"
```

Listing 6.3: VlanService example resource

Actuator Operator CRDs

The creation of Custom Resource Definitions (CRDs) for the Actuator Operator is dictated by the unique requirements of each specific actuator. Taking NSO (Network Service Orchestrator) as an instance, the CRD crafted mirrors the Network Operator's VlanService. It provides a description of the current network configuration of the switch, detailing particulars such as which VLAN is associated with which devices and interfaces.

ActuatorVlanStatus CRD

The ActuatorVlanStatus, on the other hand, represents a shared resource that spans across the realms of the Actuator and Network Operator. Essentially, it is a simplistic CRD where the specifications incorporate a reference to the VlanService of the Network Operator and to the CRD instantiated by the actuator.

Within its status field, it carries the response resulting from the actuator's API calls. Consequently, it provides insights about the state of the network configuration – whether everything is appropriately configured, or if there have been errors on any of the devices, for example.

Here an example of ActuatorVlanStatus created by the NSO Actuator Operator:


```
apiVersion: nso-operator.tim.k8s.teo/v1
kind: ActuatorVlanStatus
metadata:
  creationTimestamp: "2023-07-07T10:20:45Z"
  finalizers:
  - fabric-operator.tim.k8s.teo/finalizer
  generation: 1
  name: status-100
  namespace: clarinet4
  ownerReferences:
  - apiVersion: nso-operator.tim.k8s.teo/v1
    blockOwnerDeletion: true
    controller: true
    kind: NSOVlanService
    name: nso-100
    uid: 13840b22-7d7c-4bc5-969b-b725541fd70d
  resourceVersion: "118370149"
  uid: b23ba3b9-3746-4433-b769-07b7e7e32106
spec:
  actuator_vlanservice_id: nso-100
  vlanservice_id: vlan100
status:
  active: true
  errors: []
```


Listing 6.4: ActuatorVlanStatus example CR created by NSO operator

6.2 Implementing them with Kubebuilder

The creation of Kubernetes Operators and CRDs was facilitated by Kubebuilder, a software development kit (SDK) that simplifies the process of building the foundational aspects of our architecture.

Kubebuilder is an open-source tool developed by the Kubernetes SIG API Machinery group. It provides a framework for building Kubernetes APIs and comes with tools for building the binaries to run your application.
To start our project, we initialized a new Kubebuilder project with the command:

```
kubebuilder init --domain your.domain.com --repo github.com/your/repo
```

This command creates the basic project layout, initializes Go modules for dependency management, and generates a Makefile for build tasks.
We then selected Go as our primary programming language due to its wide usage in cloud-native applications, superior performance, strong static typing, and native concurrency support.

6.2.1 Creating the CRDs and controller

With the project initialized, we proceeded to create our CRDs. To define a new API kind, we used the Kubebuilder command:

```
kubebuilder create api --group network --version v1 --kind Fabric
kubebuilder create api --group network --version v1 --kind Device
kubebuilder create api --group network --version v1 --kind Link
```

These commands scaffolded out the Go code for the APIs. They created
api/v1/fabrictypes.go,
api/v1/devicetypes.go,
api/v1/linktypes.go, and
where we could define the Spec and Status of our CRDs. If we want to create a controller also for that CRD we can add the flag –controller=true

```
kubebuilder create api --group network --version v1 --kind
    VlanService --controller=true
```

This command generates a new file at controllers/vlanservicecontroller.go. This file contains a Reconcile method, which is called whenever an instance of the VlanService resource changes. Inside this Reconcile method is where we put our control logic. Each controller effectively becomes a domain-specific control loop that runs against the Kubernetes API server.

6.2.2 Defining the CRD Spec and Status

In the types.go files, we defined the Spec and Status for each of our CRDs. The Spec describes the desired state of the resource, and the Status reflects the observed

state. For instance, in fabrictypes.go, we have:

```
type FabricSpec struct {
// Specification of the Fabric
Actuator string json:"actuator"
Devices []DeviceReference json:"devices"
}

type FabricStatus struct {
// Observed state of the Fabric
Conditions []metav1.Condition json:"conditions,omitempty"
}
```

Once we finished defining our CRDs, we used the make command to generate the CRD manifests:

```
make manifests
```

This command generated the YAML manifests for our CRDs in the config/crd/bases directory.

6.2.3 Implementing the Controllers

Once the Custom Resource Definitions (CRDs) are defined, it is necessary to implement the controllers for these resources. In Kubernetes, controllers watch resources and act upon changes. In other words, they contain the logic needed to reconcile the current state of the resource to its desired state.
In general, a typical reconcile loop for a controller involves the following steps:

- Observe the state of the system and the desired state.
- Compare the current state with the desired state.
- Compute the operations necessary to make the current state match the desired state.
- Execute the necessary operations.
- In code, this might look like the following (in a very simplified form):

```
func (r *Reconciler) Reconcile(req ctrl.Request) (ctrl.Result, error)
    {
```

```
2 // Fetch the current state of the resource
3 currentResource := &v1.MyResource{}
4 if err := r.Get(context.Background(), req.NamespacedName,
     currentResource); err != nil {
5 log.Error(err, "unable to fetch MyResource")
6 return ctrl.Result{}, client.IgnoreNotFound(err)
7 }// Compute the desired state
8 desiredResource := currentResource.DeepCopy()
9 // modify desiredResource to reflect desired state
10
11 // Compare and reconcile
12 if !reflect.DeepEqual(currentResource.Spec, desiredResource.Spec) {
13     if err := r.Update(context.Background(), desiredResource); err !=
        nil {
14         log.Error(err, "unable to update MyResource")
15         return ctrl.Result{}, err
16     }
17 }
18
19 return ctrl.Result{}, nil
20 }
```

In conclusion, Kubebuilder provided us with a powerful and flexible toolkit for building our Kubernetes Operators and CRDs, simplifying the development process and allowing us to focus on the specific logic of our network management system.

6.3 Actuator Operator for NSO

As an illustration of an Actuator Operator, we developed one that is compatible with Cisco Network Services Orchestrator (NSO). NSO, a product of Cisco, is known for its robust industry-standard capabilities in the management and orchestration of network services. Its model-driven approach, flexibility, and wide-ranging compatibility with various network devices make it an excellent candidate for our network orchestration platform.
The two principal roles of this operator, similar to those described earlier, are resource management and NSO communication.

6.3.1 Resource Management

The actuator operator reconciles *VlanService* resources, which represent the desired configurations on devices, and manages *NSOVlanService* resources that depict the current configurations on these devices.
When a *VlanService* resource belonging to NSO's domain, i.e., within a fabric with NSO specified as the "actuator," is created, modified, or deleted, the reconciling

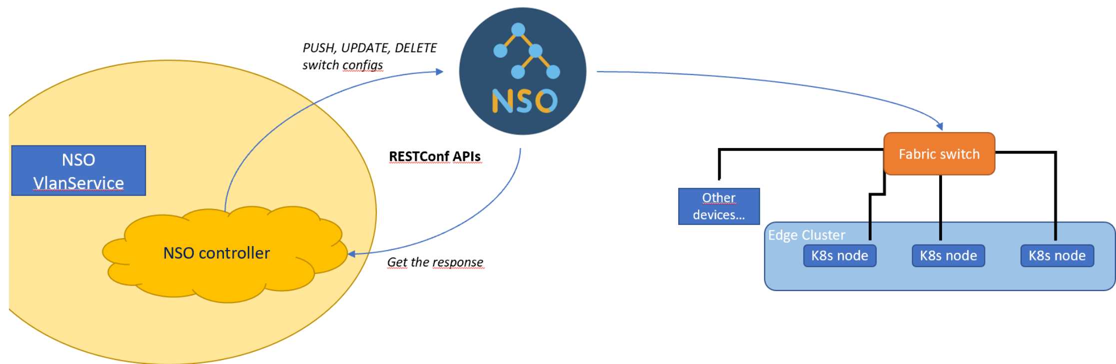


Figure 6.2: NSO interactions

controller intercepts the resource and undertakes necessary operations to reflect these changes on the devices.
The controller reconciles the *VlanService* resource and upon acquiring it, inspects the VLAN ID specification. It then checks whether there is already an existing *NSOVlanService* resource pertaining to that VLAN.
If this resource already exists, it implies that the desired VLAN is already configured on some devices in NSO. The controller must then compare the current configuration with the desired one to discern the actual devices on which the VLAN needs to be created, deleted, or differently configured (change of ports/interfaces).
Conversely, if the corresponding *NSOVlanService* resource does not exist, it signifies the configuration of a new VLAN. The controller must then proceed with the creation and configuration operations on the appropriate devices.

6.3.2 Communication with NSO

These operations are relayed to NSO, which facilitates direct communication with the devices. To this end, various RESTCONF APIs have been implemented, which the controller calls to execute the creation, modification, and deletion of configurations.
Based on the response from these API calls, the controller determines whether the operation was successful. It must then report this status within the *ActuatorVlanStatus* resource. This resource acts as a communication conduit between the Network Operator and Actuator domains and serves to report the status of executed operations. It delineates whether the desired configuration has been correctly implemented, or if there were any errors, providing comprehensive details thereof.

6.3.3 RESTCONF API Calls

getDeviceConfig

Get device configuration.

```
1 {
2   "method": "GET",
3   "url": "/device-config?device={device}",
4   "headers": {
5     "Accept": "application/yang-data+json"
6   }
7 }
```

Listing 6.5: getDeviceConfig

getVlan

Get the VLAN with the specified ID for the given device.

```
1 {
2   "method": "GET",
3   "url": "/vlan/{vlanId}/{device}",
4   "headers": {
5     "Accept": "application/yang-data+json"
6   }
7 }
```

Listing 6.6: getVlan

getVlanConfiguration

Get the VLAN configuration with the specified ID for the given device.

```
1 {
2   "method": "GET",
3   "url": "/vlan-configuration/{vlanId}/{device}",
4   "headers": {
5     "Accept": "application/yang-data+json"
6   }
7 }
```

Listing 6.7: getVlanConfiguration

createVlan

Create a new VLAN.

```
{
  "method": "POST",
  "url": "/create-vlan",
  "headers": {
    "Content-Type": "application/yang-data+json"
  },
  "body": {
    "vlanId": "{vlanId}",
    "device": "{device}"
  }
}
```

Listing 6.8: createVlan

createVlanConfiguration

Create a new VLAN configuration.

```
{
  "method": "POST",
  "url": "/create-vlan-configuration",
  "headers": {
    "Content-Type": "application/yang-data+json"
  },
  "body": {
    "vlanId": "{vlanId}",
    "device": "{device}",
    "ipAddress": "{ipAddress}"
  }
}
```

Listing 6.9: createVlanConfiguration

modifyVlanConfiguration

Modify the VLAN configuration with the specified ID for the given device.

```
{
  "method": "PATCH",
  "url": "/modify-vlan-configuration/{vlanId}/{device}",
  "headers": {
    "Content-Type": "application/yang-data+json"
  },
  "body": {
    "vlanId": "{vlanId}",
    "device": "{device}",
    "ipAddress": "{ipAddress}",
    "interfaces": ["interface1", "interface2"]
```

```
12   }
13 }
```

Listing 6.10: modifyVlanConfiguration

deleteVlan

Delete the VLAN with the specified ID for the given device.

```
1 {
2   "method": "DELETE",
3   "url": "/delete-vlan/{vlanId}/{device}",
4   "headers": {
5     "Accept": "application/yang-data+json"
6   }
7 }
```

Listing 6.11: deleteVlan

deleteVlanConfiguration

Delete the VLAN configuration with the specified ID for the given device.

```
1 {
2   "method": "DELETE",
3   "url": "/delete-vlan-configuration/{vlanId}/{device}",
4   "headers": {
5     "Accept": "application/yang-data+json"
6   }
7 }
```

Listing 6.12: deleteVlanConfiguration

6.3.4 Role Base Access Control integration

Resource management in the Actuator Operator for NSO involves handling various resources for both reading and modifying/writing purposes. To ensure secure access control, RBAC (Role-Based Access Control) is utilized in Kubernetes to define and manage permissions for different roles within the system.
RBAC in Kubernetes allows for the following key functionalities:

- Role: Defines a set of permissions associated with specific responsibilities or activities.
- Role Binding: Associates roles with users or groups, assigning permissions to specific entities.

- Rules: Specifies the permissions and restrictions for particular actions or operations.

In the context of the Actuator Operator, RBAC permissions can be integrated into the controller using the '//+kubebuilder:rbac' annotation. This annotation specifies the RBAC permissions for the resources in the controller.
To integrate RBAC permissions for resource management in the Actuator Operator's controller, add the following annotation before the controller definition:

```
kubebuilder:rbac:groups=nso-operator.tim.k8s.teo,resources=
    nsovlanservices,verbs=get;list;watch;create;update;patch;delete
```

In the annotation above, 'nsooperator.tim.k8s.teo' represents the API group, and 'nsovlanservices' represents the resource name. The specified verbs, such as 'get', 'list', 'watch', 'create', 'update', 'patch', and 'delete', define the allowed operations on the resource.
By adding this annotation, Kubebuilder will automatically generate the RBAC definitions when running the 'make manifests' command. This simplifies the RBAC management process within the Actuator Operator.
Ensure that you customize the annotation with the appropriate API group and resource name relevant to your use case.
By integrating RBAC permissions using the '//+kubebuilder:rbac' annotation, you can ensure secure and granular authorization management for the resources managed by the Actuator Operator.

Listing 6.13: RBAC into NsoController

```
kubebuilder:rbac:groups=fabric-operator.tim.k8s.teo,resources=
    vlanservices,verbs=get;list;watch;

kubebuilder:rbac:groups=nso-operator.tim.k8s.teo,resources=
    nsovlanservices,verbs=get;list;watch;create;update;patch;delete
kubebuilder:rbac:groups=nso-operator.tim.k8s.teo,resources=
    nsovlanservices/status,verbs=get;list;watch;create;update;patch;
    delete
kubebuilder:rbac:groups=nso-operator.tim.k8s.teo,resources=
    nsovlanservices/finalizers,verbs=update

kubebuilder:rbac:groups=nso-operator.tim.k8s.teo,resources=
    actuatorvlanstatus,verbs=get;list;watch;create;update;patch;delete
kubebuilder:rbac:groups=nso-operator.tim.k8s.teo,resources=
    actuatorvlanstatus/status,verbs=get;list;watch;create;update;patch
    ;delete
kubebuilder:rbac:groups=nso-operator.tim.k8s.teo,resources=
    actuatorvlanstatus/finalizers,verbs=update
```

6.4 Implementation Workflow

In this section, we detail the workflow of the system, covering three main use-cases: Creating a new VlanService resource, modifying an existing VlanService resource, and deleting a VlanService resource. We'll explore how the Actuator Operator and Network Operator work together to apply these changes and maintain the desired network configuration in our edge cluster.

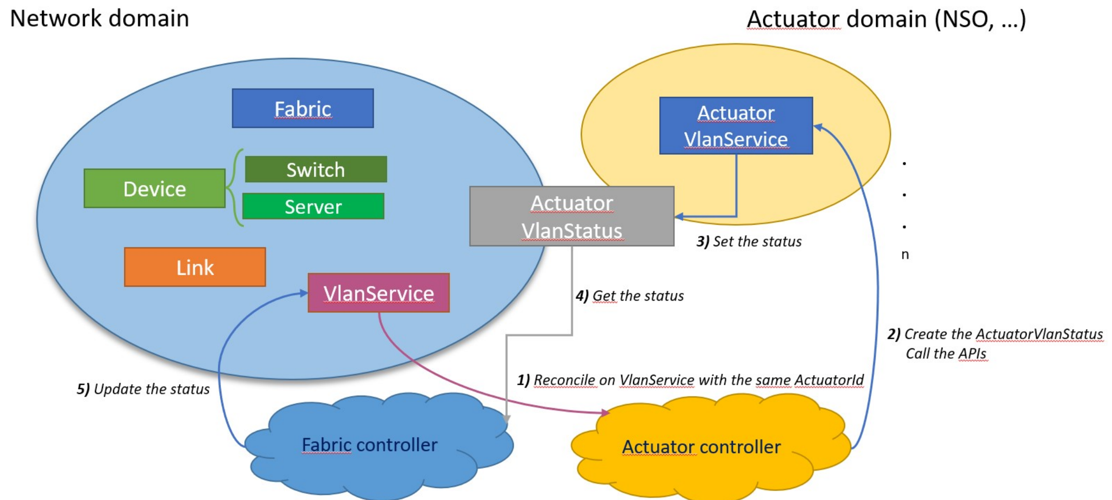


Figure 6.3: Workflow

6.4.1 Creating a New VlanService

In this scenario, we wish to establish a new VLAN within the network. This process starts with the application of a new VlanService custom resource (CR) in the cluster. The Actuator Operator, specifically the one corresponding to the Fabric (in our case, NSO), will reconcile this new CR.

Listing 6.14: Vlanservice NSO Riconciliation

```
1 {
2         //VlanService Reconciler
3     vlanservice := &fabricoperatorv1.VlanService{}
4     if err := r.Client.Get(ctx, req.NamespacedName, vlanservice);
      err != nil {
5         return ctrl.Result{}, client.IgnoreNotFound(err)
6     }
7
8         //Fabric associated to the VlanService
```

```
9       fabric := &fabricoperatorv1.Fabric{}
10      if err := r.Client.Get(ctx, client.ObjectKey{
11          Namespace: req.Namespace,
12          Name:      vlanservice.Spec.Fabric,
13      }, fabric); err != nil {
14          return ctrl.Result{}, client.IgnoreNotFound(err)
15      }
16
17          //Checks if the actuator of the Fabric correspond to NSO
18      if fabric.Spec.Actuator != "NSO" {
19          log.Info("----------Skipping reconciliation----------")
20          return ctrl.Result{}, nil
21      }
22    }
23  }
```

On encountering the new VlanService CR, the Actuator Operator analyses the specifications and makes two API calls to NSO:

- `createVlan` - A POST request to `/create-vlan` creates a new VLAN service in NSO for a specific device.
- `createVlanConfiguration` - A POST request to `/create-vlan-configuration` specifies the details of the newly created VLAN service, such as IP subnet and the MAC addresses of the interfaces for that device.

Upon successful completion of these operations, the controller creates an 'nso-vlanservice' CR reflecting the current (now effective) device configuration. If everything went as planned, this configuration matches the desired state defined in the original VlanService CR. Furthermore, an 'ActuatorVlanStatus' CR is created, which reports the status of the operation, such as whether the configuration was successful, if there were any errors, and the devices on which those errors occurred. Once the 'ActuatorVlanStatus' CR is created, the Network Operator reconciles it, inspecting the status and updating the original VlanService CR accordingly. This way, we can always know if the desired configuration was successfully applied.

6.4.2 Modifying a VlanService

In this scenario, we need to alter the existing configuration of a VlanService. This could include changes such as adding or removing VLAN from a device, modifying the interfaces on a device, changing the IP subnet, etc.
The Actuator Operator reconciling the VlanService resource checks for differences between the current configuration (as described by the 'nso-vlanservice' CR) and

the desired configuration (as described by the ‘vlanservice‘ CR). Based on these differences, the appropriate API calls to NSO are constructed:

- `modifyVlanConfiguration` - A PATCH request to `/modify-vlan-configuration/{vlanId}/{device}` alters the interfaces used by the various devices or the ipsubnet.
- `createVlan` and `createVlanConfiguration` - These requests are made if a VLAN needs to be added to a new device.
- `deleteVlan` - A DELETE request to `/delete-vlan/{vlanId}/{device}` removes the VLAN from a specific device.

Once the operations are complete, the Actuator Operator updates the ‘ActuatorVlanStatus‘ CR to reflect the results. The Network Operator then reconciles this resource, updating the status of the original VlanService CR accordingly.

6.4.3 Deleting a VlanService

If we want to remove a VLAN from all devices, we delete the corresponding VlanService CR. The appropriate Actuator Operator retrieves the information regarding the VLAN and the devices on which it was configured and sends a ‘deleteVlan‘ API call to NSO for each of them.
If these operations are successful, the corresponding ‘nso-vlanservice‘ and ‘ActuatorVlanStatus‘ CRs are deleted. This deletion is performed safely using Kubernetes' "finalizers".

Finalizers in Kubernetes

Finalizers are keys with a corresponding value that Kubernetes checks before a resource is deleted. They enable synchronous cleanup before the Kubernetes garbage collector deletes the resource. When a deletion request is made, the resource is marked as "terminating" and the deletion timestamp is set. However, the resource is not deleted from the etcd database until all finalizers are removed. In our context, finalizers are utilized to ensure the safe deletion of our custom resources. The controller, built with Kubebuilder, adds a finalizer to the custom resource. Upon a deletion request, it handles the deletion of the ‘nso-vlanservice‘ and ‘ActuatorVlanStatus‘ CRs and then removes the finalizer. Consequently, Kubernetes can safely delete the original VlanService CR.

Chapter 7

Simulation and Result

7.1 Benchmark Hardware Specifications

This section presents a detailed overview of the hardware setup used to conduct the benchmark tests.

Kubernetes Cluster

The cluster used for the deployment of the Custom Resource Definitions (CRDs) and the controller is composed of 4 Virtual Machines (VMs): one serving as the control plane and the other three as worker nodes. These VMs are hosted on a VSphere instance. The specifications of this setup are as follows:

- Kubernetes version: 1.22.8
- Ubuntu version: 20.04
- Number of CPUs per VM: 2
- Memory per VM: 8 GB
- Hard disk per VM: 50 GB

Switch

The network switch used in the setup is a Dell EMC PowerSwitch S5232F-ON. This is a multi-layer switch equipped with 32x 100 GbE QSFP28 ports and 2x 10 GbE SFP+ ports. In this case, they are utilized as 4x 10 GbE or 4x 25 GbE ports, employing breakout cables.

Servers

Three servers of the same model, all connected through the VLAN, form part of the network setup. These are Dell PowerEdge R640 servers, each with the following specifications:

- CPU: Intel(R) Xeon(R) Gold 6252N, operating at 2.30 GHz, with 24 cores
- Storage: 2x 1.2 TB SAS 10k
- RAM: 384 GB
- Network Interfaces: 4x 10GbE and 2x 25GbE

These configurations provide the underlying platform for executing the benchmark tests and evaluating the performance of our Kubernetes operators.

7.2 Benchmark Results

The benchmark tests were designed to analyze the elapsed time between the creation/modification/deletion of a VlanService CR and the moment the status of the configuration operations is reported back on this CR. In other words, this signifies the time taken to set the configuration on the devices and save their status. The tests were structured to showcase the actual time taken by one operator (the Network Operator) in comparison to the other (NSO Actuator, in this case).
Table 7.1 showcases the timing results. The *"Total"* column represents the overall time taken from CR modification to status report, while the *"Actuator Operator"* column indicates the time consumed by the NSO Actuator to execute its tasks. The *"Network Operator"* column shows the time spent by the Network Operator.

	Actuator Operator	**Network Operator**	**Total**
Vlan deleted	1450 ms	150 ms	1600 ms
Vlan updated	2290 ms	510 ms	2800 ms
Vlan created	4174 ms	70 ms	4244 ms

Table 7.1: Operators response time

NSO APIs response time

In the subsequent section of the benchmark tests, the response time of the various NSO API calls was analyzed. The aim was to understand how much time the Actuator Operator spent communicating with NSO.

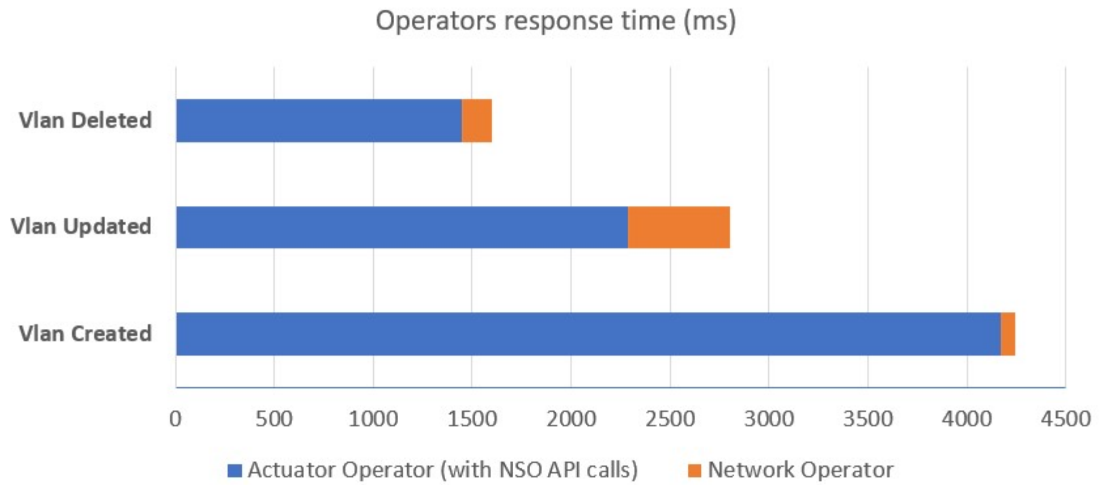


Table 7.1 represents the average response time for various NSO RESTConf API operations.

	Response time
POST vlan	1005 ms
POST vlanConfiguration	1010 ms
GET vlan	300 ms
PATCH vlanConfiguration	580 ms
DELETE vlan	650ms

Figure 7.1: NSO API calls response time

These results offer valuable insights into the performance of the Network and Actuator operators, as well as the response efficiency of NSO's RESTConf APIs in the given setup.

7.3 Conclusion

The benchmarking presented in this chapter provides preliminary insight into the performance of the Network and Actuator Operators. It specifically addresses the time taken for key operations, such as the creation, modification, and deletion of a VlanService CR, and the time required for setting and updating configurations on the devices.

As demonstrated in Table 7.1, the Network Operator's operations take a small fraction of the total response time, indicating its relative efficiency. Moreover, the analysis of the NSO RESTConf API response times, summarized in Figure 7.1, helped quantify the communication delay between the Actuator Operator and NSO, with results showing relatively efficient interaction between the two.
However, it is crucial to understand that the findings from these benchmark tests are intimately tied to the specific hardware used during the testing phase. The computational capabilities and characteristics of the servers, virtual machines, and network devices directly influence these performance outcomes. In cloud environments, the performance of a system is often tightly coupled with the capabilities of the nodes in use. Therefore, the observed performance characteristics might significantly vary if the system was to be deployed on different hardware or within a different cloud infrastructure.
This realization underscores the importance of understanding the hardware-dependency of such benchmarks. While they offer valuable insights into the performance of our system in the specific testing environment, they might not fully represent the system's performance in a different context. This implies that any potential scale-up or migration to a different hardware or cloud environment would necessitate a new series of benchmarks to accurately evaluate and optimize the system's performance in the new setting.

Chapter 8

Future Perspectives

In this chapter, we explore the future prospects of the thesis topic, considering the integration of ClusterAPI (CAPI) and Metal3 components and their interaction with the intelligent network operator. We envision the network operator becoming more intelligent, capable of recognizing the entire network topology by leveraging the functionalities provided by CAPI and Metal3. This integration is crucial in the context of an edge network cluster composed of bare metal nodes, managed within a larger ecosystem where a management cluster creates and oversees various workloads.

8.1 Scenario Overview

The scenario involves a site consisting of an edge network cluster comprised of bare metal nodes. This cluster serves as a workload cluster within a larger ecosystem where the management cluster, already integrated with Metal3 and ClusterAPI components, creates and manages various workloads.

8.2 ClusterAPI

ClusterAPI is an open-source Kubernetes project that provides declarative APIs and tools for creating, configuring, and managing Kubernetes clusters. Its primary goal is to simplify cluster lifecycle management and enable infrastructure providers to offer Kubernetes as a service. [12]

8.2.1 Key Components

ClusterAPI is composed of several key components:

- Cluster API Provider: This component extends the Cluster API to manage clusters on a specific infrastructure platform. It provides provider-specific implementations to interact with underlying infrastructure resources.
- Cluster Controller: The Cluster Controller manages the lifecycle of cluster resources. It handles operations such as creating, scaling, and deleting clusters, ensuring that the desired state of the cluster is maintained.
- Bootstrap Provider: The Bootstrap Provider handles the initialization and bootstrapping of nodes within the cluster. It provisions necessary resources and configures the cluster nodes to join the desired Kubernetes cluster.
- Infrastructure Provider: The Infrastructure Provider interacts with the underlying infrastructure to provision and manage resources required for the cluster. This includes virtual machines or bare metal servers, depending on the infrastructure being used.

8.2.2 Deploying a cluster

The typical workflow for deploying a cluster using CAPI involves the following steps: [13]

1. Infrastructure Provisioning

The first step in deploying a cluster is provisioning the necessary infrastructure. This typically involves creating virtual machines or bare metal servers that will serve as the worker nodes for the cluster. ClusterAPI provides infrastructure providers that interface with the underlying infrastructure to create and manage these resources. The infrastructure provider takes care of provisioning the required virtual machines or bare metal servers based on the cluster configuration.

2. Cluster Configuration

Once the infrastructure is provisioned, the next step is to configure the cluster itself. This includes specifying parameters such as the number of control plane nodes, worker node specifications, networking details, and any additional customizations required for the cluster. The cluster configuration is typically defined using YAML manifests or custom resources provided by ClusterAPI.

3. Bootstrap Process

After the cluster configuration is defined, the bootstrap process begins. The bootstrap provider is responsible for initializing and bootstrapping the control

plane nodes of the cluster. It provisions the necessary resources and configures the control plane nodes to join the desired Kubernetes cluster. This process ensures that the control plane is properly set up and ready to manage the cluster.

4. Worker Node Joining

Once the control plane nodes are bootstrapped, the worker nodes need to join the cluster. The worker nodes, whether virtual machines or bare metal servers, need to be configured with the necessary Kubernetes components to function as part of the cluster. ClusterAPI handles the process of joining the worker nodes to the cluster, ensuring they are properly configured and able to communicate with the control plane.

5. Cluster Validation and Health Checks

After the worker nodes join the cluster, ClusterAPI performs validation and health checks to ensure the cluster is functioning correctly. It verifies that all nodes are properly connected, the control plane components are running, and the worker nodes are ready to accept workloads. If any issues are detected, ClusterAPI provides diagnostics and recommendations to resolve the problems.
By following this workflow, administrators can easily deploy and manage Kubernetes clusters using ClusterAPI. The declarative approach simplifies cluster lifecycle management, ensures consistency across deployments, and allows for seamless scaling and upgrading of clusters.

8.3 Metal3

Metal3 is an open-source project that provides management capabilities for bare metal infrastructure in Kubernetes clusters. It enables administrators to treat physical servers as disposable resources, allowing them to be seamlessly integrated into Kubernetes deployments. Metal3 abstracts away the complexities of managing bare metal hardware, simplifying the process of provisioning, configuring, and maintaining these resources within a Kubernetes environment. [14]
Metal3 leverages the Kubernetes control plane and various components to manage bare metal infrastructure. It operates based on the concept of Custom Resources (CRs), which allow administrators to define and manage bare metal hosts as Kubernetes objects. Metal3 combines the flexibility of Kubernetes with the capabilities required for managing physical hardware, providing a unified and scalable approach to bare metal infrastructure management.

8.3.1 Major Components of Metal3

Metal3 comprises several major components that work together to manage and operate bare metal infrastructure. These components include:

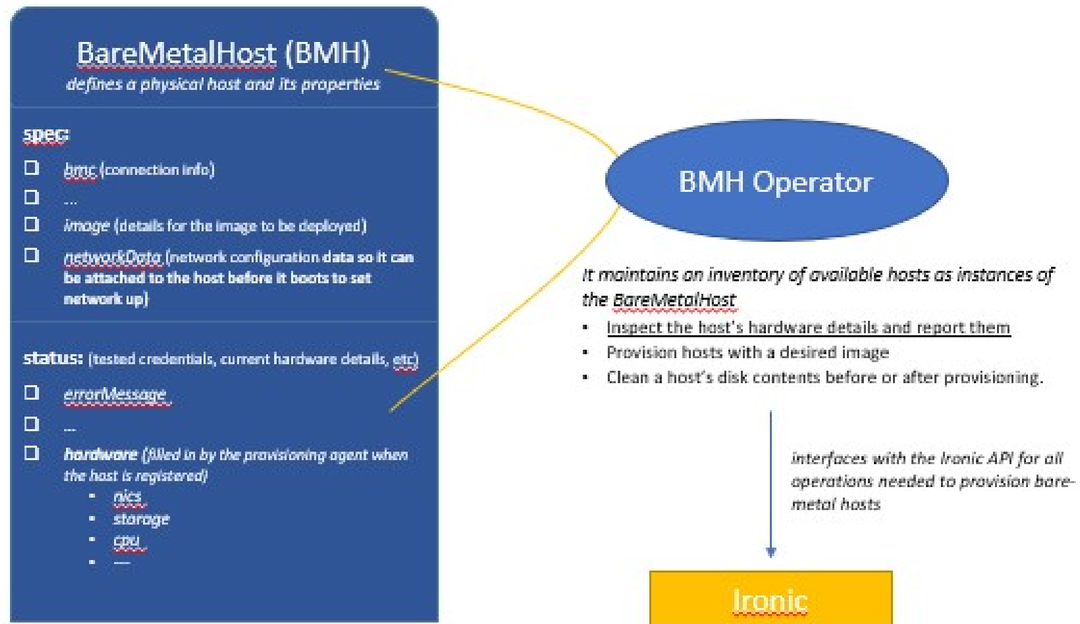


Figure 8.1: BareMetalHost operator

Bare Metal Operator

The Bare Metal Operator is a core component of Metal3 responsible for the lifecycle management of bare metal hosts. It interacts with the underlying infrastructure and orchestrates the provisioning, registration, and maintenance of bare metal hosts within the Kubernetes cluster. The Bare Metal Operator ensures that the hosts are properly discovered, provisioned with the necessary operating system and software, and integrated as worker nodes within the cluster. It monitors the health and status of the hosts, performs automated repairs and updates, and manages the decommissioning of hosts when necessary.

Image and Metadata Operator

The Image and Metadata Operator is responsible for managing machine images and metadata used for provisioning bare metal hosts. It allows administrators to define machine images with specific operating systems and configurations. These machine images are stored and made accessible to the provisioning process. The Image and

Metadata Operator also manages metadata related to the hosts, including network configuration, storage settings, and other custom attributes. It ensures that the necessary image and metadata are available during the provisioning process to enable the successful configuration of the bare metal hosts.

Machine API

The Machine API is a Kubernetes API extension provided by Metal3. It introduces the BareMetalHost Custom Resource Definition (CRD), which allows administrators to define and manage bare metal hosts as Kubernetes objects. The BareMetalHost CRD specifies the desired configuration for each host, including hardware characteristics, network settings, storage options, and other custom attributes. Administrators can create, update, and delete BareMetalHost objects using standard Kubernetes API operations. The Machine API provides a declarative approach to managing bare metal hosts, ensuring consistency and enabling easy integration with other Kubernetes resources and tooling.

BareMetalHost Custom Resource Definition (CRD)

The BareMetalHost CRD defines the specifications and status of a bare metal host within the Kubernetes cluster. It includes the following key fields:

- **Spec**: The Spec field specifies the desired configuration of the bare metal host. It includes information such as the hardware profile, network settings, storage configuration, and any custom attributes required for the specific workload. Administrators define the desired configuration by setting values for these fields.

- **Status**: The Status field provides information about the current state of the bare metal host. It includes details such as the provisioning status, operational status, hardware inventory, and any error conditions. The Bare Metal Operator updates the Status field as it progresses through the lifecycle management tasks, providing visibility into the host's current state.

The BareMetalHost CRD allows administrators to define and manage the desired configuration of bare metal hosts, while the Bare Metal Operator ensures that the actual state of the hosts matches the desired state defined in the BareMetalHost objects.

8.3.2 Provisioning Process

The provisioning process in Metal3 involves several steps: [15]

1. Machine Definition

Administrators define the desired configuration for each bare metal host by creating a machine object using the Machine API. The machine object contains specifications for hardware characteristics, network settings, and other custom configurations.

2. Bare Metal Host Provisioning

Metal3 provisions the bare metal host based on the specifications defined in the machine object. This includes tasks such as network configuration, storage setup, and applying any custom configurations specified in the machine object.

3. Verification and Integration

Once the provisioning process is complete, Metal3 verifies the status of the provisioned host to ensure that the desired configuration has been applied successfully. It checks network connectivity, storage availability, and any other defined attributes to validate that the host is ready for use. The provisioned host is then integrated into the Kubernetes cluster as a worker node, enabling it to participate in the cluster's workload distribution.

8.3.3 Retrieving Information about the Physical Host

Metal3 provides capabilities to retrieve information about the physical host. The following specifications can typically be retrieved:

- CPU Details: Metal3 can retrieve information about the CPU, including the number of cores, clock speed, architecture, and vendor.
- Memory Configuration: Information about the memory configuration, such as the total memory capacity and memory speed, can be retrieved.
- Network Interfaces: Metal3 can detect and provide details about the installed network interfaces on the physical host, including MAC addresses, link speeds, and supported features.
- Storage Devices: Information about the storage devices present on the host, such as hard drives or SSDs, can be retrieved, including the device model, capacity, and connection type.
- BIOS/Firmware: Metal3 can gather details about the host's BIOS or firmware version, vendor, and other relevant information.

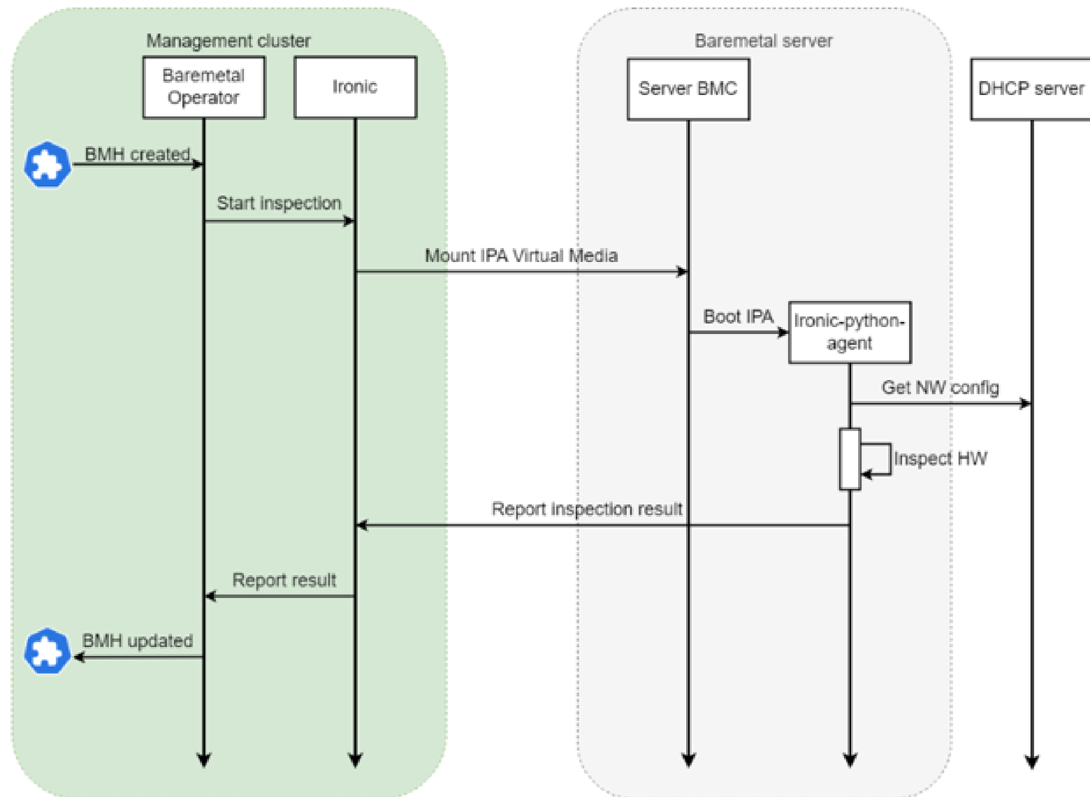


Figure 8.2: metal3 inspection workflow

By retrieving this information, administrators can gain insights into the physical host's hardware capabilities, which can be used for workload placement, resource allocation, and performance optimization within the Kubernetes cluster.

8.3.4 Conclusion

Metal3 provides a powerful solution for managing bare metal infrastructure within Kubernetes clusters. By abstracting away the complexities of physical hardware management, Metal3 enables administrators to seamlessly integrate bare metal resources into their Kubernetes deployments. The combination of Metal3's major components, including the Bare Metal Operator, Image and Metadata Operator, and Machine API, facilitates efficient provisioning and management of bare metal hosts. Additionally, Metal3's ability to retrieve information about the physical host's specifications further enhances the flexibility and optimization of workload placement within the Kubernetes cluster.

With Metal3, administrators can leverage bare metal infrastructure for their Kubernetes clusters, benefiting from the performance and flexibility offered by physical servers.

8.4 Integration of ClusterAPI and Metal3

The integration of ClusterAPI and Metal3 provides a powerful combination for managing Kubernetes clusters on bare metal infrastructure. By leveraging ClusterAPI's capabilities for cluster lifecycle management and Metal3's functionalities for bare metal provisioning, administrators can seamlessly deploy and manage Kubernetes clusters on bare metal servers.
In the context of the intelligent network operator developed in this thesis, integrating with ClusterAPI and Metal3 becomes crucial. The intelligent network operator aims to autonomously discover and understand the network topology, eliminating the need for manual configuration by administrators. By interacting with ClusterAPI's Metal3 provider, a key point of convergence between these components emerges—the IP Address Management (IPAM) functionality.

8.4.1 IP Address Management (IPAM)

In the context of the integration between Metal3, ClusterAPI, and the intelligent network operator, IP Address Management (IPAM) plays a crucial role. IPAM is responsible for efficiently allocating and managing IP addresses within the Kubernetes clusters deployed on bare metal infrastructure. It ensures that each bare metal host within the cluster is assigned a unique and routable IP address.

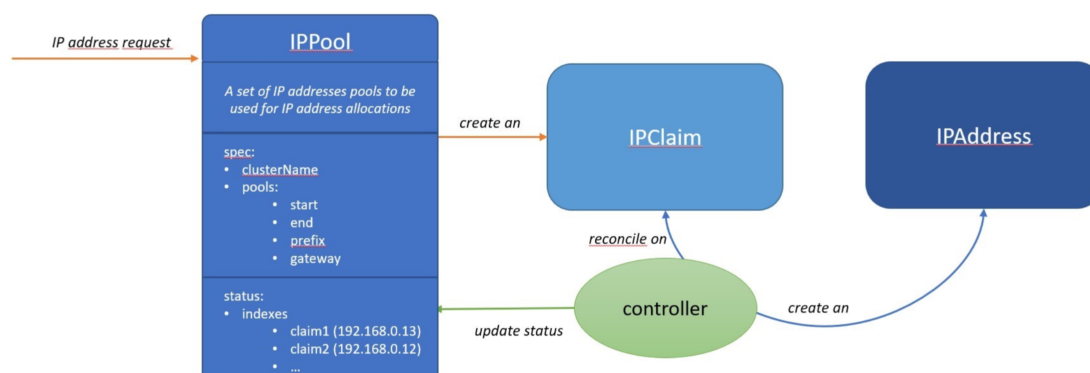


Figure 8.3: IPAM workflow

The IPAM process involves the following steps:

1. IP Address Reservation

When a bare metal host is provisioned or discovered by Metal3, the IPAM component reserves an available IP address from the designated IP address pool. This pool is managed and configured by the IPAM system, which can be part of ClusterAPI's Metal3 provider.

2. IP Address Request

Once a bare metal host is assigned to a cluster and the IP address is reserved, the host requests the reserved IP address from the IPAM component. This request typically includes the host's unique identifier, such as its MAC address, as well as any necessary metadata.

3. IP Address Assignment

The IPAM component validates the host's request and assigns the reserved IP address to the host. It updates the host's IP configuration to include the assigned IP address, subnet mask, gateway, and any other relevant network configuration parameters.

4. Network Configuration

The assigned IP address is then applied to the bare metal host's network interface, enabling it to communicate within the network. The host's networking stack is configured with the assigned IP address, ensuring proper connectivity and routing within the cluster.

5. IP Address Release

When a bare metal host is decommissioned or no longer part of the cluster, the IPAM component releases the assigned IP address, making it available for future allocation. This step ensures efficient utilization of IP addresses within the cluster.
By leveraging the IPAM functionality provided by ClusterAPI's Metal3 provider, the intelligent network operator can interact with the IPAM system to retrieve information about assigned IP addresses and associated hosts. This allows the network operator to autonomously discover and map the network topology based on the IP addressing information provided by the IPAM component.
In summary, IPAM plays a critical role in managing IP address allocation and configuration within Kubernetes clusters deployed on bare metal infrastructure. By integrating with the IPAM system, the intelligent network operator can gain insights into the network topology and dynamically adapt its configuration based on the assigned IP addresses. This integration enables more efficient and autonomous

network management within the cluster, enhancing overall network performance and flexibility.

8.5 Conclusion

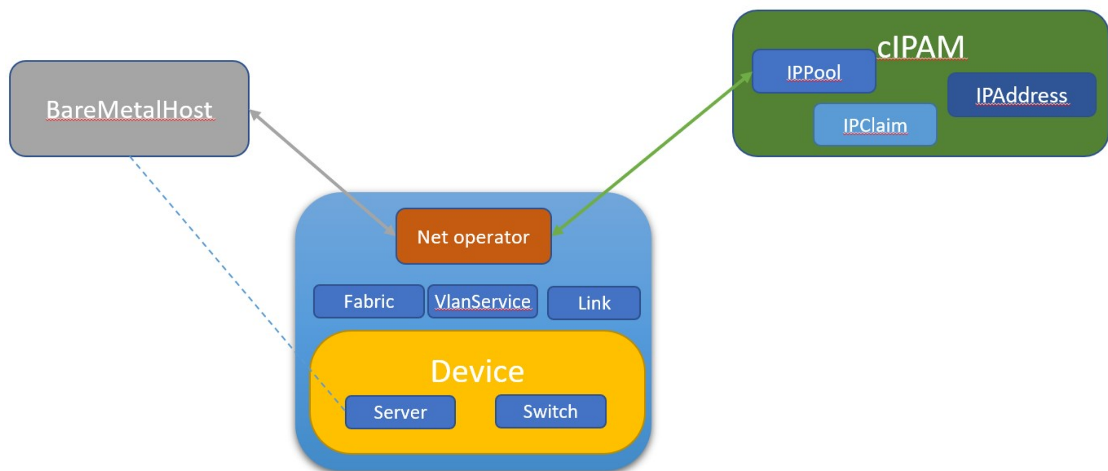


Figure 8.4: Smart Network Operator

In conclusion, the integration of ClusterAPI and Metal3, along with the intelligent network operator, opens up new possibilities for automated and intelligent network management in Kubernetes clusters deployed on bare metal infrastructure. By leveraging ClusterAPI's cluster lifecycle management capabilities, Metal3's bare metal provisioning functionalities, and the autonomous network discovery and configuration of the intelligent network operator, administrators can achieve more efficient and scalable network management. This integration lays the foundation for future advancements in edge network clusters, where the network becomes self-adaptive, self-healing, and able to optimize performance based on real-time insights gained from the underlying infrastructure.

8.5.1 Next Steps

In the future, further research and development can focus on enhancing the intelligent network operator's capabilities by integrating with additional components and technologies. Some potential areas of exploration include:

- Integration with network monitoring and telemetry systems to enable real-time network analysis and performance optimization.

- Incorporation of machine learning algorithms to enable predictive network management and proactive fault detection.

By continuously improving the network operator and exploring new avenues for integration and enhancement, administrators can achieve highly efficient and autonomous network management within Kubernetes clusters deployed on bare metal infrastructure.

8.5.2 Final Remarks

The future of network management in Kubernetes clusters is promising, particularly in the context of edge computing. As edge deployments continue to grow, the need for intelligent and automated network management becomes increasingly crucial. The integration of ClusterAPI, Metal3, and the smart network operator provides a solid foundation for building advanced network management solutions tailored to the unique requirements of edge environments.

Chapter 9

Conclusion

In the rapidly evolving landscape of cloud-native technologies, this thesis addressed a pertinent problem area: the automation of network configuration in bare metal infrastructures at the network's edge. This pursuit was set against the backdrop of rigorous standards defined by telecommunications contexts and regulatory requirements, with the solution designed to be compatible with the principles of the Sylva project.

The primary challenge that this thesis tackled was the traditionally manual, labor-intensive, and error-prone process of network switch configuration in bare metal environments. The proposed solution was a comprehensive approach that leveraged Kubernetes operators to automate this task, aiming to improve reliability, efficiency, and ease of management.

The proposed architecture centered on two critical components: the Network Operator and the Actuator Operators. The Network Operator, through a set of specially designed Custom Resource Definitions (CRDs), managed the resources defining network topology and connectivity. On the other hand, the Actuator Operators acted as intermediaries, executing the network configurations on the fabric and facilitating the interaction with the actual network devices.

The operators were implemented using Kubebuilder, an open-source Software Development Kit (SDK), with Go as the primary programming language. An exemplary outcome of this implementation was the creation of an Actuator Operator compatible with Cisco Network Services Orchestrator (NSO), a widely used network orchestrator, demonstrating the practical applicability and flexibility of the system.

Looking forward, the future directions for this work involve significant advancements in the design and capabilities of the Network Operator. Future developments aim to enable the Network Operator to autonomously build the network topology by interfacing with other components present within the management cluster. Key amongst these are ClusterApi and Metal3, which provide critical information about the cluster's resources and bare metal nodes, respectively.

In conclusion, this thesis successfully showcases the potential of Kubernetes operators to automate network configurations in complex, bare metal environments. This automation strategy presents a significant stride towards the realization of a comprehensive, flexible, and highly automated network infrastructure.

Bibliography

[1] Kubernetes. *What is Kubernetes*. 2021. URL: https://kubernetes.io/what-is-kubernetes/ (visited on 06/16/2023) (cit. on pp. 3, 5).

[2] GitHub. *Kubernetes*. URL: https://github.com/kubernetes/kubernetes (visited on 06/16/2023) (cit. on p. 3).

[3] Telecoms.com. *TIM gets on board with cloud-native 5G network*. 2021. URL: https://telecoms.com/508726/tim-gets-on-board-with-cloud-native-5g-network/ (visited on 06/16/2023) (cit. on p. 4).

[4] IBM. *What is bare metal?* 2021. URL: https://www.ibm.com/cloud/learn/bare-metal-servers (visited on 06/16/2023) (cit. on p. 4).

[5] Docker. *What is a Container?* 2021. URL: https://www.docker.com/resources/what-container (visited on 06/16/2023) (cit. on p. 5).

[6] Kubernetes. *Kubernetes Architecture*. 2022. URL: https://kubernetes.io/docs/concepts/architecture/ (visited on 06/16/2023) (cit. on p. 5).

[7] *GitHub - kubernetes-sigs/kubebuilder: Kubebuilder - SDK for building Kubernetes APIs using CRDs*. 2022. URL: https://github.com/kubernetes-sigs/kubebuilder (visited on 06/26/2023) (cit. on p. 15).

[8] Cisco. *Cisco Network Services Orchestrator (NSO)*. 2022. URL: https://www.cisco.com/c/en/us/products/cloud-systems-management/network-services-orchestrator/index.html (visited on 06/16/2023) (cit. on p. 16).

[9] Cisco. *Cisco Network Services Orchestrator (NSO)*. 2023. URL: https://www.slideshare.net/CiscoCanada/nso-network-service-orchestrator-enabled-by-tailf-handson-lab (visited on 06/27/2023) (cit. on p. 19).

[10] Linux Foundation Europe. *Project Sylva Announcement*. 2022. URL: https://www.linuxfoundation.org/press/linux-foundation-europe-announces-project-sylva-to-create-open-source-telco-cloud-software-framework-to-complement-open-networking-momentum (visited on 06/16/2023) (cit. on p. 23).

[11] Linux Foundation Europe. *Project Sylva Technical*. 2022. URL: https://gitlab.com/sylva-projects/sylva (visited on 06/16/2023) (cit. on p. 24).

[12] Kubernetes Cluster API. *Design Documentation*. Cluster API Design Documentation. Kubernetes Special Interest Group (SIG). 2023. URL: https://cluster-api.sigs.k8s.io/design/ (visited on 06/22/2023) (cit. on p. 58).

[13] Kubernetes Cluster API. *GitHub Repository*. Cluster API GitHub Repository. Kubernetes Special Interest Group (SIG). 2023. URL: https://github.com/kubernetes-sigs/cluster-api (visited on 06/22/2023) (cit. on p. 59).

[14] Metal3. *Introduction Page*. Metal3 Community Page. Metal3 Project. 2023. URL: https://book.metal3.io/introduction.html (visited on 06/22/2023) (cit. on p. 60).

[15] Metal3 Project. *GitHub Repository*. Metal3 GitHub Repository. Metal3 Project. 2023. URL: https://github.com/metal3-io/metal3-docs (visited on 06/22/2023) (cit. on p. 62).