Enabling Edge Computing Using Container Orchestration and Software Defined Networking

Felipe Andres Rodriguez Yaguache

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 02.09.2019

Thesis supervisor:

Prof. Raimo Kantola

Thesis advisor:

M.Sc. Kimmo Ahola

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Felipe Andres Rodriguez Yaguache

Title: Enabling Edge Computing Using Container Orchestration and Software Defined Networking

Date: 02.09.2019 Language: English Number of pages: 7+57

Department of Communications and Networking

Professorship: ELEC3029 - Communications Engineering

Supervisor: Prof. Raimo Kantola

Advisor: M.Sc. Kimmo Ahola

With software-defined wide-area networks (SD-WAN) being increasingly adopted, and Kubernetes becoming the de-facto container orchestration tool, the opportunities for deploying edge-computing applications running over a SD-WAN scenario are vast. In this context, a service discovery function will help developing a dynamic infrastructure where clients are able to seek and find particular services. Service discovery also enables a self-healing network capable of detecting the unavailable services. Most of the research in the service discovery field focuses in the discovery of cloud-based services over software-defined networks (SDN). A lack of research in containerized service discovery over SD-WAN is evident. In this thesis, an in-house service discovery solution that works alongside a container orchestrator for allowing an improved traffic handling and better user experience through containerized service discovery and service requests redirection is developed. First, a proof-of-concept SD-WAN topology was implemented alongside a Kubernetes cluster and the in-house service discovery solution. Next, the implementation's performance is tested based on the time required for discovering whether a service has been created, updated or removed. Finally, improvements in node distance computation, local breakout support and the usage of data plane programmability are discussed.

Keywords: Edge computing, SD-WAN, Virtualization, Orchestration

Preface

I would like to thank professor Raimo Kantola and advisor Kimmo Ahola for their guidance and especially for their patience during the development of this work. A special thanks also to the Technical Research Centre of Finland (VTT) for the opportunity given to prove, develop and strengthen my research capabilities.

This work is dedicated with infinite love to my parents Mabel and Carlos, for teaching me by example to never give up and shine brightest in the darkest night. This is also dedicated to my sister Karla and my brother Juan Diego, thanks for all the lovely and deep moments during our family calls that for sure have made my time in Finland much more enjoyable. Finally, I would like to thank Truman for being a source of infinite love in our lives, our time with him is flowing like milk and honey.

Contents

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
BGP	Border Gateway Protocol
CI	Continuous Integration
CD	Continuous Development
CLI	Command Line Interface
CNI	Container Network Interface
CO	Central Office
CRDT	Conflict-free Replicated Data Type
DNS	Domain Name System
DPI	Deep Packet Inspection
EPC	Evolved Packet Core
GCP	Google Cloud Platform
GPRS	General Packet Radio Service
GTP	GPRS Tunneling Protocol
H-PCRF	Home Policy and Charging Rules Function
H-PLMN	Home Public Land Mobile Network
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
IoT	Internet of Things
IP	Internet Protocol
ISP	Internet Service Provider
IT	Information Technology
MAC	Media Access Control
MME	Mobility Management Entity
MQTT	MQ Telemetry Transport
NAT	Network Address Translation
NBI	Northbound Interface
NOS	Network Operating System
ONOS	Open Networking Operating System
OS	Operating System
OVSDB	Open Virtual Switch Database
P4	Programming Protocol-Independent Packet Processors
PGW	Packet Data Network Gateway
QoS	Quality of Service
RAN	Radio Access Network
REST	Representational State Transfer
SaaS	Software as a Service
SBI	Southbound Interface
SDN	Software Defined Networking
SD-WAN	Software Defined Wide Area Network
SGW	Serving Gateway
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

URL	Uniform Resource Locator
URI	Uniform Resource Identifier
vCPE	Virtual Customer Premises Equipment
vEPC	Virtual Evolved Packet Core
VIF	Virtual Network Interfaces
VLAN	Virtual Local Area Network
VM	Virtual Machine
V-PCRF	Visited Policy and Charging Rules Function
V-PLMN	Visited Public Land Mobile Network
VPN	Virtual Private Network
vSwitch	Virtual Switch
VTT	Technical Research Centre of Finland
WAN	Wide Area Network
WiFi	Wireless Fidelity

1 Introduction

In this section, motivation for this thesis will be presented alongside the problem that will be addressed. Both main objective and the scope of this thesis will be discussed in order to present the hypothesis used during the attempts for solving the problem. Possible use cases based on the presented solution are also extended as well as the methodology indicating, in a detailed way, the work that is being carried out. Finally, the remaining structure of the document is also included.

1.1 Motivation

Virtualization has become the cornerstone of Internet and the cloud-based services, it has evolved from a cost saving solution to the technology capable of providing the required agility and flexibility needed for service delivery in data centers as well as the infrastructure supporting business essential applications. Virtualization is the abstraction of hardware resources such as network connectivity, processing power or storage. Its main goal is the optimization of IT assets, helping in achieving a superior system utilization, cost reduction, and ease of deployment and management by allowing multiple operating system images to run in parallel using only one piece of hardware [24]. Container-based virtualization and Virtual Machines (VMs) are perhaps the most common types of virtualization. Although there are many differences between them, they both have the necessity to communicate within an IP network. Before the execution of a container or VM, they need to be assigned IP and MAC addresses. When these virtualized entities are assigned IP addresses, the traditional Ethernet and IP networks are stretched to exist inside the physical hosts located in datacenters, not only between them. Virtualization in cloud-computing creates a challenge in the application of traffic engineering for maximizing the utilization of the available conventional networks [9, 25].

Traditional communication networks are distributed systems with multiple routing algorithms running over many different devices such as routers and switches. Every single one of these devices possesses its own configuration and state, and must be configured separately, which makes networks difficult and expensive to maintain and migrate. Software Define Networking (SDN) tackles this issue through the separation of the control plane from the data plane. This is achieved by moving the control logic of the network to a centralized controller, transforming the switches into mere forwarding devices that follow the rules set by the controller. By centralizing the control logic, configuration and maintenance become easier, with new features being able to be deployed much faster as well. The centralized control has information regarding the whole network, being able to optimize the available network resources. SDN is therefore widely spread in datacenters, especially in order to cope with the virtualization and cloud/edge computing related issue [9, 26].

Edge computing is a paradigm referring to the technologies that enable computation to be performed at the edge of the network. However, as data processing power is moving towards the edge of a network in form of containers instead of remaining in a centralized cloud or datacenter, migration is also occurring for services or applications. This trend requires the use of resources that are not constantly connected to the network. This is the case of laptops, smartphones, wireless sensors, etc [27]. The more this approach is adopted, the more businesses think their Wide Area Networks (WANs) are not prepared to carry such a burden, especially when taking into account traditional corporate WANs. Such networks are built by backhauling routed services and Internet traffic throughout the Central Office (CO), which can cause performance issues when combined with edge computing. It is obvious that traditional approaches lack the agility and flexibility to achieve the required performance and availability needed by edge computing [3]. Software-Defined Wide-Area Network (SD–WAN) is the application of SDN to WAN connections and presents a solution to the aforementioned issues. In a SD-WAN implementation the traditional Command Line Interface (CLI) configuration is replaced by centralized, ubiquitous resource control and orchestration. A typical SD-WAN solution uses a centralized WAN edge device in order to establish logical connectivity with other edge devices through the physical WAN that uses a hybrid of Internet alongside another WAN technology, as already mentioned [30]. It is possible to clearly identify three drivers pushing the adoption of SD-WAN solutions: an improvement in WAN performance, a consistent security, and a reduced overall complexity [2, 4].

By adopting edge computing, awareness shall be raised noting an evolution in the traffic patterns is also happening. Therefore the company's adoption of SD-WAN might not be enough for tackling all the resulting issues. The changes in traffic patterns are directly related to the increase in the number of devices connected in every branch, the major drivers for this increase come from: the number of connected devices per employee, number of end-point devices (for example IoT equipment, WiFi access points, etc.) and Microservices. A Microservice is an application that comprises a collection of services provided to customers, such as guest WiFi or artificial reality [3, 4]. Taking this information into account, it is not a surprise that nowadays almost 80% of traffic generated in corporation's branches correspond to the aforementioned micro-services instead of the traditional client/application traffic. Although SD-WAN helps with a general WAN performance improvement, as more micro-services are rolled out to branches, bandwidth and latency limitations may also occur, especially in situations where all the data shall be sent to the CO for every decision to be taken [4].

The more traffic is sent towards the CO, the higher are the bandwidth requirements, as it has already been discussed. A method for solving this issue is to make the corporation's branch smarter by sending traffic towards the CO only for critical processing and if strictly necessary. By applying this, requests coming from branch offices will be directed towards the branch hosting the required service, a quite useful strategy taking into account that computational costs have dropped and that

applications such as IoT and Big Data require or produce a big amount of data [4].

Enabling a smart, remote branch, requires two components: the already discussed SD-WAN and a container orchestration framework that is capable of deploying edge-enabled, cloud-enabled and web-scalable workloads [28]. The idea of containers has existed for more than a decade, with built-in support offered by Operating Systems (OS) such as Linux. However, it was Docker in 2013 that made them manageable and therefore popular among development and IT teams due to scalability and portability provided to developed applications. Core implementations usually use a different approach than production implementation, with the first one being built around the life cycle of individual containers and the latter dealing with many of them running across multiple hosts. The complexity of dealing with multiple hosts raises a demand for new management tools, in other words, orchestration [6].

Orchestration is the automation of management and/or organization of systems or services while reducing errors introduced by personnel involvement in tasks such as provisioning or scaling. Container orchestration is expected to be present in almost all the service deployments in a near future through its massive adoption by companies and startups [29]. Its merge with the SD-WAN technology is still to happen, making SD-WAN adopters unable to obtain the most out of their investments. A few works have somehow dive into service discovery orchestration in SDN networks. In [22] Jarraya et al. analyzed the importance of computing and storage orchestration alongside networking resources as a quite important part in SDN, while also taking into account the lack of research that aims at easing the creation and deployment of network services. In [23] Kreutz et al. identify computing infrastructure and networking challenges, presenting a series of constraints that must be overcome in order to improve efficiency by means of network orchestration. The aforementioned works focus on cloud computing resources orchestration on a data center environment having and underlying SDN network. This thesis focuses on the discovery of deployed containerized services, redirecting service requests and facilitating their access for the end users through the development of an orchestrator composed by a service discovery system that works alongside an external DNS and a reverse proxy.

1.2 Objective

The objective of this work is to determine the plausibility and issues that may arise when implementing an orchestrator for discovering services deployed on a Kubernetes cluster over a SD-WAN network. The role of this orchestrator is to automatize the service discovery in order to make it more agile and responsive for external users. For achieving this, a proof of concept implementation shall be developed and validated. This proof of concept should demonstrate the feasibility of the idea while also bringing to light the possible complications with such approach. It will be implemented as a SD-WAN topology simulation and a series of applications (orchestrator) being able

to identify the changes happening in the container cluster and translating them to information that helps the user to access the service from outside the cluster. The simulation will assume a corporation with one CO and multiple remote branches. Network intelligence, comprising the SD-WAN controller, Kubernetes master node and the service discovery of the proposed orchestrator shall be situated at the CO. Remote branches will be equipped with a SD-WAN gateway and a Kubernetes worker, although it is also possible that the worker will not be present in certain branches. Connectivity between branches and the CO can be enabled through the use of Virtual Private Networks (VPNs) coming across the Internet. Internet network will be simulated as a group of OpenFlow switches with end hosts located in different address spaces.

The orchestrator will work alongside the Kubernetes master node for the dynamic and efficient discovery and association of services in the deployed workers, facilitating their access from external locations, helping with traffic control and improving the overall management experience. As was mentioned before, the orchestrator will include an external DNS and an external reverse proxy located in the branch gateways. This characteristic allows a simple and fast deployment while avoiding overlapping with the internal Kubernetes services. External DNS and reverse proxy are used by the clients when accessing a local o remote service and rely on Kubernetes' internal DNS and proxy. As changes occur in the Kubernetes cluster, the DNS and reverse proxy service will be updated accordingly.

1.3 Use case

With SD-WAN being the natural extension of SDN and Kubernetes becoming the de-facto container orchestrator, the possible use cases involving Edge Computing are really high in number. In the present use case, a Kubernetes master node will be deployed in the CO alongside the proposed orchestrator that will be aware of the changes happening in the Kubernetes cluster and will react accordingly. The development of this orchestrator will enable the possibility of deploying container workloads on remote branch locations and, at the same time, facilitating access towards its services by properly discovering the nodes containing them. In Figure 1, the proposed topology is depicted.

The deployed workloads can be of an almost infinite variety. Some examples are: smart web pages, authentication applications and IoT data processing. Each of these use cases have different requirements and setups. The smart web page can possess three main components, a front-end, a web application, and the logic. Each of these three components can be deployed in different Kubernetes workers while the Kubernetes master perform loads balancing. Network requirements will be low latency and dynamic route adaptation in case a Kubernetes worker is replaced or moved. The authentication application can be deployed into one remote branch, then

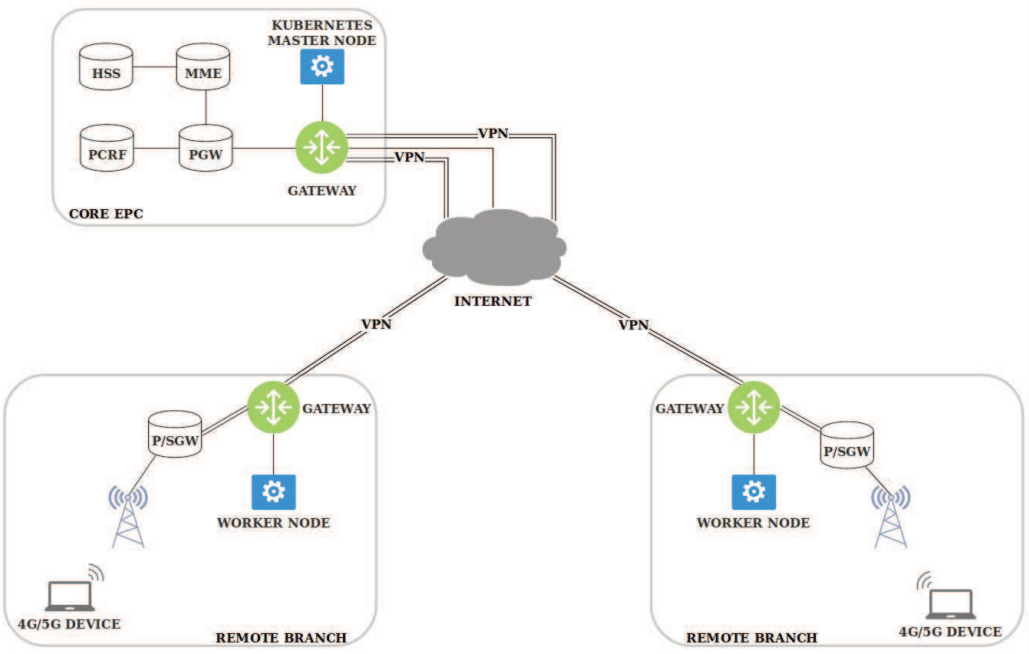

Figure 1: Proposed use case topology.

all authentication queries coming from closer branches will be redirected towards it instead of going to the CO. This will require the discovery of the closest node running the authentication application. Finally, for IoT data processing use case, some branches can generate the data by using sensors and store it in an Kubernetes worker, while the data processing unit can be deployed in a different one. The massive amount of traffic generated by sending and receiving raw data as well as processed data should not be directed through the central office unless it is strictly necessary. In all the use cases, the need of dynamic service discovery is evident. Although SD-WAN is just starting to be adopted, the integration with Kubernetes will definitely ease the deployment of applications and provide an improvement in the user experience. Due to the lack of commercial solutions that provide Kubernetes service discovery in a SD-WAN environment, the proposed paradigm can be further developed as a viable business idea.

1.4 Methodology

This work will be carried out in three different, sequential steps. First, a literature survey required to fully understand the functioning and requirements of SD-WAN networks is carried out covering existing solutions and projects as well as the role of an orchestrator in such implementation. The same step is applied for Kubernetes, in order to fully understand the usage of its API solution. The second step is the proof of concept implementation. The main goal of this stage is to add as many features as possible and to achieve a real-life WAN environment for allowing the

testing of a wide variety of use cases. ONOS controller was selected for being an extremely reliable and well tested solution, with an active network of developers working on it as well as a large enough number of already available applications. Network simulation will be done using Mininet, a virtual network emulator that uses Linux namespaces for separating individual nodes in an emulated network. Although the SD-WAN gateways emulated with Mininet will be running in the same host, they can as well be deployed in different physical machines as well as NoviFlow or Pica8 switches [8]. Finally, the implementation is validated and compared against commercial solutions, the whole system will be examined looking for problems and limitations that shall be discussed so improvements can be proposed. A testing using a bare metal implementation will also be considered, so a conclusion can be obtained from this experiment.

1.5 Structure

This thesis is structured as follows. In Chapter 2 the state of the art regarding the technologies used in this work will be presented. In Chapter 3, the implementation of the proof of concept will be detailed, focusing on its capabilities and limitations. In Chapter 4, the solution provided will be validated and its features discussed, and analysis on its impact as a commercial solution will be provided. In the same way, the measurements result as well as future research towards improvements is also included. Finally, in Chapter 5, a conclusion from all the carried-out work is drawn.

2 State of the art

In this chapter, an insight into the technologies this work uses is provided. These are: Software Defined Wide Area Network, containers and orchestration, and switch virtualization.

2.1 Software defined wide area network

Traditional WAN solutions have fallen behind in the era of applications and cloud adoption. WAN is still a complex technology depending on box-by-box manual configurations, including QoS with parameters like bandwidth, which has to be manually configured and does not adapt to link conditions. The necessity of being fluent in a specific vendor's CLI language makes the training and troubleshooting a complex and costly process, especially when different vendors products are present in the network. Similarly, when load balancing is required in a WAN network, this must be manually configured. WAN resources are usually utilized in an inefficient way due to failover circuits being most of the time in standby mode, and after a link failure service usually takes several seconds to converge, affecting negatively the end user experience [8]. WAN limitations are often more visible during cloud migration due to Internet, Software as a Service (SaaS) and cloud—hosted or edge–hosted applications still being backhauled using private networks which are usually congested, causing a negative impact on performance. SD-WAN is the paradigm that has emerged to help tackle these limitations [3, 8].

SD-WAN is capable of providing to the enterprise's branches all the advantages that are associated with the adoption of SDN in data centers. Both technologies virtualize network resources in order to provide an improvement in performance and availability through the automation of network deployment and a reduction in the cost of ownership. The basic principle of SDN is the abstraction of the network to a series of capabilities independent of the way they are provided. This means that applications being deployed do not have to include specific details about the underlying network. In the same way, SD-WAN provides a software abstraction in order to create a network overlay decoupling the virtualized network services from the underlying WAN network. This new abstraction allows a much simpler network management compared with the traditional, vendor-specific, underlying hardware WAN network. The network overlay provides a common interface located across different physical components that will enable businesses to build their own infrastructure independent applications [3, 8].

SD-WAN as well as SDN performs a separation of the control plane and data plane. Control plane is the part of network responsible for routing decisions, signaling traffic and including the system configuration and management. Data plane is the part that

forwards application's and user's data based on the rules set by the control plane. A core concept of these software-defined technologies is that one logical instance of the control plane is able to serve multiple data plane instances. In contrast, a traditional network element contains its own control plane and data plane, thus making it impossible to program the whole network [6, 8]. By separating the control plane from the data plane, the agility of the network service is boosted due to intelligence being moved from the network elements to the more centralized control plane. In the same way, OpenFlow protocol enables the communication between the control plane and all the data plane devices. This protocol is often called Southbound Interface (SBI).

Finally, an API will enable applications to program the network as an abstraction. APIs are often called Northbound Interface (NBI). SDN can be applied not only in data centers, but also in the links between them. With the democratization of OpenFlow, Internet service providers (ISPs) have successfully applied SDN in their Radio Access Networks (RANs), Virtual Evolved Packet Core (vEPC), Virtual Customer Premises Equipment (vCPE), among others [3].

2.1.1 OpenFlow

OpenFlow is a widely accepted and deployed open southbound standard for SDN. Its fundamental abstraction is defining the packet forwarding process, installing forwarding policies, tracking the forwarding process timely and dynamically controlling this process. It started in the 00s addressing the need of conducting research in unrestricted network architecture. OpenFlow designers discovered and leveraged a set of functions that were present in routing and switching devices such as the flow tables, allowing its first version to be easily deployed on existing hardware without requiring exposing the internal structure of the devices. This led to a quick adoption and support from many commercial vendors such as Google, although its flexibility was somehow compromised due to the legacy switch hardware [5, 8].

Thanks to OpenFlow, popular network devices and middle-boxes such as Network Address Translators (NAT) or Firewalls could be easily replaced by general OpenFlow enabled switches or software applications, due to these devices differing mainly in the fields of the packet header they match and the actions they perform [5]. A flow can be defined as a train of packets following a common pattern. The flow table contains a list of rules called flow entries, each of these rules defining a pattern of packets and how to process those packets. Each of the flow entries have a priority for matching precedence and a packet counter for statistical purposes. Based on this, matching can be defined as the process of checking if an incoming packet follows the pattern determined in a flow entry. The section of the flow that is used to determine whether a packet matches it, is known as matching fields. Generally speaking, a packet shall only match one flow entry, this being the entry with the highest priority

value. However, when located in different tables, packets can match more than one flow entry. Once a match is found in the flow table, a set of actions are performed. On the other hand, if no match is found, the packet is either dropped or sent to the controller [5, 8].

An OpenFlow action is an operation that can forward the packet to a port, modify the packet or change its state. Both a list of actions and a set of actions represent a certain number of actions that must be performed in a determined order, although there is a minor difference between them. Actions in a set can be applied only once while those in a list are cascaded and their effect accumulated. Forwarding is understood as the process of selecting the out port(s) of an incoming packet and performing the corresponding redirection. This process can involve matching the packet against a specific flow table, finding the best matching entry and applying the determined actions. The flow tables used for forwarding form the forwarding pipeline and the pipeline fields indicate the set of values attached to the processed packet along the forwarding pipeline. The set of values involved in packet processing is called datapath, term also associated with the virtual switch [5, 31].

With the adoption of OpenFlow, development switched towards distributed systems due to the need of more than one OpenFlow controller acting as one single entity in order to maintain the scalability, reliability and performance in a deployed network. OpenFlow allowed the conception of the Network Operating System (NOS), allowing the split of an SDN network into three layers: application layer, control layer, and infrastructure layer as depicted in Figure 2. The network controller is located in the control layer and communicates with the OpenFlow enabled switches via the SBI. The application layer contains the software applications that dictate the logic of the entire network and communicates with the control layer via the NBI [5, 31].

2.2 Containers and orchestration

Virtualization done at an OS level, also known as containerization or container-based virtualization, is a popular, lightweight alternative against the traditional hypervisor-based virtualization. Container based virtualization is defined as lightweight due to the complete absence of a hypervisor, drastically reducing runtime and storage overhead, and allowing all virtualized entities (i.e. containers) to share a common OS kernel. However, container-based virtualization shall not be considered a silver bullet. By sharing a common OS kernel, it provides a much weaker isolation when compared with hypervisor-based virtualization. The level of abstraction provided by containers rests on top of the host OS kernel, allowing each one of them to behave as an independent, isolated operative system [6, 32]. Abstraction level of container-based virtualization is shown in Figure 3.

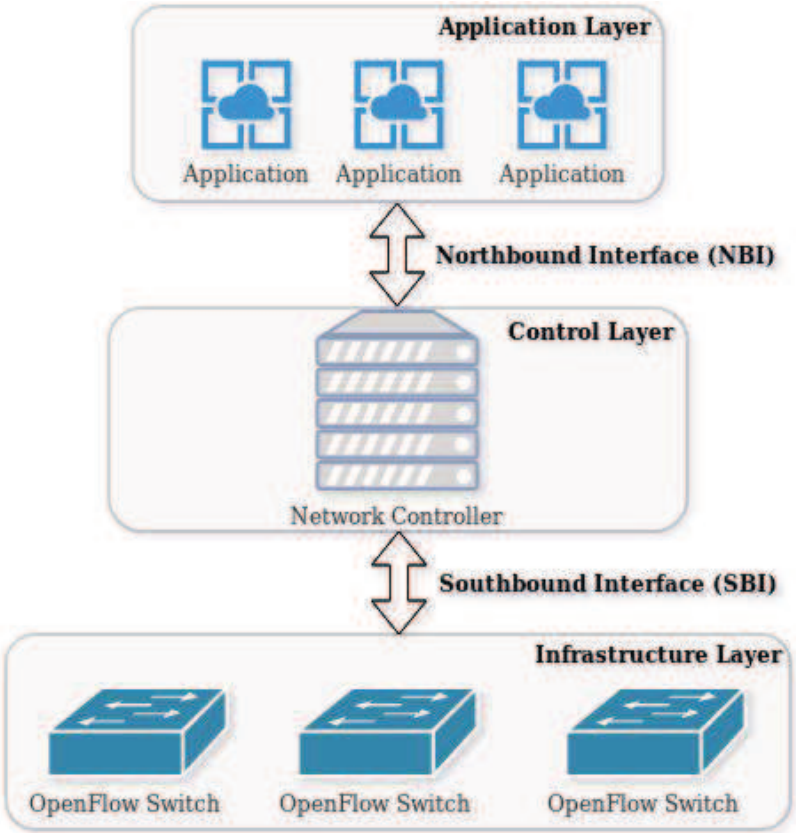

Figure 2: OpenFlow SDN architecture.

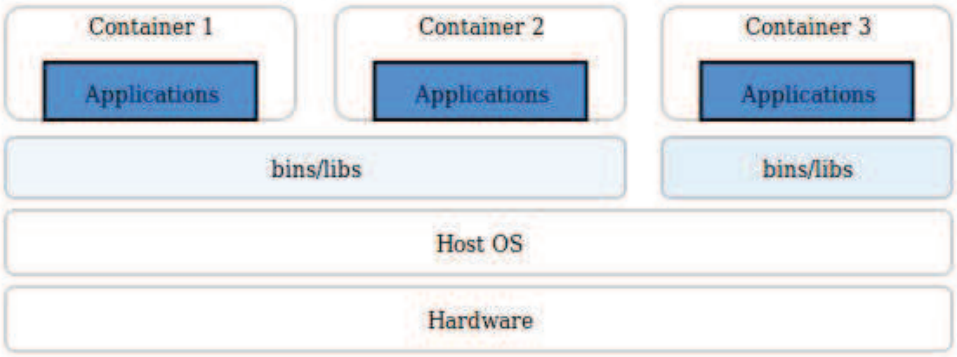

Figure 3: Container-based virtualization architecture.

Containers are created based on images, an image is a standalone executable package that can be backed-up and distributed and contains all the necessary files for the container to run, such as system resources and runtime. Depending on fault tolerance requirements, multiple instances of a container can be deployed on the same or different host machine. From the end-user perspective, containers provide a

strong isolation illusion, leading to the assumption that processes executing inside them are actually running on different virtual machines. Most of the container-based virtualization is carried out using Linux kernels, whose management interface enables an easy execution, monitoring and administration of containers. Isolation is provided through the use of Linux namespaces, allowing each container to have a different view of the underlying OS and enabling resource management through the control groups (cgroups). The most widely adopted container-based solution is Docker [6, 43].

2.2.1 Docker

Docker is the containerization software by default and has almost become a synonym with the term. It is now developed under the Moby Opensource project, which also includes elements such as containerd, linuxkit, and infrakit. Docker allows a separation of application and infrastructure, with Linux cgroups and namespaces as its building blocks it enables an easy developing, maintenance, shipping and running of container-based applications. It provides a mechanism for easing the version management, container management and deployment of containers. Due to Docker leveraging container-based virtualization, it is possible to run multiple containers on a single host machine in a secured enough manner [1, 44]. Docker containers are run by Docker Engine, which is illustrated in Figure 4.

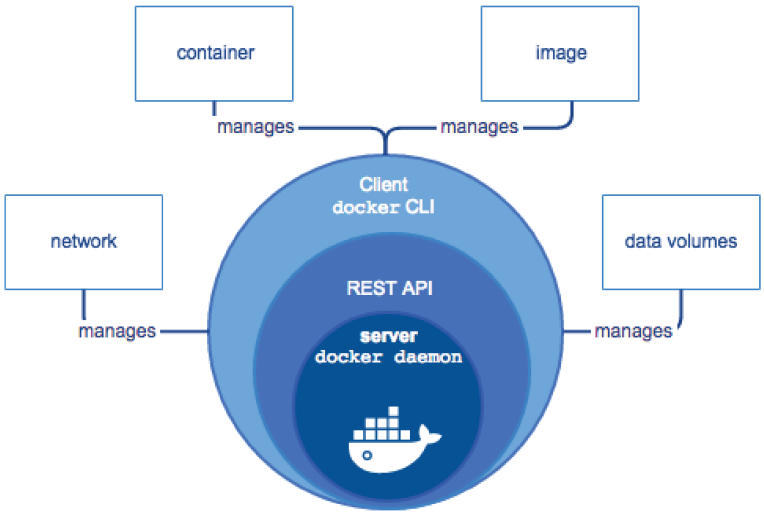

Figure 4: Docker engine architecture.

Source: https://docs.docker.com/engine/images/engine-components-flow.png

Docker has become popular with Continuous Integration (CI) and Continuous Development (CD) workflow approach in Agile, mainly thanks to its capacity of merging the development and production environments, reducing issues and also the market time for software-based products. Containers are stateless entities that can be created and deleted dynamically according to the requirements. For example, in certain IoT use cases that require large storage, Docker provides a database image that can be used to scale the database vertically. When they are not needed anymore, they can be scaled down, providing smart scaling and workload management [6, 33].

Docker Engine is based on the client-server architecture and contains three main components: a server running as a daemon process, a REST API that indicates the interfaces available for applications to communicate with the server, and a CLI running as client that uses the REST APIs to communicate with and control the daemon through the docker command or scripts. Docker containers are managed by the user via the docker command. The docker command communicates with the docker daemon, which in turn builds, runs and distributes the containers. For the client to communicate with the daemon they are not required to be in the same host machine or to use the same system, communication is carried out by using Unix sockets or over a defined network interface [1, 6].

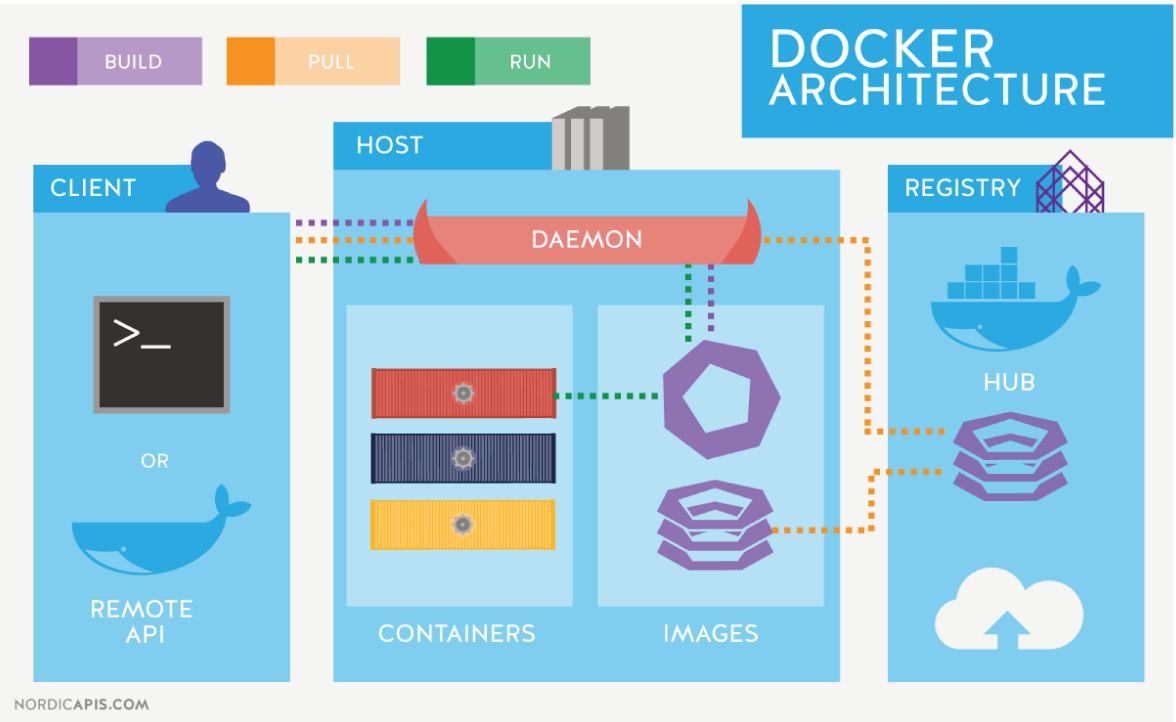

Figure 5: Docker architecture.

Source: https://www.aquasec.com/wiki/download/attachments/2854889/Docker

As seen in Figure 5, docker architecture is built by multiple equally important elements. First there is the docker daemon, which manages the docker objects such as images, containers and volumes and communicates with the docker client for managing the containers; it is also capable of communicating with other daemons for a more efficient service management. Direct communication with the daemon is restricted, only a docker client is able to communicate with it [1, 6]. Next, there is the docker client that is used for interaction with the docker platform via the CLI provided by the client. User commands are sent to the daemon that executes them. An important feature of the docker client is its ability to communicate with more than one docker daemon [1, 6]. The docker registries are used for storing the docker images. The most popular registry is Docker Hub which is docker's default selection when searching for images. They simplify image distribution due to being accessible for the clients to download docker images. The client's commands associated with registries are "docker pull" and "docker push", used to pull and update images in a registry [6].

A docker image is a template used for creating docker containers. A container is based on a determined base image with some additional customization that is needed according to the application running. Users can also create Docker images by creating a Dockerfile that contains the required characteristics of the required image. Every single characteristic defined in the Dockerfile will create a layer in the image, subsequent changes will result in only the modified layers being rebuilt, this allows for a complete audit trail of changes between different versions [1, 6]. Dockerfile is a text file containing all the commands and instructions required for building a docker image, configuring a container or running an application inside a container. Dockerfiles are pointed when using the command "docker build" in order to build an image. As mentioned earlier, the build system will only modify the layers based on the line changes in the Dockerfile compared to previous versions. The first line of a Dockerfile is always the command "FROM", that specifies the base image to be used. In addition, system configuration such as listening ports can also be performed [1, 6]. Finally, there are the docker containers, that are defined as the running instances of an image. They are isolated from each other and the host machine they are running on top. One single docker image can be used for instantiating several containers, with their network, storage as well as other system properties modified according to the application being deployed. This is possible due to each container having its own writable container layer, allowing them to share the same image but having different data states. All changes are saved in a writable layer, as the container is deleted or removed, the writable layer is also removed, thus assuring the base image is unchanged [1, 6].

Docker is the ideal solution for micro-services deployment. It allows an application to run its own, isolated container. This container could be located either in the same host or a different one. As it was explained earlier, the image repository in docker allows for upgrading or downgrading applications in a trusted manner thanks to the help of the docker client. The most useful tools during the deployment,

maintenance and automation of micro-services are the already discussed docker registry, Dockerfile and docker client. Dockerfile is the element enabling automation and ease of deployment using images from the docker registry while the docker client allows communication and control with the container hosting the micro-service as well as logging and monitoring it. Automation of deployment and testing of applications are the base of the CI/CD pipeline. Therefore, Docker has become the enabling technology for CI/CD, features that make docker containers the most popular and well-suited platform for the implementation of the micro-services architecture [6, 33].

2.2.2 Kubernetes

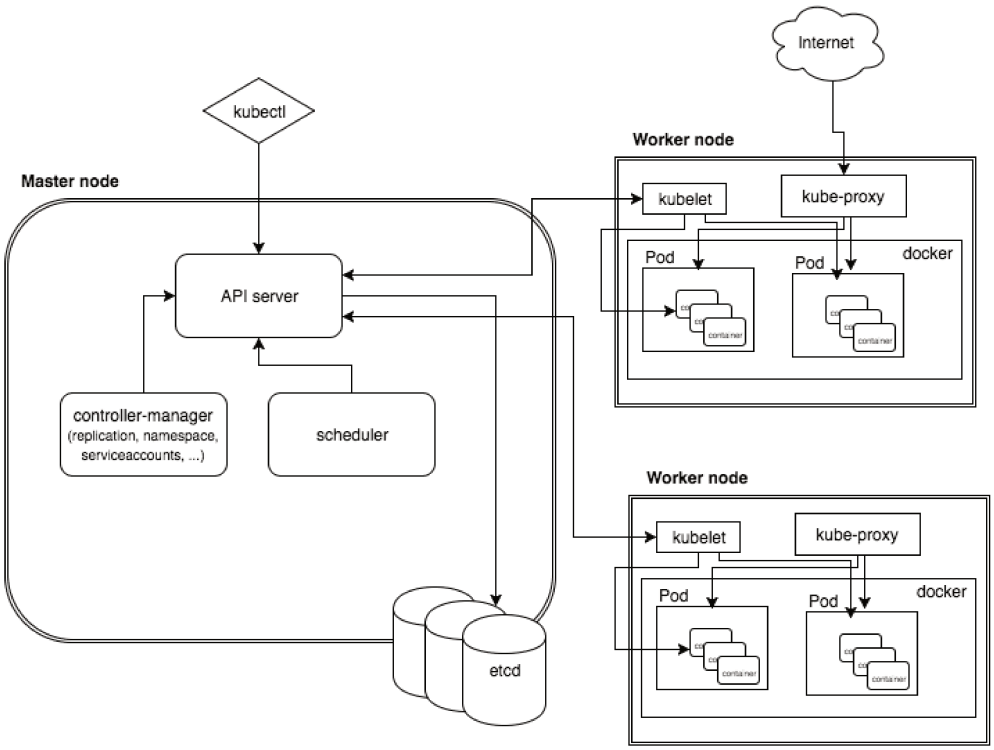

Figure 6: Simplified Kubernetes architecture.

Source: https://x-team.com/blog/introduction-kubernetes-architecture/

With docker containers becoming the standard for scalable applications development, the increase in the number of containers required for satisfying the micro-service architecture shall not be ignored as it raises multiple challenges for managing the containers, with their distributed nature complicating this scenario even more. In order to deploy the micro-services, several manual configuration steps are required, and automating this stage requires long and complex scripts that are difficult to

maintain. A container orchestrator provides many features such as host provisioning, container instantiation, rescheduling of failed containers, container interconnection, resilience, secure container exposure out of its cluster, scaling containers up and down as well as updating the container's images. The de-facto solution for docker container orchestration is Kubernetes [34]. Kubernetes is an extensible, open source platform for managing containers as well as services, it support declarative configuration and automation. Kubernetes has its origins in Google's cluster management system called Borg, which Google used for running and managing containers inside data centers for more than a decade. Kubernetes can manage systems with hundreds of thousands of workloads from different clusters, it was designed with the idea of providing the tools for easing the deployment, scaling and management of applications, as well as providing a management platform known for being container-centric [1, 6].

As seen in Figure 6, a pod is considered one the basic blocks of Kubernetes, its smallest and simplest unit. It can be defined as a representation of a process that is running in the cluster. The basic pod implementation requires one single container per pod while some deployments will require multiple containers per pod sharing the common resources such as storage or network, with containers inside the pod always being created in the same host machine. Containers located inside the same pod communicate with each others using localhost and with external resources using shared network resources like ports and a unique IP address that is assigned to every pod. Kubernetes does not orchestrate containers, but pods. When an application needs to scale, it can be translated to "adding more pods", a process that is also known as replication. Pods are ephemeral and can be created or destroyed at will, although some pods can also be destroyed due to the lack of resources. Therefore, keeping them stateless is quite important [1, 6].

Another element is the node, also known as a worker. It can be a physical or virtual machine, they provide the runtime for the pods and are managed by the Kubernetes master. A node has two components: kubelet and kube-proxy [1, 6]. Kubelet drives the execution layer and implements the pod and node APIs, it takes the PodSpec from a ".yaml" file and ensures that the containers are running without any issues. Kubelet does not manage any container created outside the Kubernetes environment and among its main responsibilities are the creation, monitoring and graceful termination of containers as well as node status reporting [6]. Next, we have kube-proxy, which runs on each node and is in charge of configuring the IPTABLES, for enabling the correct redirection of the service traffic providing a highly-available load balancing solution with a low performance overhead. Kube-proxy works in the transport layer and can perform a simple TCP/UDP stream forwarding, uses round-robin for implementing load balancing and uses Domain Name Service (DNS) for end point resolution and discovery [1, 6].

One of Kubernetes main components is the master, which serves as the control plane of a Kubernetes cluster; an individual master node can be run on any machine in a determined cluster although for simplicity all master nodes are usually started in

the same machine. The main components of a Kubernetes master are: kube-apiserver, etcd, kube-scheduler and kube-controller-manager [1, 6].

Kube-apiserver can be considered the front-end of the Kubernetes control plane, it is a simple server that exposes the Kubernetes API, processes REST operations and updates the corresponding objects in etcd. The simple server provides a gateway towards the cluster for clients located outside and is horizontally scalable. In this context, horizontal scalability means the possibility of adding more pods to the system, while vertical scalability is related to the increase in power (i.e. CPU). API server is the responsible entity for the communication between the cluster control plane and the nodes, it is used by the commands "kubectl" and "kubeadm" [1, 6]. Etcd is the element providing storage for the cluster data, including the cluster's state. It is a distributed and highly-available key-value store and provides a reliable access to all of the cluster's critical data. It also watches for any change to the cluster state so multiple components can be notified in time [1, 6]. Next, there is Kube-scheduler. It is the entity responsible for assigning nodes to pods by constantly watching the recently created pods that do not have a node assigned and assigning a node for them to run. Among the factors used at the moment of selecting a node for a pod, the following can be listed: availability of resources, hardware constraints, software constraints, data locality and affinity. Kube-scheduler is also responsible from removing a pod from a node by taking into account the current state of the cluster [1, 6].

Kube-controller-manager is a control loop that is in charge of managing the controllers. It watches the share state of a cluster and also makes changes to the components to move towards the desired state by using kube-apiserver. There are different controllers treated as one single entity for simplicity reasons. They are: node controller in charge of managing nodes and their state, replication controller controlling the desired number of pods, endpoints controller that links services to their respective pods, and service account and token controllers in charge with the creation of default accounts and API access tokens for the creation of new namespaces [1, 6]. Finally, if a cloud provider exists, then cloud-controller-manager will interact with this underlying cloud. It is a new feature still in alpha state released in Kubernetes 1.6, although present in previous versions as part of the control manager. The difference in development cycles between Kubernetes and cloud providers led to the introduction of this feature. Cloud-controller-manager allows vendors to abstract their cloud specific code, leading the most popular providers like Google or Amazon to develop their own versions of cloud-controller-manager [1, 6].

The replication controller is the element in charge of assuring that a pod or set of pods are always up and available. A set of pods is also known as "replica", and they are needed for scalability purposes. During horizontal scaling, more pods are created in order to handle the incoming traffic. The replication controller is able to supervise multiple pods in different nodes, replacing failed pods with new ones. As explained before, pods are ephemeral and can be created and destroyed at will,

which is a strong point in favor for the use of replication controller even if one single pod is being used. Replication controller is also in charge of deleting the excess of pods from the cluster. The controller can be terminated either alongside its managed pods or without terminating the pods, at which point the pods will disassociate from the controller [6].

Services in Kubernetes are defined as a logical set of pods and the policies required to access them. Due to pods being constantly created and deleted, a pod can be replaced at any point of time with a different IP address associated to it. These characteristics complicate communication with pods, a problem that is solved by using a service abstraction where pods providing the same service are labeled using the same label selector. It must be taken into account that services are REST objects just like pods, and therefore, RESTful operations can also be performed on them. The goal of the service is to ensure a high-availability to the users. Thus, the service object is updated as a response to any change in the pod(s), service endpoints are created alongside the service containing details of the pods [1, 6].

Addons are pods and services used for extending the functionality of Kubernetes. There are addons available for networking and network policies, service discovery as well as visualization and control, among others. A quite useful addon is Dashboard, which allows visualization of Kubernetes clusters through a web interface [6].

2.3 Switch virtualization

Traditional physical switches are boxes with a determined number of ports and they connect to other network devices through Ethernet, fiber or wirelessly. As networks are increasingly virtualized, so are the switches, transforming these hardware boxes into software capable of interconnecting Virtual Network Interfaces (VIFs) in the same manner a physical switch would do with physical interfaces. Virtualized switches were initially developed with the purpose of enabling network connectivity among Virtual Machines (VMs) deployed on the host. Their main advantage is that they allow the use of information such as VM state, which is not always possible with a normal physical switch. Virtual switches are usually leaf nodes of the physical network, with the connectivity enabled through the mapping of VIFs to physical interfaces [7 ,8]. Their usage as leaf nodes is also related to simplicity, there is no need for routing protocols to be deployed in virtual switches under this condition. Virtual switches are an important part of network research and simulation, Mininet is perhaps the most widely used network simulation tool, and it makes use of virtual switches for the prototyping of large networks [8, 35].

2.3.1 Open vSwitch

Open vSwitch is probably the most popular open source virtual switch. It was designed as a cross platform solution for offering an adequate performance and flexibility while still remaining generic. Open vSwitch has been OpenFlow enabled from the very beginning, and today it is actively being developed by The Linux Foundation. It has been included in multiple virtualization platforms and operating systems and has widely been adopted in different commercial and production environments [7, 8]. One of the most important components of Open vSwitch is ovs-vswitchd, a daemon implementing the switch and a companion Linux kernel module for flow-based switching. When a packet arrives, the datapath kernel performs an action, like forwarding or modifying the packets, based on the action or set of actions defined by ovs-vswitchd, if no instructions are set up, then the packets will be forwarded to the ovs-vswitchd by the datapath kernel module. Ovs-switchd is also responsible for communication with the network controller using the OpenFlow protocol, maintaining OpenFlow tables, and also matching packets received by the datapath kernel against these OpenFlow tables translating the matched flows into instruction for the datapath kernel module. For configuring Open vSwitch, it is necessary to use ovsdb-server, a lightweight database queried by ovs-switchd for obtaining its configuration. OVS-server is part of the OVSDB protocol that, although being part of Open vSwitch in the beginning, has been open sourced and implemented by multiple vendors worldwide, used primarily as a method for accessing the Open vSwitch remotely from the network controller [7, 8].

2.4 Atomix

Atomix is a event driven Java framework that uses a wide variety of distributed system protocols for achieving and coordinating fault-tolerant distributed systems. Its use cases are quite broad, taking into account that Atomix does not really understand the problems it solves and limits itself to providing required primitives used for solving those problems. The provided primitives are related to distributed data structures (such as maps, trees or counters), communication (such as direct or publish/subscribe), coordination (locks, leader elections, semaphores) or group membership. The primitives can be replicated using one of the available distributed systems protocols such as Multi-raft, Multi-primary, Anti-entropy and Conflict-free Replicated Data Type (CRDT). Atomix was primarily developed for its use alongside ONOS controller, and according to the former's documentation, it represents an excellent use case for all the previously outlined features. ONOS controller's core, can use Atomix for group membership and messaging, as well as cluster management and communication. Thanks to Atomix, ONOS has the possibility of using the right tool for the right job, a milestone reached through the encapsulation of most of the complexity in the distributed systems by using the aforementioned primitives [10,

41].

2.5 MQTT

Message Queuing Telemetry Transport, also known as MQTT, is a relatively simple and lightweight messaging protocol designed for scenarios presenting either constrained devices, limited bandwidth, or high-latency. MQTT emphasizes the separation of data producers (publishers) and data consumers (subscribers) through the use of topics, also in the reduction of bandwidth and device resource consumption, while at the same time providing reliability and delivery assurance to a certain point. These features make this protocol quite suitable for the upcoming machine-to-machine and IoT solutions. MQTT clients are the entities that publish or subscribe to a topic in order to publish or receive a message [36]. A MQTT server, also known as broker, is the entity in charge of receiving topic subscriptions from clients, as well as receiving and forwarding messages from or towards clients based on their topic subscriptions. The standard port which MQTT runs on is 1883, with the port 8883 being used when Transport Layer Security (TLS) is activated. Mosquitto is the MQTT message broker selected for this work. It implements the MQTT protocol versions 5.0, 3.1.1 and 3.1. A Mosquitto broker can be implemented using Python, although a C library is also available for client creation [11, 42].

3 Orchestrator implementation

The testbed for the orchestrator proof-of-concept was implemented as three interconnected virtual machines running on a Linux server located at VTT Espoo premises and having Ubuntu 18.04 LTS as host operating system. Each of the aforementioned virtual entities plays a different role: SD-WAN network simulation, Kubernetes master node and Kubernetes worker node. This chapter will give a complete in depth description of every entity forming the testbed, taking into account the complexity of the final system.

First, we have the SD-WAN network simulation virtual machine. This entity contains the OpenFlow speaking SDN controller as well as all the Mininet simulated gateways and hosts, providing an underlying physical network. Gateways are connected to each other through a set of Open vSwitch virtual switches that emulate the internet, the protocol used for the communication between these gateways is Border Gateway Protocol (BGP), which was implemented by using Quagga [15]. Each of the gateways represents a corporation's branch office, which is perfectly capable of hosting a Kubernetes master or worker node and the services deployed on them. The SDN controller runs on ONOS [14], for this thesis the version of ONOS used is 14. ONOS controller is deployed by using Docker alongside ATOMIX for supporting the creation of extra ONOS instances, effectively forming a cluster [10]. ATOMIX is a tool for creating fault-tolerant distributed systems and is maintained as a core feature of ONOS. In Figure 7, a cluster of three ATOMIX instances are connected to one ONOS node. ONOS is a controller written in Java and offers high modularity in the form of a wide variety of applications that can be activated depending on the developer's needs [14].

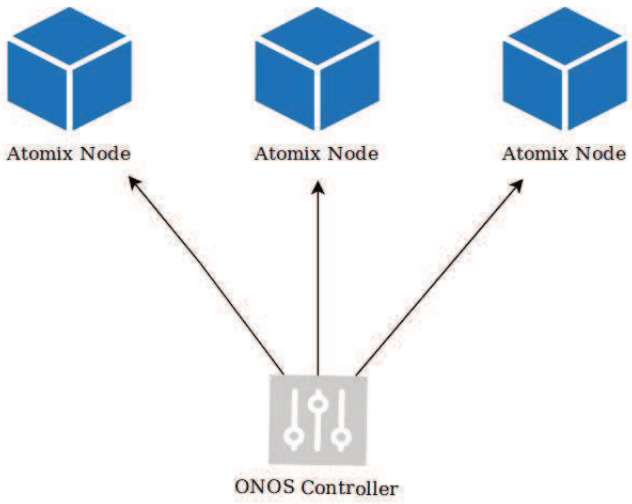

Figure 7: Basic ATOMIX architecture for one ONOS node.

The SDN controller runs on the SD-WAN network simulation virtual machine and the gateways communicate with it through the main BGP speaker that is located at the main office. A BGP speaker is a gateway that is able to understand the BGP protocol. An ATOMIX cluster consisting of three nodes is used for relieving the ONOS instances from cluster management, service discovery and data storage functions. The created ATOMIX cluster is configured through a JSON configuration file describing each constituent node. This configuration file includes information regarding each node's discovery and communication methods, management partition configuration and storage and replication partition configuration. In this work, the discovery protocol specified is Raft. ONOS entities are not listed during the discovery configuration due to the connection between ATOMIX and ONOS being of a client-server type. Raft was also used in ONOS's former releases for cluster formation, however, it requires strict cluster membership information in order to successfully form a cluster. With the adoption of ATOMIX as a separate cluster using Raft, all the ONOS nodes can easily discover peers by using dynamic discovery mechanisms, supporting the failure of all but one node [10, 41].

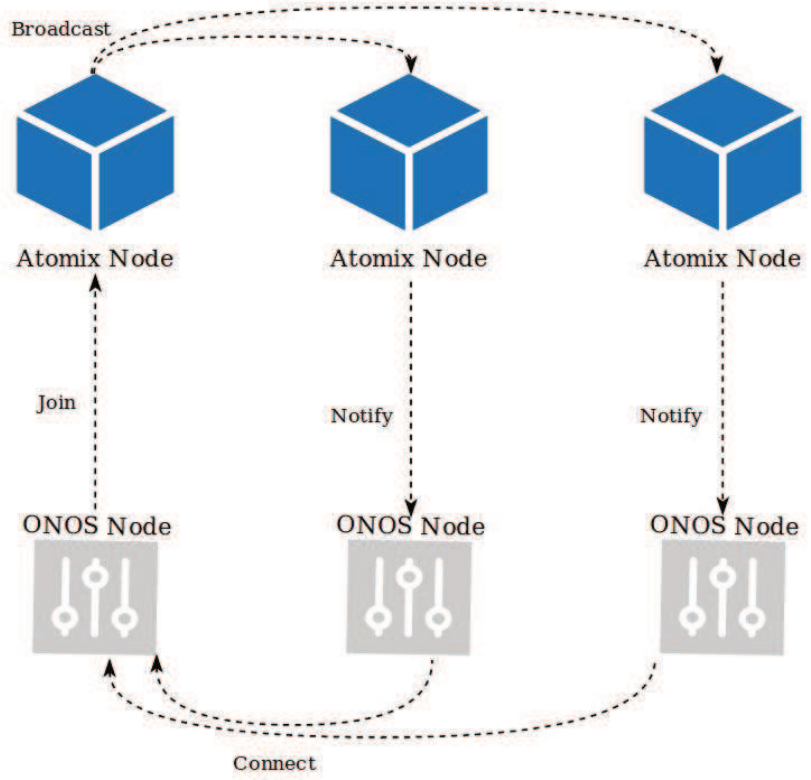

Figure 8: ATOMIX architecture for three ONOS nodes.

In order to successfully form an ONOS cluster, every single ONOS entity that is started will connect to the ATOMIX cluster. After a successful session start, the ONOS entity will notify the ATOMIX node of its existence, this node will proceed

to broadcast information regarding the new entity to all the other connected entities [41]. After receiving the information, the other ONOS entities will connect directly to the new entity in order to form a peer-to-peer connection. Each of the external gateways are preconfigured with the Quagga configuration loaded during Mininet's initialization. The configuration files are inherent to every one of the gateways, containing the router ID and the networks attached to it. It must be taken into account that if a worker node is attached to a gateway, then it shall as well advertise the worker's subnetwork.

In order for the controller to be able to receive the BGP advertisements, a JSON network configuration file is required. This configuration file possesses two main parts, ports section and BGP speakers section. The ports section is meant to configure every single interface that is connected to a BGP gateway, the information contained are the IP and MAC addresses as well as VLAN tag, if any. These ports represent the interfaces of the Open vSwitch virtual routers and this process can be analogous to interface setting up in a Linux environment. BGP speakers section is meant to add the required configuration per BGP speaker. Each speaker has an interconnection point as well as a list of peers' addresses the speaker will peer with. For each peer address configured in this section, a corresponding address in the ports side should have also been configured [14]. A complete view of the resulting network can be appreciated in Figure 9.

Next, we have the Kubernetes master node virtual machine. The master node is connected to one of the gateways through a Linux bridge created for this sole purpose. The virtual interface located on the SD-WAN virtual machine is loaded to Mininet, simulating a direct connection. In order to successfully deploy the master node, kubeadm, kubelet and kubectl alongside Docker must be installed in the virtual machine. The master node is not a single entity, it is the result of a combination of a group of pods, each of them having a specific function. Each of the pods that form the master will have one or more containers that are created using Docker, among them, the Container Network Interface (cni). The cni is the entity providing an IP address to every single one of the created pods and is also in charge of networking for the whole cluster, including the internal DNS service. Once the master node is ready, the gateway to which it is attached will serve as a gateway for both a host machine and the master node. Host machines are able to connect remotely to the master and create pods, deployments or expose services, although this is not straightforward. The security certificates must be copied to the host machine first, among the copied files there is a configuration archive containing the master's IP address and port number, allowing kubectl command to know the right destination of its queries. Due to security reasons, the master node will not be scheduled any pod and only a limited, selected number of hosts are able to access the cluster to deploy or delete services.

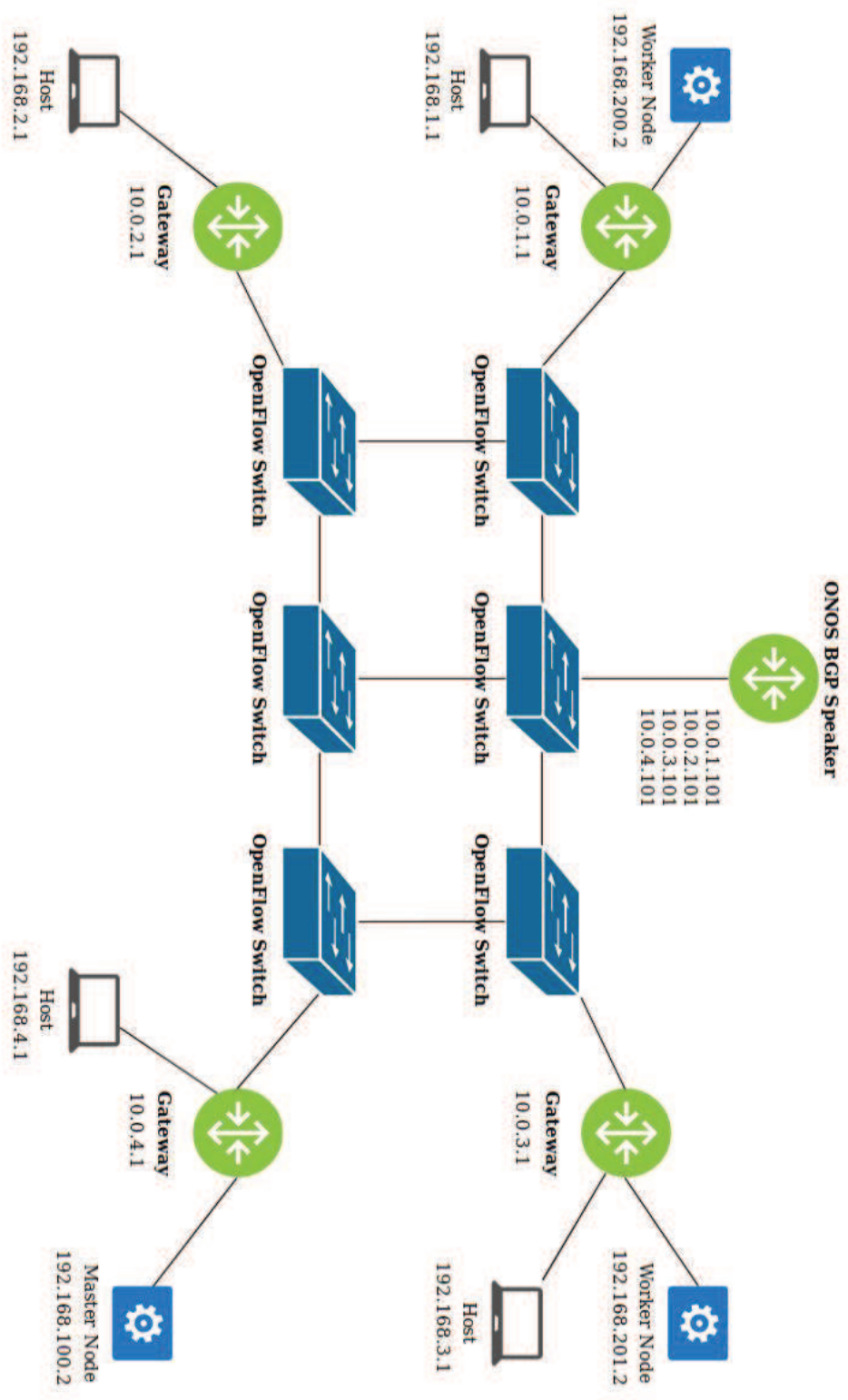

Figure 9: Complete topology of the implementation.

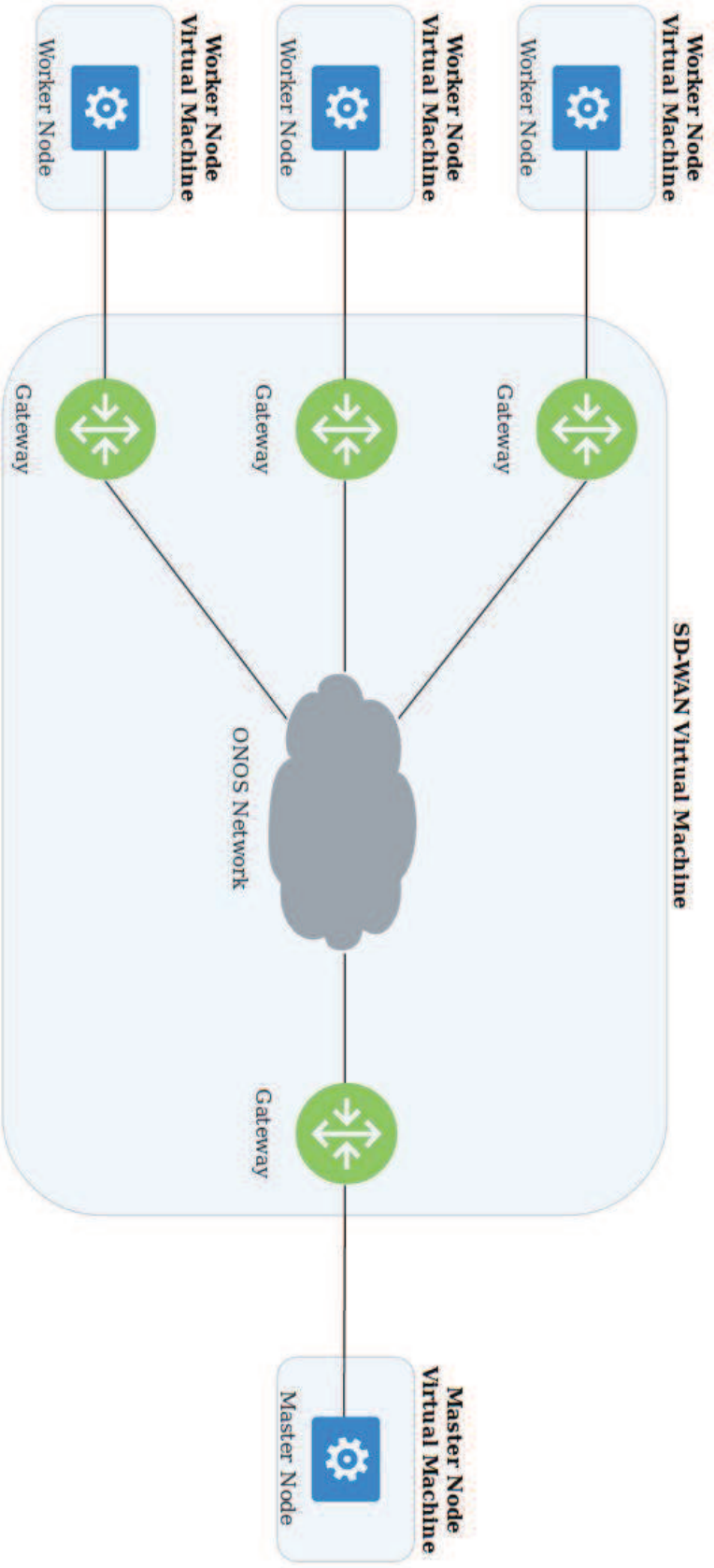

Figure 10: High level network overview representing the different virtual machines.

Finally, we have the Kubernetes worker node virtual machine. In the same way as the master node, the worker node is attached to a gateway through a Linux bridge and the corresponding virtual interface in the SD-WAN virtual machine is also loaded to Mininet. Configuration required for this node is quite minimalist in comparison with what is required for the master node, although kubeadm, kubelet, kubectl and Docker must also be installed. At least in the beginning, the worker node will not run as many pods as the master node, as most of the required services are handled by the master node. A pod can be scheduled to a specific worker node through label selectors, allowing a better resource usage, taking into account that services are not created in the master node, but in the worker nodes. A label is a key/value pair that is attached to objects such as nodes or pods and can be used to identify them. Worker node's pods are also assigned an IP address inside the specified pod-cidr-range by the cni. In this work, the separation of the conforming entities into different virtual machines, was done with the sole purpose of increasing the isolation between running software, especially for avoiding conflicts between Kubernetes and Docker. Considering that the ONOS version is a dockerized one, any issue affecting the performance of Docker would definitely hinder the effort carried out while building the proof-of-concept testbed. In the following sections, the external services that are part of the orchestrator will be introduced. The external services were conceived as a counterpart of Kubernetes internal services performing quite similar tasks but oriented to the external requests. If the external services were conceived as part of the Kubernetes cluster then extra containers would have been needed for their deployment, consuming more resources. Also, this would have required extra configuration in the Kubernetes cluster as well as for the end users. In Figure 10 we can observe the virtual machines network overview.

3.1 Domain Name System (DNS)

Kubernetes schedules a DNS pod and service in the master node with the purpose of serving individual containers by resolving a DNS name to its corresponding IP address, therefore directing their requests to the proper node. When a service is created in the cluster, it is assigned a DNS name, which will be used by a client pod during its queries in the client's pod namespace as well as in the cluster's default domain. As an example of the DNS principles in Kubernetes, we can imagine a service called "hello-world" scheduled in the namespace "kuber-system". A query coming from a pod also located in "kuber-system" must only ask for hello-world. On the other hand, a pod running in the namespace "test-system" must look up for the service with a query for hello-world.kuber-system [12]. This scenario represents the behaviour of the internal DNS, this means that only entities belonging to the Kubernetes cluster will be able to take advantage of it. In the proposed smart branch scenario, most of the requests will come from hosts located outside the cluster, they can not make use of this internal DNS service. To connect to services from outside the cluster, Kubernetes offers three solutions: accessing services through a public

IP, accessing services through the Proxy Verb, and accessing the services from a node or pod in the cluster. The first option requires the use of the NodePort or a LoadBalancer service type, the service will be exposed either on the Internet or limited to a corporate network. Its limitations are due to the fact that a request to the service will be performed using the syntax <master/worker node ip>:NodePort, making it necessary for the end user to know the node's IP address. A query sent to the master node will produce another request heading from the master node towards the worker, creating unnecessary overhead [13]. Next, we have the Proxy Verb. This solution works exclusively for HTTP/HTTPS services and may cause issues with some web services. It also performs some authentication and authorization at apiserver level before granting access to the service. The last option is to access a service using a pod or node.

It must be taken into account that although some nodes or pods might be accessed in this way, this is a non standard method, and the environment varies depending on the host, some tools might or might not be installed. Neither of the aforementioned methods is viable from the end user's perspective, being either too complex or posing a security risk for the company when exposing IP addresses of nodes containing vital services [13]. To overcome the aforementioned limitations, an external DNS service, written in Python, was conceived. Every gateway in the SD-WAN virtual machine will run their own DNS server, the service is bound to the interface heading towards the host subnetwork and listening on port 53. The server will load the zones from a .txt file containing the zone entries regarding all the available Kubernetes worker nodes. The DNS names for worker nodes have been formed by adding the node's name and a predetermined suffix. Hosts will access the available services located at the closest node under the entry "vtt.kubernetes.services", the remote non-local available services will have entries that correspond to their respective worker node name followed by the suffix ".kubernetes.services". As an example, let us assume a cluster with two worker nodes worker1 and worker2. For hosts located closer to worker1, the zone file will be as shown is Figure 11, whereas for those closer to worker2, Figure 12 shows the corresponding entry.

```
# Zones entries for external DNS service
#

vtt.kubernetes.services A 192.168.200.2
worker2.kubernetes.services A 192.168.201.2

#
#
```

Figure 11: Custom DNS entries for hosts close to worker1.

```
# Zones entries for external DNS service
#

vtt.kubernetes.services A 192.168.201.2
worker1.kubernetes.services A 192.168.200.2

#
#
```

Figure 12: Custom DNS entries for hosts close to worker2.

By making use of this service, whatever host is connected to the corporate network and close to worker1 shall be able to access the services located in the node by making a request to <vtt.kubernetes.services>:<NodePort>, and services in worker2 by using <worker2.kubernetes.services>:<NodePort> saving efforts of sharing the IP address of any node in the cluster or limiting access for only certain type of traffic. When a host located behind one of the gateways sends a query for certain service, the request will not go to the master node, but instead will go directly to the worker node running the service, as it can be appreciated in Figure 13, where the dashed lines represent a normal request, going through the master node. The continuous lines represent a direct request, enabled by the orchestrator, performed towards the worker node. This approach avoids the overhead of sending a request to the master and it sending another request to the worker node on behalf of the host. A python based DNS server was preferred at this stage due to the simplicity of using a .txt file for loading zones, being able to format the zones at will, and the easiness of making changes and binding the server to any IP address or interface without the need to install, stop or restart a service. However, solutions like bind 9 would definitely be a preferred option on a production environment.

The external DNS service allows the sending of requests towards the closest worker node. Under normal circumstances, the requests would be directed towards the central office with the client being in charge of acquire the corresponding IP address and NodePort. Due to the ephemeral nature of pods and the services they host, it is quite common that both pods and services are susceptible to suffer alterations. This alteration can be in the form of a change in their name, their location, or the creation of new services. This will lead to a change of the NodePort forcing the user to acquire the new NodePort value or not being able to perform requests. For this reason, and also taking security into account, a reverse proxy service was conceived in order to eliminate the need of using the NodePort when accessing services, as well as for providing other “user-friendly” features that will help with service access. In the next section, the implementation of this solution will be described.

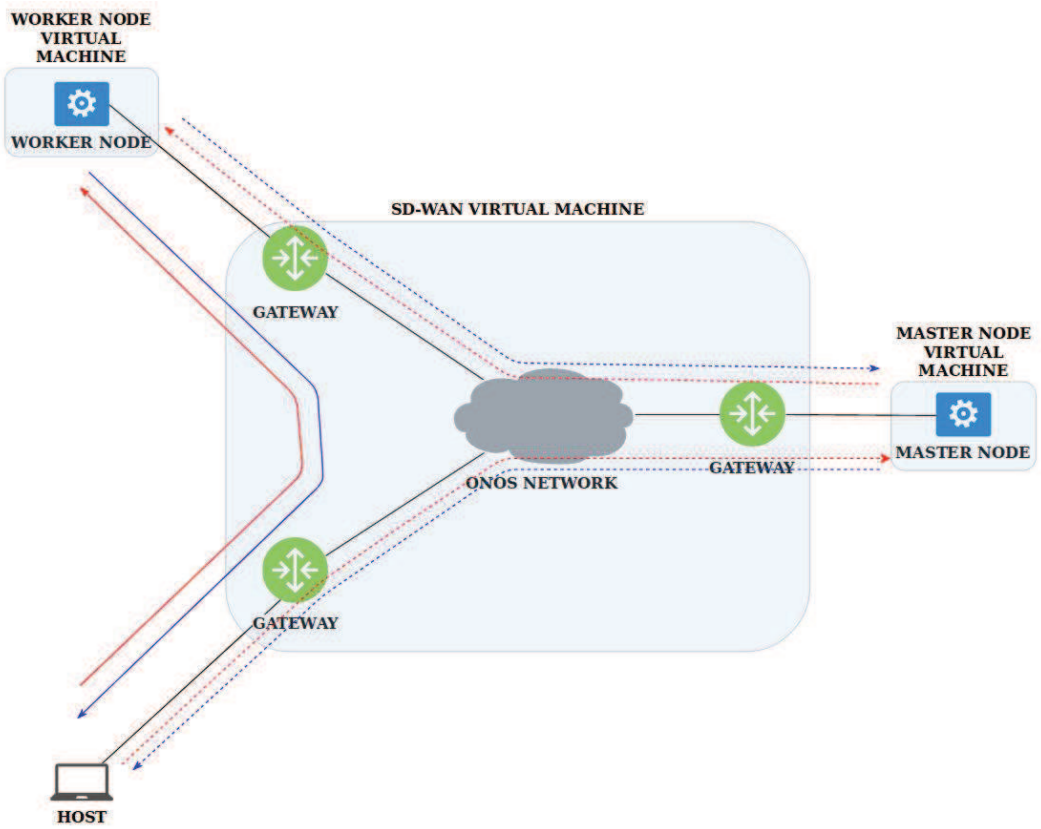

Figure 13: Requests sent by a host.

3.2 Reverse proxy service

Traditionally, when putting an application server on a network, attackers may exploit the underlying vulnerabilities of the available services. Although this is not the case when using containerized services, security is still something we all must be concerned about. In production environments, a security measure is to deny Internet access into a corporate branch and instead use a proxy server. A proxy is a server that attends requests from a web browser and it can be used to bypass security restrictions. On the other hand, a reverse proxy service is used by a web server in order to distribute application or network traffic among servers, a process known as load-balancing. Dockerized Nginx is an open source solution web server used in this work due to its user-friendly configuration and the ability of handling a great number of connections with a significantly less overhead than its alternatives. Nginx can be used as a reverse proxy, load balancer, mail proxy and even HTTP cache. The idea behind this implementation is reducing to the minimum the amount of requirements needed for running the worker nodes but installing Nginx on them would have for sure undermined this principle as every worker joining the cluster would need to have Nginx installed before being able to serve its purpose.

On the other hand, a normal Kubernetes nginx service running in the cluster

would have been limited to internal requests due the lack of an external-ip not being granted to bare metal Kubernetes load-balancers, and which only work with Kubernetes implementations running on IaaS such as GCP, AWS, or Azure. MetalLB [16] is the load balancer-implementation selected for supporting the reverse proxy service, and enables a layer 2 load-balancing through the creation of a controller and speaker deployments on every node it is running. Nginx entities are deployed one per worker node on top of MetalLB receiving the worker node's IP address as their external-ip, enabling a reverse proxy behavior for requests coming from outside hosts towards port 80 and creating a corresponding Nginx pod. The need for the NodePort on the end user's side has been avoided. Instead of this, the "location" command in Nginx is being used to redirect users to the right service based on service's name. As an example, let us consider the Nginx configuration file for a worker node called worker1 that is running a service called "hello-world" and a worker node called worker2 running a service called "my-app".

From the point of view of worker1 and as explained in section 3.1, a host close to worker1 will use the entry <vtt.kubernetes.services> for accessing services located in worker1, and an entry in the form <worker2.kubernetes.services> for all the other remote nodes, in this case a node called worker2. For avoiding the usage of NodePort corresponding to "hello-world" service, this Nginx configuration file will enable the adding of "/hello-world/" to the requested URL for accessing this service, via the location command. For a host whose closest node is worker1, the request's URL is now in the form "vtt.kubernetes.services/hello-world/" when accessing this service located in worker1. For accessing the service "my-app" in worker2, the request's URL would be "worker2.kubernetes.services/my-app/". Figure 14 shows the example Nginx configuration file for worker1.

```
server {
   listen 80;
   location /help/ {
      index help.html;
      alias /etc/nginx/conf.d/;
   }
   location /hello-world/ {
      if ( $host ~ ".kubernetes.services" ) {
         proxy_pass http://192.168.200.2:34567;
      }
   }
}
```

Figure 14: Nginx configuration file for worker1.

By adding a comparison including the URL of the request to contain the string ".kubernetes.services" the access from remote hosts to the service is guaranteed. Hosts might not know what services are available or the names of the remote worker nodes. Therefore, a request heading towards "vtt.kubernetes.services/help/" will deploy a HTML list of available services in the closest as well as in the remote nodes and their

corresponding URI. As it has been mentioned before, pods and services are not static, they are prone to being deleted or changed. Because of this, the Nginx configuration files located in every worker node must be updated dynamically as services are being added or deleted, and the Nginx service in its corresponding pod must be reloaded when these changes occur in its worker node. This is possible thanks to the service discovery implemented at the master node and that is introduced in the next section.

3.3 Master node service discovery

As explained in the previous sections, the zone files used in the external DNS service as well as the Nginx configuration files cannot be static. A master node service discovery is therefore necessary for the continuous update of all zone files in the gateways running the external DNS service, as well as for the service location updates in the Nginx pod running in every worker node. The service discovery works based on the principle that deployment and services' pods are not created at the master node, but at worker nodes. Kubernetes does not provide a default way of associating a certain service with the scheduled worker node, therefore, knowing the pods running on a determined worker node alongside the list of all available services in the cluster provides a way to start a service-node matching. Before performing the matching, the pods output must be filtered in order to avoid cluster management related pods to be counted as services. That is, pods such as calico, Nginx or MetalLB's controller and speaker must not be included.

The discovery starts when all the available worker nodes and their corresponding IP addresses are obtained as an array using the Kubernetes API. The node array structure corresponds to a node's name followed by its IP address, therefore, worker's node names will always be located in an even index within the array, with their respective IP addresses located at the subsequent, odd position. Next, for every node in existence, we obtain the running pods and all the available services in the whole cluster.

During the first iteration and for every single node, the current amount of running pods is saved within an associative array, the next step is to create the zones files, creating and copying the Nginx configuration to the corresponding pod as well as reloading the Nginx service and sending the zones file to the respective gateways. For copying files into the Nginx pods, kubectl tool is used, thus avoiding the creation of extra communication channels between the master and the worker nodes. The service discovery supports dynamically adding new worker nodes to the cluster. Those new members will be automatically detected and after deploying a new MetalLB controller and speaker entity in the node alongside the Nginx service, the worker node will be ready for being scheduled pods. During the subsequent iterations, the number of obtained pods per worker node is compared against the values previously saved in the associative array. If there is a change in one of the values, then the discovery

service is able to identify if a service might have been added, moved or deleted. After this, it deletes the current worker node's zones file and Nginx configuration in order to create new files with updated information. The Nginx service corresponding to the worker node where a change occurred is reloaded and the zone files are sent to the respective gateways. These actions only occur in the nodes where a service was added or deleted leaving the unchanged nodes working continuously without any disruption. The service discovery was conceived in a way that no extra efforts such as copying master's certificates or installing extra software is necessary for a given worker node when joining the cluster. Only the compulsory kubeadm, kubectl, kubelet and Docker are required. Being written in bash script language, portability is assured as no modifications are required for running it in any Unix-like operating system. In Figure 15, a high level version of the discovery algorithm used for the master service discovery is shown.

```
procedure Service Discovery
begin

A ← Associative array containing nodes information
N ← Array of available nodes at the cluster

for each item j in N do
 if i_j % 2 == 0 then
   P ← List of available pods for N_j
   S ← List of available services at the cluster
   if length of A < ( length of N ) / 2 then
     A[N_j] ← Length of P
     Zones ← DNS records for hosts close to N_j
     SVC ← Proxy configuration for N_j based on P and S
     p ← Nginx pod for N_j
     send Zones to corresponding gateways
     copy SVC to corresponding p
     reload nginx service in p
   end if
 end if

 if length of A ≥ ( length of N ) / 2 then
   if A[N_j] ← Length of P then
     remove Zones
     remove SVC
     A[N_j] ← Length of P
     Zones ← DNS record for hosts close to N_j
     SVC ← Proxy configuration for N_j based on P and S
     p ← Nginx pod for N_j
     send Zones to corresponding gateways
     copy SVC to corresponding p
     reload Nginx service in p
   end if
 end if
end for
end procedure Service Discovery
```

Figure 15: Service discovery algorithm.

It is worth noting that due to actions being taken only in the worker nodes where a change has occurred, sending the DNS zones files will happen only once per

change, a detail that helps in reducing the bandwidth due to the lack of continuous advertisement. Another reason behind this behavior is the fact that the contents of a zone file are mere URLs and their corresponding IP addresses, which are not prone to change. If the addresses change, this does not happen quite often.

3.4 Service update system

The updated zones file generated at the master node must be sent to the border gateways that are running the external DNS service. For this purpose, a Mosquitto [11] broker working in bridge mode was set up on the master node, listening on port 1883. Mosquitto was selected due to it being a lightweight publish/subscribe transport protocol, its capability of coping with unreliable networks, and most important, its reduced consumption of bandwidth. A MQTT publisher was implemented using the paho-mqtt python library, and set up on the master node. This publisher reads the zones file, transforms it into an array of bytes and publishes it under a determined topic. On the other hand, the border gateways running the external DNS service have a MQTT subscriber running. They subscribe to the determined topic and save the received array of bytes as a .txt file. After the file is saved, the subscriber will reload the DNS service running in the worker node. By default, MQTT does not provide encryption, however, security can be enforced by using an username/password scheme or certificate authentication using the TLS protocol, with the latter being the most practical and secure option. The usage of the TLS protocol in MQTT requires the creation of the respective key pairs and certificates for both the broker and the clients.

4 Results and discussion

The orchestrator developed as part of this thesis serves as a basic proof-of-concept for enabling edge computing in SD-WAN scenarios. The main achievement is to demonstrate that its possible to develop in-house code that eases the routing of traffic towards a Kubernetes cluster. In this chapter, the proposed solution is validated, its possible limitations discussed and future research alongside some development is proposed. First, a comparison will be performed. It must be taken into account that as this implementation consists of in-house code and due to available commercial solutions offering this kind of service being non-existent at the moment this document is being written, the comparison will be done with its counterpart system inside Kubernetes clusters, kube-dns and kube-proxy. Next, some practical measurements will also be presented and discussed. Finally, special system requirements and its limitations will also be discussed.

4.1 Comparison

Three major components of the proof-of-concept are the external DNS running on every gateway in the network, the Proxy service that is deployed in all the Kubernetes worker nodes and the service discovery that runs on the Kubernetes master node. These entities have their corresponding counterparts called kube-dns and kube-proxy. Although kube-dns and kube-proxy perform tasks only inside the Kubernetes cluster, their behavior is similar to those used in the proof-of-concept and therefore, can be used to perform a validation. This section will include a deep analysis on the behavior of the internal kube-dns and kube-proxy as well as a behavioral comparison with the external solutions conceived for this work. In Table 1, a comparison between the external solution developed in the proof-of-concept and the internal services existing in Kubernetes is performed.

Table 1: Comparison between the proof-of-concept and Kubernetes.

In-house solution		Kubernetes	
DNS	Yes	DNS	Yes
Proxy service	Yes	Proxy service	Yes
Service Association	Yes	Service Association	No

4.1.1 DNS service

The Kubernetes internal DNS service helps resolving IP addresses by performing a map of the name of a certain service to its IP address, therefore easing the finding of services by other pods. Just like a real DNS service, the domain names used in Kubernetes must be unique. In most of the cases, Kubernetes automatically starts the internal DNS service in order to offer a lightweight service discovery feature. Enabling DNS based service discovery in a Kubernetes cluster definitely facilitates for applications to find and work with each other, even when a service has been changed, moved or deleted. When an internal DNS service runs in a cluster, it does it in the following way: a service named coreDNS is started and one pod per node is created, the DNS service listens for service associated events through the Kubernetes API in order to keep its record updated, these events happen every time a service or a pod associated to it is created, changed or deleted. Kubelet sets the resolv.conf nameserver option to the coreDNS IP address with the corresponding search option that will allow the use of short hostnames. CoreDNS support the creation of two types of DNS records, A and SVC. The normal A records for services are formed in the following way: <service.namespace.svc.cluster.local>. In the same way, an entry for a pod will include its IP address like: <1.2.3.4.namespace.pod.cluster.local>. The SVC records created look like the following: <_port-name._protocol.svc.cluster.local>. This will lead to the creation of a consistent DNS-based discovery service that will enable the communication between applications and pods in the cluster.

The external DNS service implemented for the proof-of-concept supports the dynamic update of the records via MQTT. In the same way, the external DNS must be present in every single worker node for efficiently allowing a complete update. However, the main focus of the external DNS heavily differs from the internal Kubernetes DNS. While the internal DNS provides translation for services or pods, the external service provides translation at node level, this means that each of the entries in the zones files correspond to a Kubernetes node, not to a service nor a pod. Modifying the structure of the DNS entries is quite simple thanks to them being in a .txt format, thus improving automation. Currently, the external DNS service does not provide support for SVC entries due to their compatibility not being universal and therefore, not present in all networks. A quite complete solution for the implementation of a DNS server is bind 9. However, the use of it implies its installation as well as extra configuration on the gateways, which might not be useful in a developing environment. It also requires the gateways hardware to be able to support the deployment of daemons heavier than, in this case, Python. For a production environment, the use of bind 9 like solutions is encouraged.

4.1.2 Proxy service

Kubernetes internal Proxy service follows a concept quite close to a reverse proxy. Its main task is to watch for the client requests heading towards a certain IP address and port, and forwarding them to the corresponding service or application inside the cluster. From a certain point of view, the only difference between the internal proxy and a normal proxy server is the fact that Kubernetes' internal proxy will perform forwarding towards a service and its corresponding pod, not to a host. A very important characteristic of Kubernetes services' is that at the moment of their creation, the system will automatically assign a "Virtual IP address" to this new service.

This IP address is known as virtual because there is not interface nor MAC associated with it. Therefore the network does not know how to route the packets going towards it. In other words, the internal proxy service will forward requests performed by other applications or pods towards the backend pods that are managed by the required services while translating the virtual IP addresses of the services into IP addresses of the corresponding pods, thus making it not necessary for the entity making the request to actually know what pod is behind a determined service. In order to know how to route traffic from the assigned virtual IP to the corresponding pod, the proxy service interacts with netfilter and iptables, a pair of Linux configuration tools that help in the creation of forwarding rules for the virtual IPs [45].

Kubernetes is a quite complex system, although some networking tasks such as load balancing or specifying some packet rules are directly performed at userspace level, the internal proxy will often have to switch between the userpace and kernelspace in order to be able to interact with iptables and performing the load balancing, thus behaving as a reverse proxy. The process of forwarding the traffic between the virtual IPs and the corresponding pods is performed in four steps. First, the kube-proxy is permanently watching for the modification, creation or deletion of a service and their respective pods.

Next, If a new service is created, kube-proxy will open a random port on the node in order to forward any incoming connection on the port towards the corresponding pod. The pod that will be used is chosen based on the SessionAffinity option under the Spec parameter in the service configuration. After this, kube-proxy will install the iptables rules that will redirect traffic going to the virtual IPs and the service port towards the proxy port that was opened in the last step. Finally, the incoming traffic in the proxy port is forwarded towards the existing pods using a round robin schema. This involves a lot of switching between the userspace and kernelspace as kube-proxy has go to kernelspace for the virtual IPs redirection and then switch back to userspace for performing the load balancing. A general overview of the internal proxy is shown in Figure 16.

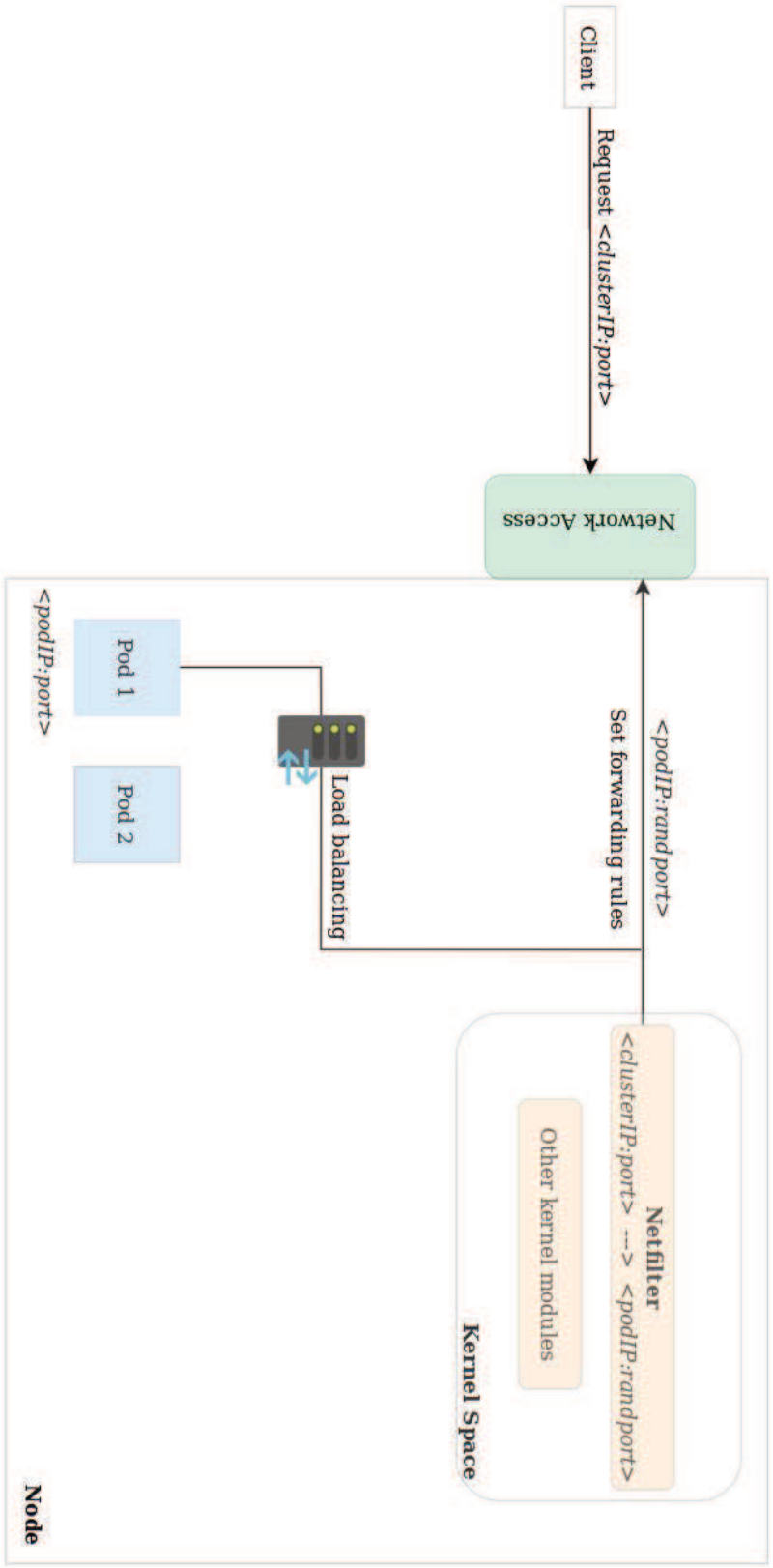

Figure 16: Kubernetes internal proxy behavior.

In the same way as the internal Kubernetes proxy, the proof-of-concept's reverse proxy implementation runs on every available worker node in the cluster and is automatically updated as new services are modified, created or deleted. However, in order to avoid the re-sending of requests between nodes, the external proxy service serves only locally available services, based on the work of the master node service discovery that helps with the association of services and nodes. Although it seems that the functionalities of the internal and external proxy servers are overlapping each other, in reality, the external proxy service works based on the NodePort created after exposing a service outside the cluster, and it needs the internal proxy service for achieving its purpose.

In the scenario where communication is performed only inside the cluster, a external proxy service will not be necessary, but when most, if not all of the requests come from outside the cluster, then an external proxy service comes in handy. Any service can be accessed either from inside or outside the cluster using the NodePort, however, Kubernetes opens a NodePort on all existing worker nodes every time a service is exposed, therefore a request for certain service heading towards a node where the backend pod is not located, will cause kube-proxy to route it towards the right node. With the external proxy service running, the internal Kubernetes proxy will only receive requests that can be processed locally, in other words, requests asking for services whose backend pods are located in the node. It is possible to implement a reverse proxy using Python, but the proof-of-concept's implementation uses the widely known Nginx running on top of a load balancer in order to achieve the desired behavior.

4.1.3 Master service discovery

As explained in the previous sections, Kubernetes performs a service discovery based on its internal DNS service and proxy service. Although not completely efficient due to the multiple request redirections occurring between the worker nodes, it gets things working in a fairly reliable way. When it comes to the implementation of external in-house applications such as the DNS and proxy service developed in this work, Kubernetes does not provide a way to associate existing services with its backend pods and the corresponding worker node in order to enable direct requests. The master service discovery developed for the proof-of-concept provides a method for associating the existing services with their backend pods, by assuming the service belongs to the node where its corresponding pod was scheduled. This has proven to be quite effective with the condition that pods have a name that is related to the service they are providing.

4.2 Testing

The proof-of-concept was tested in the same environment it was developed. The testing topology corresponds to that of Figure 9, with all the developed services running in their corresponding locations, the Kubernetes master and worker nodes attached to their respective gateways running all the needed pods for proper functioning, such as coreDNS and kube-proxy, among others. The gateways run their custom DNS servers that are bound to their specific interface, the worker nodes run the Nginx service and the master node is running the service discovery alongside its MQTT publisher and Mosquitto broker. This can be appreciated in Figure 17. As it was aforementioned, no Kubernetes-related configuration work is performed on the worker nodes as the master node service discovery is going to perform all the required tasks without needing any action by the workers.

Under the aforementioned testing considerations, it must be taken into account that Mininet's virtualization is done only at a network level, and each host process sees the same set of processes and directories, thus hindering the functions of the DNS service and the MQTT-based update system. The issue arises due to the lack of directory isolation. The update system will send the corresponding zone files to the gateways, and they will vary according to the gateway's location. Taking into account Figure 17, gateway 1 shall receive a different zones file than gateway 2 due to it being located closer to a worker node. However, this does not happen in Mininet, where the files received would be overwritten, causing the DNS service to upload the wrong zones. In the same way, the resolv.conf file that contains the DNS server's addresses must be unique per host. Otherwise, all of them will be pointing at the same DNS server causing the network to be flooded with wrong requests.

A similar situation occurs with the custom DNS server, it has to be bound to the gateway's interface that is going towards the host network, therefore, a different custom DNS service file is needed per gateway. These issues were overcome by creating the needed files in the directory /etc/netns/<host-name> containing the resolv.conf and the custom DNS files. The principle behind this is that ip-netns creates the namespaces as a logical copy of the network stack, but it inherits the whole network namespace from its parent. In the case of network namespace aware applications, a global network configuration is first looked for in the above mentioned directory and after this in /etc/. So by creating the files in /etc/netns/<host-name> we are loading them as global network configuration.

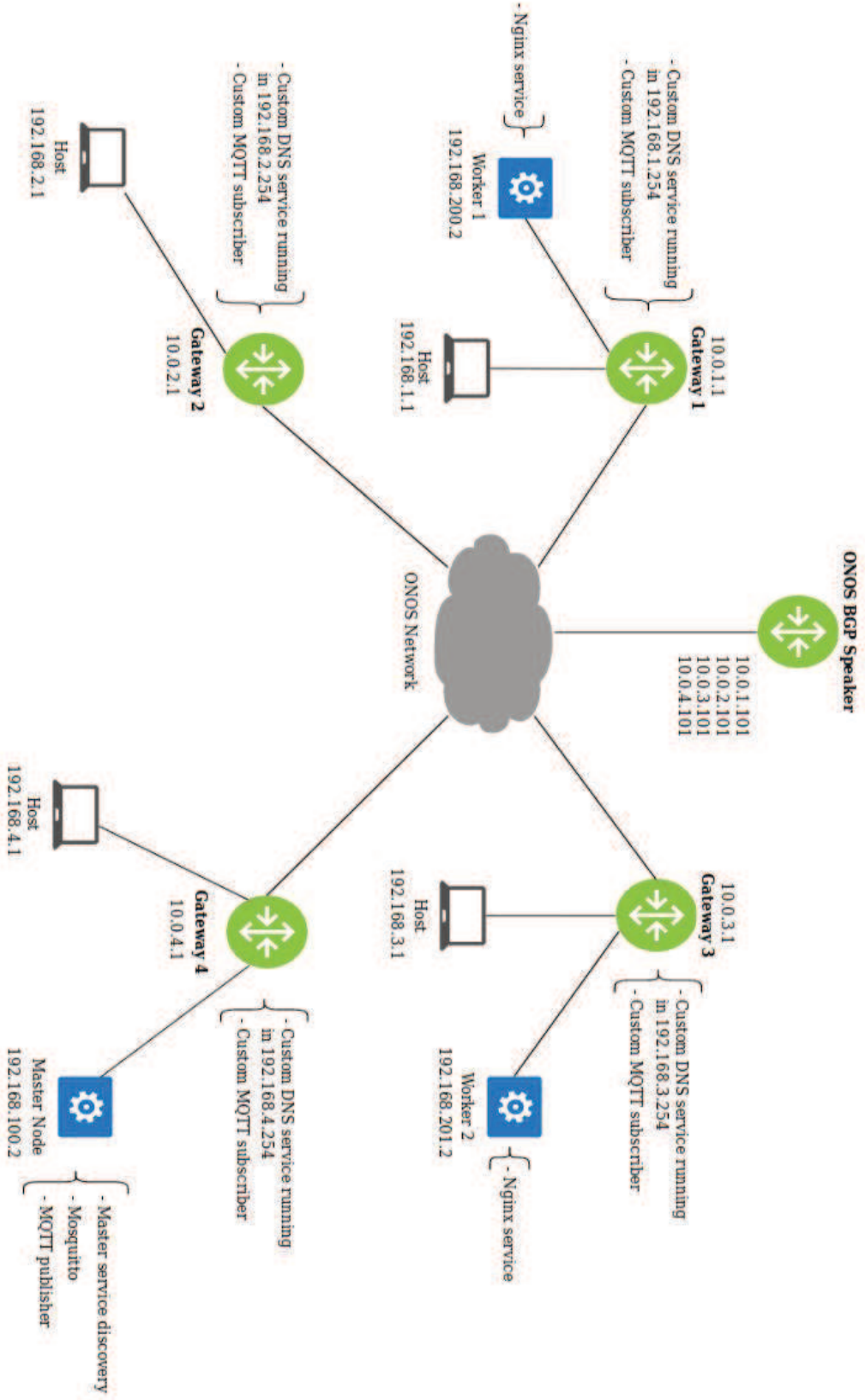

Figure 17: Testing network topology and the services running.

Taking into account that this work is based on the dynamic discovery of updated, newly created and deleted services, the measurements will focus on the time required for creating the zones files and the Nginx configuration files, as well as the time until the changes have been applied to both the DNS server and the Nginx service. The first measurement in Table 2 and Table 3 will be carried out with only one worker node while another separate measurement in Table 4 and Table 5 will be performed for two worker nodes. This in order to find out whether the number of nodes has an effect on the system's performance, and in a more realistic scenario this number could scale up until having tenths of worker nodes.

Table 2: Time required for the orchestrator to perform the necessary tasks in order to fully discover a newly created service when only one worker node is available.

One worker node – creating service	
Time to create zones files and Nginx configuration files after a change in the worker node.	2 seconds
Time for changes being available in Nginx after a change in the worker node.	3 seconds
Time for the zones being loaded in the DNS server after a change in the worker node.	3 seconds

Table 3: Time required for the orchestrator to perform the necessary tasks in order to fully eliminate a recently deleted service when only one worker node is available.

One worker node – deleting service	
Time to create zones files and Nginx configuration files after a change in the worker node.	12 seconds
Time for changes being available in Nginx after a change in the worker node.	3 seconds
Time for the zones being loaded in the DNS server after a change in the worker node.	3 seconds

Table 4: Time required for the orchestrator to perform the necessary tasks in order to fully discover a newly created service when two worker nodes are available.

Two worker nodes – creating service	
Time to create zones files and Nginx configuration files after a change in the worker node.	2 seconds
Time for changes being available in Nginx after a change in the worker node.	3 seconds
Time for the zones being loaded in the DNS server after a change in the worker node.	3 seconds

Table 5: Time required for the orchestrator to perform the necessary tasks in order to fully eliminate a recently deleted service when two worker nodes are available.

Two worker nodes – deleting service	
Time to create zones files and Nginx configuration files after a change in the worker node.	15 seconds
Time for changes being available in Nginx after a change in the worker node.	3 seconds
Time for the zones being loaded in the DNS server after a change in the worker node.	3 seconds

When measuring the availability of the services, it must be taken into account that the reload functions in the orchestrator are carried out almost simultaneously. Thus, the time for a service to be available to the end user, in Table 2 for example, is the time required to create the file plus the time required to reload the Nginx and DNS, in this case 5 seconds. From Figure 18, it can be inferred that at the moment of creating a service, the number of worker nodes available does not influence the general performance. One reason for this might be the fact that Docker containers backing those services are created only in the scheduled worker nodes. On the other hand, in Figure 19, it is possible to appreciate a slight difference in the files creation function when two or more worker nodes are present. A reason behind this behavior might be the fact that Kubernetes master node also has to delete the corresponding API object for every deleted service.

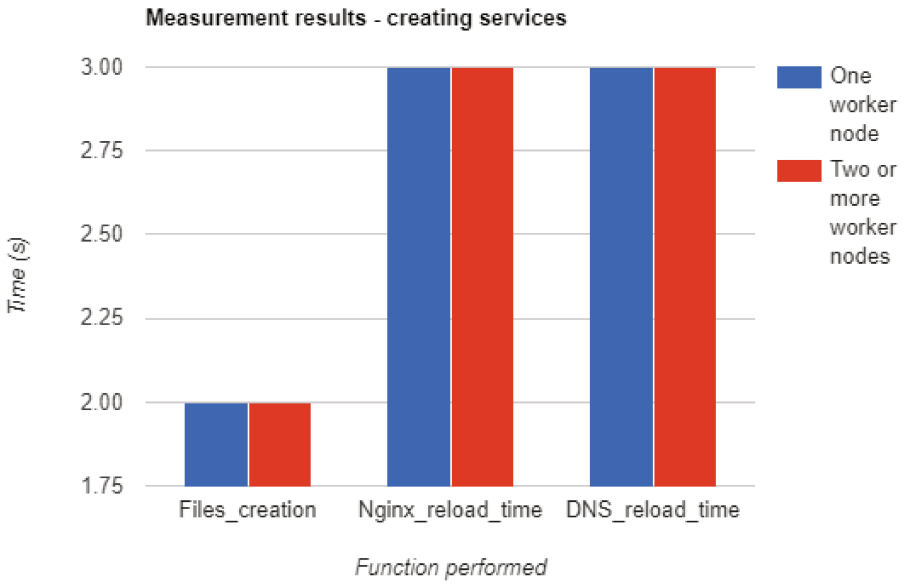

Figure 18: Time required for the orchestrator to perform the necessary tasks in order to fully discover a newly created service.

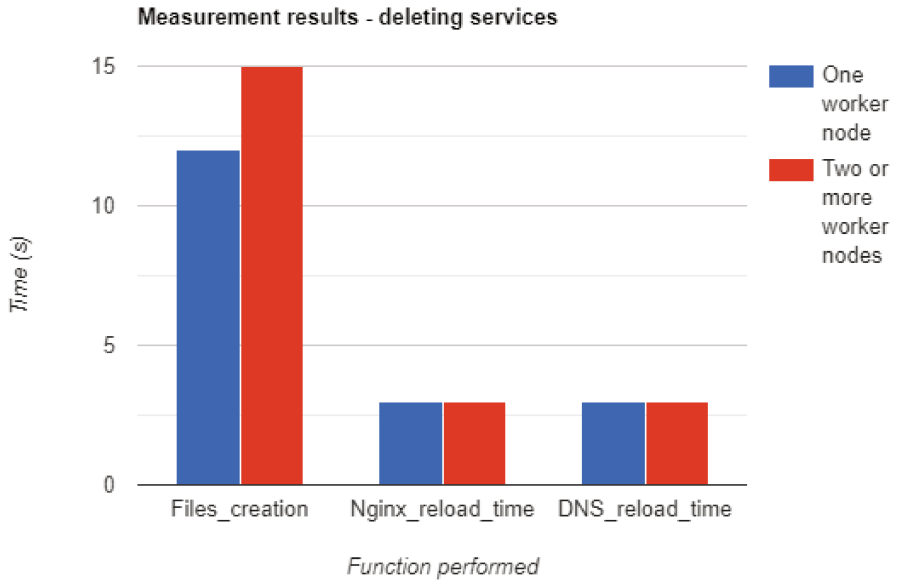

Figure 19: Time required for the orchestrator to perform the necessary tasks in order to eliminate a recently deleted service.

4.3 General performance

An important time to be taken into account is that required for the master's node to be ready. This includes the time for important system's components to start and get the corresponding IP address from the cni. This is heavily affected by the status of the required images of those components. If the images are already downloaded in the host machine then starting the master node will only take around 3 or 4 minutes. On the other hand, if no images are available or it is the first time starting the master node, then the process can easily double that amount of time. By comparison, the time required by a worker node to join the cluster can be considered negligible, as it only takes a coupe of seconds.

The deployment of pods and services shall be done in an ordered manner. This means avoiding the parallel launch of several deployments on the same worker node. Otherwise performance issues will be present even when the host system possesses copious amounts of resources, making pods that would normally start in a couple of seconds to take around 1 minute to be ready. It is assumed that the underlying network does not present any performance issue as the ping values are low, normally in the order of 0.5 milliseconds with the first ping sent having times of around 1.5 milliseconds.

4.4 Results discussion

Taking into account the nature of this work, and the fact that no commercial solutions are available for a comparison, the times presented are in the measurements are good. With such times, the creation and update of services will be almost invisible from the host's point of view. Delays can appear at various stages in the system. One of the first delays encountered is the time it takes to make a request using kubectl in the master node. The first request usually takes around 2 or 3 seconds to complete, with the subsequent requests being much faster, almost immediate. The same happens when performing a request with a JSON output, the first request will always induce a certain amount of delay. In the same way, when creating a deployment and exposing it as a service, it takes time for Kubernetes to create the new pod. This time varies depending on the available bandwidth, whether the required image has already been downloaded on the host machine or not, as well as its size, and the time it takes to the pod for being scheduled.

It takes a slightly longer time to update the files during the deletion of a service and corresponding back-end pods. This occurs because pods are granted a grace time in order to not only delete the process but also the API object. The time required for deleting a back-end pod will always be higher than the time required for creating it, even if the process running inside the pod is lightweight. One solution to this is to use the flag –wait=false when deleting the pod, though it is highly recommended

to grant this grace time in order to ensure a proper deletion. When a service is deleted and its back-end pods stay in the terminating phase for a determined period of time, no concern exists as the service will not be available anymore, and even if the Nginx service has not been updated yet, requests being forwarded towards the deleted service will not be successful.

The convergence times can also be modified by altering certain values in the elements such as the master service discovery, the MQTT subscriber running in the gateways and the MQTT publisher running in the master node. As an example, a delay can be added or reduced before restarting the Nginx service in the master service discovery script, although taking into account that the restart will happen only after performing a request for obtaining the corresponding pod's name, deleting the previous Nginx configuration file from that pod, copying the newly created configuration file and finally, restarting the service. Similarly, some delays can be added before publishing the created files in the MQTT publisher code or before saving the received files in the subscriber code. One possibility for dropping the times associated with the restart of the services, at least for Nginx, is to use the daemon-based Nginx installation in Linux, instead of the Docker-based version. This would reduce the amount of time spent to obtain the name of the Nginx pod, copying the files and restarting the service to only restarting the daemon.

4.5 Requirements and limitations

This proof-of-concept implementation has some general special requirements and considerations that must be taken into account before considering its implementation in a real-life scenario.

4.5.1 Hardware requirements

Although quite basic, Kubernetes has some system requirements needed in order to work as expected. A Kubernetes master node requires a system with at least 2 CPUs and a minimum RAM of 2 GB. On the other hand, the worker nodes have lesser requirements needing only 1 CPU and a minimum RAM of 1GB. Although some different versions of Kubernetes are available, the one used for this implementation is k8s which has the aforementioned requirements. Versions like k3s might present lesser requirements, but this was not tested in this work. With master service discovery being a fundamental part of the system, and due to the fact this is written in bash script language, a Linux operating system is the ideal platform to run the proof-of-concept. In a real hardware implementation, the worker nodes must run on a separate, bare metal server in order to avoid RAM limitations when running multiple worker nodes altogether.

4.5.2 Swap requirements

A swap space is a quite common part of a Linux system. It is used in order to increase the host's available amount of virtual memory. In other words, a swap space substitutes disk space for RAM when no RAM is available. Kubernetes requires swap to be turned off, this can be troublesome if the system does not have enough RAM, causing failure when trying to allocate memory for new requests. The reason behind the lack of swap support in Kubernetes is it is focused on providing a utilization percentage of almost 100% of a node's capacity, also, input and output in swap scale very poorly. For this reason, in a real world deployment, all the applications must include CPU and RAM limits, so when a pod is scheduled on a worker node, swap is not necessary. As an example, if a deployment requires 3 GB of RAM and the there are two nodes in the cluster, one with 2GB and another with 4 GB left, the deployment will be scheduled to the worker node with the most RAM available. In a scenario with no swap and not enough RAM, the kernel will decide to kill the high-memory processes such as MySQL or Java. From this point of view, it would not be recommendable to run memory demanding processes in the Kubernetes cluster, and if this is necessary, then enough RAM must be provided.

4.5.3 Onos requirements

ONOS has also specific requirements that will guarantee its proper working, for ONOS version 1.14, Linux 14.04 LTS is recommended alongside 2 GB or more of RAM and 2 CPUs. As for this implementation, a Linux 16.04 LTS virtual machine with 4 GB of RAM and 2 CPUs was used. However, there are no specific requirements for Atomix when used alongside ONOS, but based on this work, the minimum requirements for running ONOS 1.14 and Atomix for providing distributed backup is 4 GB of RAM and 2 CPUs. In the same way, for building ONOS the following software is needed: Java 8 JDK (Oracle Java), Apache Maven and Apache Karaf, the versions of the last two depending on the used version of ONOS.

4.6 Published work

This thesis work was published at the "8th International Conference on Cloud Computing: Services and Architecture (CLOUD 2019)" under the title "Enabling Edge Computing Using Container Orchestration and Software Defined Wide Area Networks". After some modifications, this paper was selected for the journal "International Journal of Computer Networks & Communications (IJCNC)" under the title "Containerized Services Orchestration for Edge Computing in Software Defined Wide Area Networks".

4.7 Future research

The proof-of-concept presents a working service discovery and update implementation that eases the adoption of edge computing in a SD-WAN scenario. However, some improvements are possible and in this section some future work is suggested, so the system can be ready for its implementation in a production environment.

4.7.1 Node distance computing function

A way to dynamically obtain and update the distance between a gateway and the available worker nodes is a quite important feature that would allow the MQTT subscribers located in the gateways to select the closest node available so they can receive updates regarding this specific worker node as the local node. A common misconception lies in the belief that using ping is an acceptably accurate tool for this purpose. However, IP addresses can not be characterized by geographical reasons as a determined region might or might not be assigned a certain IP addresses block to it. With this in mind, an IP address assigned to Finland may easily be announced by a device at any other country, thus not guaranteeing that an entire network has been assigned to a single geographical location [40].

Similarly, there are many limitations considering the use of ping times. A ping time is determined by a great number of factors, which include: the number of routers, or hops, between the host and the target machine, the "quality" of the routing performed between these two points, any networking issue located between the source of the request and the target, excessive traffic or congestion happening between the target and the source, different transmission mediums along the route, with copper having a different propagation time than, for example fiber or a satellite link, among other factors [40].

Results from [18] can be used for the implementation of this feature, where some coordinates-based approaches for network distance estimation are discussed. The idea behind the coordinates-based distance measurement is that hosts maintain a determined set of numbers, also known as coordinates, that are used for characterizing their locations in the network and allowing a distance prediction based on the result of a distance function run over the host's coordinates. This approach works particularly well on peer-to-peer architecture, when a host discovers another host's identity, their coordinates would be exchanged and then the distance will be computed instantly. The mentioned work points out that coordinates have proven to be quite efficient at summarizing large amounts of distance information. A concern regarding the proposed approach is related to the assumption of stability in the network, such as consistent propagation delays, if this does not hold true due to the constant changes in network topology, distance estimations will be affected.

4.7.2 Local breakout support

In order to improve the experience of users who constantly move from one corporate branch to another, a local brakeout solution results quite helpful. In simple terms, a local breakout solution will route the IP packets of the roaming hosts directly from the visited network towards the internet. This option has been available since the appearance of GPRS, but has not really been implemented in practice, instead of this, a solution called home routing is commonly used. Under a home routing approach, all the IP packets originating from roaming subscribers are tunneled back to their respective home network, and only from there they are sent towards the internet. More recently, in the 5G Roaming Architecture, local breakout is considered as the main option over home routing [37].

The implemented system is meant to provide easy access to a wide range of different applications or services, if a corporate, unmanaged network access is provided, a higher bandwidth and direct access will be provided for all available applications, although they have different requirements and need custom security considerations. Thus, this represents a possibility for developing an extra in-house local breakout service that would enable a certain degree of security based on the traffic's level of trust, this thanks to the possibility of simulating an EPC with the tool OpenEPC. For example, the proof-of-concept already allows the forwarding of traffic only towards the node that runs a certain application or service, but Deep Packet Inspection Techniques (DPI) might be used in order to identify the application on the first packet sent. After the flow's destination is established, it cannot change within the same session. This will enable the redirection of untrusted traffic towards a security gateway or firewall, while sending trusted traffic directly to the internet, allowing extra bandwidth savings by eliminating bottlenecks due to the questionable traffic being sent to extra secure security appliances. In Figure 20, a general description of the proposed improvement is shown [38].

The Packet Data Network Gateway (PDN-GW / PGW) supports the implementation of DPI on both, GTP-based and PMIP-based traffic on a per-user basis. It must be clearly stated that the implementation assumes the whole system is served by a unique ISP, so roaming will not be implemented and the local breakout will be carried out by the PGW/SGW selection performed by the SGSN according to the MME's specifications. In the case where each branch is served by a different ISP and a user moves from one branch to the other, a roaming scenario will be present at the Visited Public Land Mobile Network (V-PLMN). In this case, a control session going from the Visited-PGW towards the Visited Policy and Charging Rules Function (V-PCRF) will trigger a request to the Home Policy and Charging Rules Function (H-PCRF) in the Home Public Land Mobile Network (H-PLMN). The V-PCRF will identify the request for roaming based on the subscriber identity [38].

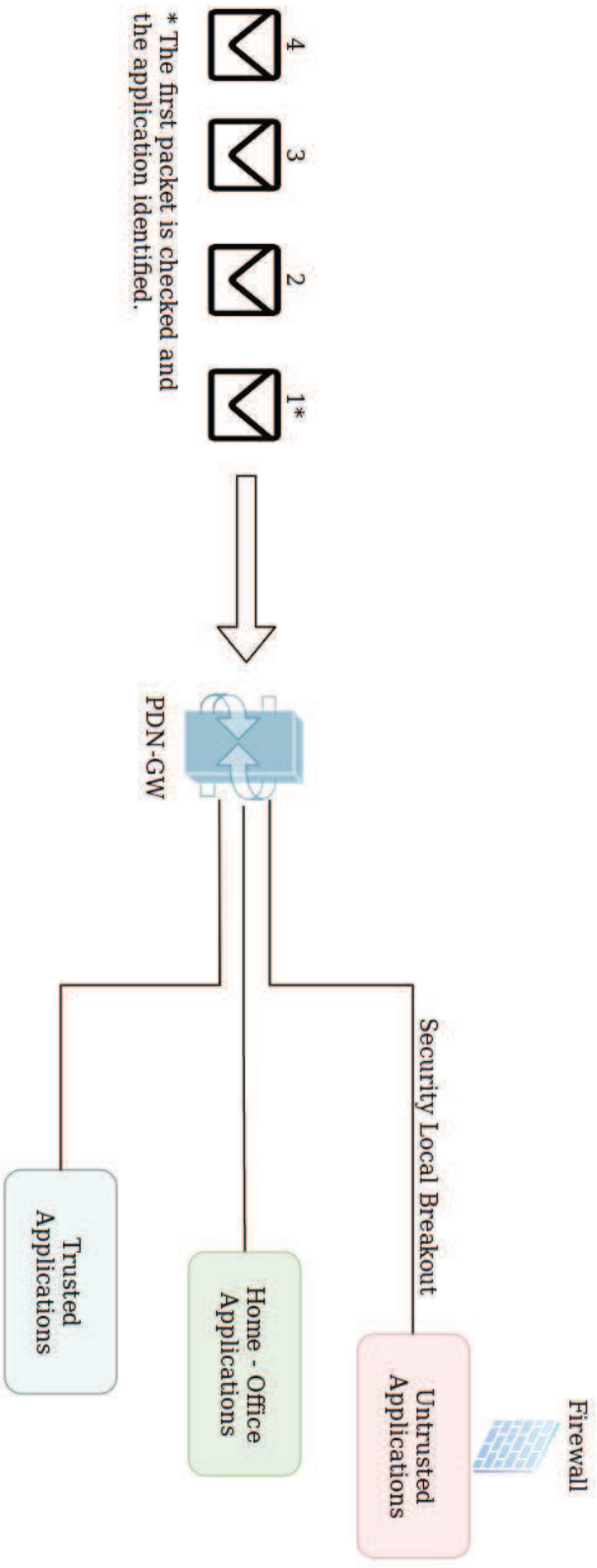

Figure 20: Overview of a possible Local Breakout solution in the terms of security.

4.7.3 Data plane programmability

Data plane programmability can be considered as the natural evolution of SDN, as it enables a much more flexible networking when being contrasted with a normal control plane based programmable network. Programming Protocol-independent Packet Processors, also known as P4, is the de-facto language for dataplane programmability, it allows several features extension of SDN networks as well as a dynamic configuration of actions that goes far beyond those allowed by the OpenFlow specifications. However, data plane programmability is not a silver bullet and although it allows to easily add new protocols, or remove unused protocols in a network chip, its effectiveness can only be appreciated at networks carrying huge amounts of traffic, therefore, some companies would not really require to implement it [39].

The proof-of-concept system can take advantage of the support of P4 by ONOS [19], by implementing some novel features that will improve the experience and manageability of the system by creating a custom P4-based Load balancer and Telemetry system. By diving into these topics, it is assumed that a real-life implementation of the proof-of-concept is meant to possess a high traffic rate.

Normally, load balancers used in the cloud data centers as well as the load balancer used for this work are software based. Software based load balancers work by mapping a virtual IP address to a direct IP address that corresponds to a server or group of servers offering the required service. The usage of software based load balancers has some drawbacks such as: high use of server resources, high delay and weak performance isolation. In [20], Miao Rui et al, demonstrate that it is possible to implement a fully functional 400 lines P4-based load balancer that can support millions of simultaneous connections while providing per-connection consistency. The same principle can be applied for developing an in-house layer 4 load balancer instead of the currently used MetalLB, bringing a higher performance, lower delay and the relief of the MetalLB related pods in all the running worker nodes, while decreasing the change of user experience degradation based on broken connections.

An in-house in-band network telemetry system is also possible to implement by using P4 as it has been demonstrated in [21]. In-band network telemetry allows data packets to query for switch internal state statistics such as link utilization and queue size. Thanks to each P4 switch having a control channel that allows the insertion, deletion and modification of matching tables, it is possible to send probe packets periodically that contain the switch ID and the specific time spent in determined switch, once the packet arrives to the end user, an almost real-time measurement of this will be processed, allowing the detection of switches having large queues. The resulting telemetry system can be used for easing the debug and diagnostics of network issues in a fast and intuitively manner.

5 Conclusions

Containerization is nowadays the most common way of deploying a service, widely used applications all run on containerized environments. As is expected, containerization offers many advantages to corporate wide applications, from those running basic web services to the more complex, edge-computing related applications. Knowing that containerized applications are prone to suffer modifications, the current work aims at providing a method that can be used for easily discovering and accessing containerized services deployed in a Kubernetes environment. However, the proof-of-concept cannot yet be efficiently implemented in a real life scenario as an increased degree of automation as well as some performance improvements must be carried out.

This proof-of-concept is able to properly run in a testing environment. Although no commercial options are available to correctly perform a comparison, quite useful features are present regarding the service discovery and the utilization of a proxy server to access the deployed services. Some improvements are needed in the performance of the service to pod association in the master service discovery application. Until now the deployed pod must have the same name as the deployed service in order to be able to identify it.

The implemented system requires servers running as gateways, as it is necessary to run the DNS service and the MQTT subscriber on them. A normal router would not be able to run these processes. This can cause an increase in the required budget but there is a substantial gain in flexibility as it would allow the system to run extra processes on the gateways, being a quite useful feature for future improvements.

DNS zones files are updated based on a predetermined subscription to a certain worker node. However, in a real scenario, the usability would be benefited if subscription is automatized based on a method specifically developed for measuring the distance from the gateway to the worker node. Such an option was already discussed in the last section, although it is worth mentioning that implementing such a solution is not an easy task, and measurements cannot be guaranteed as there are many factors influencing the delay in such a distributed network like Internet. In case of a possible implementation it shall be run as a service being executed alongside the MQTT subscriber in every existing worker node.

The application of data plane programmability that has recently been supported by ONOS controller can help to simplify the system in an incredible way. The most notorious change will be the complete removal of MetalLB alongside its corresponding pods created in every worker node from the proof-of-concept. This approach would lead to a more lightweight and less resource consuming solution. Another factor to take into account is the usage of P4 ready switches which will also improve portability as the P4 code can be run in different P4 capable switches without any change.

The proof-of-concept in its current status is quite useful for testing purposes, especially those related to the deployment of in-house applications and services. It can also be used for research based on data plane programability or the simulation of local breakout by using virtual Multi-access Edge Computing (vMEC).

References

[1] Janakiram, MSV. *Use cases for Kubernetes.* Oregon, The New Stack, 2017.

[2] Lerner, A., Rickard, N. *Technology overview for SD-WAN.* Gartner, Stanford, 2015.

[3] Horrel, J., Karimullah, A. *SD-WAN Set to Transform WAN in Australia.* IDC Custom Solutions, Framingham, 2017.

[4] Skaria, S., *Edge Computing vs Cloud Computing, Where does the future lie?.* Visited 31.01.2019. Available: https://www.linkedin.com/pulse/edge-computing-vs-cloud-where-does-future-lie-saju-skaria

[5] Yanbiao, L., Zhang, D., Taheri, J., Li, K. *SDN components and OpenFlow.* Big Data and Software Defined Networks, 2018, EIT Digital Library. ISNB: 978-1-78561-304-3. Pages: 49-65.

[6] Ranganathan, R. *A highly-available and scalable microservice architecture for access management.* Aalto University, 2018. Available at: https://aaltodoc.aalto.fi/bitstream/handle/123456789/34401/master_Ranganathan_Rajagopalan_2018.pdf?sequence=1&isAllowed=y

[7] Linux, Foundation. *Open vSwitch Documentation.* The Linux Foundation, San Francisco, 2017. Visited 15.02.2019. Available at: https://buildmedia.readthedocs.org/media/pdf/ovs-reviews/latest/ovs-reviews.pdf

[8] Langenskiöld, T. *Network Slicing using Switch Virtualization.* Aalto University, 2018. Available at: https://aaltodoc.aalto.fi/bitstream/handle/123456789/29159/master_Langenski%C3%B6ld_Thomas_2017.pdf?sequence=1&isAllowed=y

[9] Padhy, R., Patra, M., Satapathy, S. *Virtualization Techniques & Technologies: State-of-The-Art.* Journal of Global Research in Computer Science, 2018, vol. 2, nro.12. ISSN: 2229-371X. Available at: https://www.researchgate.net/publication/264884756_VIRTUALIZATION_TECHNIQUES_TECHNOLOGIES_STATE-OF-THE-ART

[10] Open Networking Foundation. *Atomix.* Open Networking Foundation. Visited 15.02.2019. Available at: https://atomix.io/docs/latest/user-manual/introduction/what-is-atomix/

[11] Stanford-Clark, A., Nipper, A. *Message Queuing Telemetry Transport (MQTT).* Organization for the Advancement of Structured Information Standards (OASIS). Visited 15.02.2019. Available at: http://mqtt.org/

[12] Kubernetes. *DNS for services and pods.* The Linux Foundation, San Francisco, 2019. Visited 15.02.2019. Available at: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

[13] Kubernetes. *Access services running on clusters.* The Linux Foundation, San Francisco, 2019. Visited 15.02.2019. Available at: kubernetes.io/docs/tasks/administer-cluster/access-cluster-services/

[14] Open Network Operating System (ONOS). *ONOS features.* Open Networking Foundation & The Linux Foundation, San Francisco, 2019. Visited 15.02.2019. Available at: https://onosproject.org/features/

[15] Jakma, P. *Quagga Routing Software Suite.* Quagga Routing Suite. Visited 15.02.2019. Available at: https://www.quagga.net/

[16] MetalLB *Metal Load-Balancer (MetalLB).* Google. Visited 15.02.2019. Available at: https://metallb.universe.tf/

[17] Eclipse Foundation. *Mosquitto, an open source MQTT client.* Eclipse Mosquitto. Visited 15.02.2019. Available at: https://mosquitto.org/

[18] Eugene, TS., Zhang, Hui. *Predicting Internet Network Distance with Coordinates-Based Approaches.* Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies, 2002, DOI: 10.1109/INFCOM.2002.1019258, ISSN: 0743-166X. Available at: https://www.cs.rice.edu/~eugeneng/papers/INFOCOM02.pdf

[19] Open Network Operating System (ONOS). *ONOS + P4 Tutorial.* Open Networking Foundation & The Linux Foundation, San Francisco, 2019. Visited 15.02.2019. Available at: https://wiki.onosproject.org/pages/viewpage.action?pageId=16122675

[20] Miao, Rui., Hongyi, Zeng., Changhoon, Kim., Jeongkeun, Lee., Minlan, Yu. *SilkRoad: Making Stateful Layer-4 Load Balancing Fast andCheap Using Switching ASICs.* Association for Computing Machinery's Special Interest Group on Data Communications (SIGCOMM), 2017, DOI: 10.1145/3098822.3098824, ISBN: 78-1-4503-4653-5/17/08. Available at: https://eastzone.bitbucket.io/paper/sigcomm17-silkroad.pdf

[21] Changhoon, Kim., Sivaraman, Anirudh., Katta, Naga., Bas, Antonin., Wobker, Lawrence J. *In-band Network Telemetry via Programmable Dataplanes.* 2015, Visited 12.05.2019. Available at: https://pdfs.semanticscholar.org/a3f1/9dc8520e2f42673be7cbd8d80cd96e3ec0c1.pdf?_ga=2.76525468.802012735.1559031914-713298922.1559031914

[22] Jarraya, Y., Madi, T., Debbabi, M. *A Survey and a Layered Taxonomy of Software Defined Networking.* IEEE Communications Surveys & Tutorials 16,1955–1980, 2014, DOI: 10.1109/COMST.2014.2320094, Visited 12.05.2019. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6805151

[23] Kreutz, D., Ramos, F.M.V., Esteves Verissimo, P., Esteve Rothenberg, C., Azodolmolky, S., Uhlig, S. *Software-Defined Networking: A Comprehensive Survey.* Proceedings of the IEEE 103,14–76, 2015, DOI: 10.1109/JPROC.2014.2371999, Visited 12.05.2019. Available at: http://ieeexplore.ieee.org/document/6994333/

[24] Mijumbi, R., Serrat, J., Gorricho, J.L., Bouten, N., De Turk, F., Boutaba, R. *Network Function Virtualization: State-of-the-Art and Research Challenges.* IEEE Communications Surveys & Tutorials, 2016, DOI: 10.1109/COMST.2015.2477041, Visited 15.05.2019. Available at: https://ieeexplore.ieee.org/document/7243304

[25] Celesti, A., Mulfari, D., Fazio, M., Villari, M., Puliafito, A. *Exploring Container Virtualization in IoT Clouds.* IEEE International Conference on Smart Computing (SMARTCOMP), 2016, DOI: 10.1109/SMARTCOMP.2016.7501691, ISBN: 978-1-5090-0898-8. Available at: https://ieeexplore.ieee.org/document/7501691

[26] Bannour, F., Suihi, S., Mellouk, A. *Distributed SDN Control: Survey, Taxonomy, and Challenges.* IEEE Communications Surveys & Tutorials 20, 333-354, 2018, DOI: 10.1109/COMST.2017.2782482, Visited 15.05.2019. Available at: https://ieeexplore.ieee.org/document/8187644

[27] Hassan, N., Gillani, S., Ahmed, E., Yaqoob, I., Imran, M. *The Role of Edge Computing in Internet of Things.* IEEE Communications Magazine, 56, 110-115, 2018, DOI:10.1109/MCOM.2018.1700906, ISSN: 1558-1896. Available at: https://ieeexplore.ieee.org/document/8450541

[28] Chiang, M., Zhang, T. *Fog and IoT: An Overview of Research Opportunities.* IEEE Internet of Things Journal, 3, 854-864, 2016, DOI: 10.1109/JIOT.2016.2584538, ISSN: 2327-4662. Available at: https://ieeexplore.ieee.org/document/7498684

[29] Rufino, J., Alam, M., Ferreira, J., Rehman, A., Fung Tsang, K. *Orchestration of containerized microservices for IIoT using Docker.* IEEE International Conference on Industrial Technology (ICIT), 2017, DOI: 10.1109/ICIT.2017.7915594, ISSN: 978-1-5090-5320-9. Available at: https://ieeexplore.ieee.org/document/7915594

[30] Yan, H., Li, Y., Dong, W., Jin, D. *Software-Defined WAN via Open APIs.* IEEE Access, 6, 33752-33765, 2018, DOI: 10.1109/ACCESS.2018.2833211, ISSN: 2169-3536. Available at: https://ieeexplore.ieee.org/document/8354829

[31] Hu, F., Hao, Q., Bao, K. *A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation.* IEEE Communications Surveys & Tutorials, 16, 2181-2206, 2014, DOI: 10.1109/COMST.2014.2326417, ISSN: 1553-877X. Available at: https://ieeexplore.ieee.org/document/6819788

[32] Tran, G.P.C., Chen, Y.A., Kang, D.I., Walters, J.P., Crago , S.P. *Hypervisor performance analysis for real-time workloads.* IEEE High Performance Extreme Computing Conference (HPEC), 2016, DOI: 10.1109/HPEC.2016.7761610, ISSN: 978-1-5090-3525-0. Available at: https://ieeexplore.ieee.org/document/7761610

[33] Garg, Somya., Garg, Satvik. *Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security.* IEEE Conference on Multimedia Information Processing and Retrieval (MIPR), 2019, DOI: 10.1109/MIPR.2019.00094, ISSN: 978-1-7281-1198-8. Available at: https://ieeexplore.ieee.org/document/8695332

[34] Vayghan, L.A., Saied, M.A., Toeroe, M., Khendek, F. *Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned.* IEEE 11th International Conference on Cloud Computing (CLOUD), 2018, DOI: 10.1109/CLOUD.2018.00148, ISSN: 978-1-5386-7235-8. Available at: https://ieeexplore.ieee.org/document/8457916

[35] Wang, Y., Tai, T.Y.C., Wang, R., Gobriel, S., Tseng , J., Tsai, J. *Optimizing Open vSwitch to Support Millions of Flows.* IEEE Global Communications Conference, 2017, DOI: 10.1109/GLOCOM.2017.8254754, ISBN: 978-1-5090-5019-2. Available at: https://ieeexplore.ieee.org/document/8254754

[36] Jutadhamakorn, P., Pillavas, T., Visoottiviseth, V., Takano, R., Haga , J., Kobayashi, D. *A scalable and low-cost MQTT broker clustering system.* 2nd International Conference on Information Technology (INCIT), 2017, DOI: 10.1109/INCIT.2017.8257870, ISBN: 978-1-5386-1431-0. Available at: https://ieeexplore.ieee.org/document/8257870

[37] Yu, Y. *SDN-based Local breakout for mobile edge computing in radio access network.* IEEE Wireless Communications and Networking Conference (WCNC), 2018, DOI: 10.1109/WCNC.2018.8377224, ISSN: 978-1-5386-1734-2. Available at: https://ieeexplore.ieee.org/document/8377224

[38] Lee, S.Q., Kim, J. *Local breakout of mobile access network traffic by mobile edge computing.* International Conference on Information and Communication Technology Convergence (ICTC), 2016, DOI: 10.1109/ICTC.2016.7763283, ISSN: 978-1-5090-1325-8. Available at: https://ieeexplore.ieee.org/document/7763283

[39] Farhad, H., Lee, H., Nakao, A. *Data Plane Programmability in SDN.* IEEE 22nd International Conference on Network Protocols, 2014, DOI: 10.1109/ICNP.2014.93, ISBN: 978-1-4799-6204-4. Available at: https://ieeexplore.ieee.org/document/6980432

[40] Krajsa, O., Fojtova, L. *RTT measurement and its dependence on the real geographical distance.* 34th International Conference on Telecommunications and Signal Processing (TSP), 2011, DOI: 10.1109/TSP.2011.6043737, ISBN:978-1-4577-1411-5. Available at: https://ieeexplore.ieee.org/document/6043737

[41] Hanmer, R., Jagadeesan, L., Mendiratta, V., Zhang, H. *Friend or Foe: Strong Consistency vs. Overload in High-Availability Distributed Systems and SDN.* IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018, DOI: 10.1109/ISSREW.2018.00-30, ISBN: 978-1-5386-9443-5. Available at: https://ieeexplore.ieee.org/document/8539164

[42] Singh, M., Rajan, M.A., Shivraj, V.L., Balamuralidhar, P. *Secure MQTT for Internet of Things (IoT).* Fifth International Conference on Communication Systems and Network Technologies, 2015, DOI: 10.1109/CSNT.2015.16, ISBN: 978-1-4799-1797-6. Available at: https://ieeexplore.ieee.org/document/7280018

[43] Alobaidan, I., Mackay, M., Tso, P. *Build Trust in the Cloud Computing - Isolation in Container Based Virtualisation.* 9th International Conference on Developments in eSystems Engineering (DeSE), 2016, DOI: 10.1109/DeSE.2016.24, ISBN: 978-1-5090-5487-9. Available at: https://ieeexplore.ieee.org/document/7930638

[44] Singh, S., Singh, N. *Containers & Docker: Emerging roles & future of Cloud technology.* 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), 2016, DOI: 10.1109/ICATCCT.2016.7912109, ISBN: 978-1-5090-2399-8. Available at: https://ieeexplore.ieee.org/document/7912109

[45] Ishikawa, N. *Virtual IP Layer: An architecture for virtually extending IP connectivity.* IEEE International Conference on Networking, Architecture and Storage (NAS), 2015, DOI: 10.1109/NAS.2015.7255205, ISBN: 978-1-4673-7891-8. Available at: https://ieeexplore.ieee.org/document/7255205