CONTAINERIZED SERVICES ORCHESTRATION FOR EDGE COMPUTING IN SOFTWARE-DEFINED WIDE AREA NETWORKS

Felipe Rodriguez Yaguache and Kimmo Ahola

5G Networks & Beyond, Technical Research Centre of Finland (VTT), Espoo, Finland

ABSTRACT

As SD-WAN disrupts legacy WAN technologies and becomes the preferred WAN technology adopted by corporations, and Kubernetes becomes the de-facto container orchestration tool, the opportunities for deploying edge-computing containerized applications running over SD-WAN are vast. Service orchestration in SD-WAN has not been provided with enough attention, resulting in the lack of research focused on service discovery in these scenarios. In this article, an in-house service discovery solution that works alongside Kubernetes' master node for allowing improved traffic handling and better user experience when running micro-services is developed. The service discovery solution was conceived following a design science research approach. Our research includes the implementation of a proof-of-concept SD-WAN topology alongside a Kubernetes cluster that allows us to deploy custom services and delimit the necessary characteristics of our in-house solution. Also, the implementation's performance is tested based on the required times for updating the discovery solution according to service updates. Finally, some conclusions and modifications are pointed out based on the results, while also discussing possible enhancements.

KEYWORDS

SD-WAN, Edge computing, Virtualization, Kubernetes, Containers, Services

1. INTRODUCTION

Virtualization is the cornerstone of the 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. The main goal of virtualization 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. Container-based virtualization and Virtual Machines (VMs) are perhaps the most common types of virtualization, although there are many differences among 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 data centers, not only between them. Virtualization alongside cloud-computing supposes a challenge in the application of traffic engineering for maximizing the utilization of the available conventional networks [1].

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-Defined 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 ease, with new features being able to be deployed much faster as well. A centralized control has information regarding the whole network, being able to optimize the available network resources. SDN is therefore widely spread among data centers, especially in order to cope with the virtualization and cloud-computing related issue [1].

Edge computing has arisen as a new approach that alongside SDN could be able to offer a solution to network optimization in cloud environments. This new perspective is nothing more than reducing the number of processes running in the centralized cloud and moving them to locally available edge servers. However, as data processing power is moving towards the edge of a network in the form of containers instead of remaining in a cloud or data center, migration is also occurring for services or applications. This trend requires the usage of processing power from devices that are not capable of being constantly connected to the network, this is the case of laptops, smartphones, wireless sensors, etc. 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 main office, 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 [2].

Edge computing’s adoption arises awareness regarding a substantial change in traffic patterns. The changes in traffic patterns are directly related to the increase in the number of host 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 extra applications that comprise a collection of services provided to customers (guest WiFi, artificial reality, etc.), which are known as "micro-services". The complexity of dealing with multiple hosts raises demand for new management tools, in other words, orchestration. Orchestration automates the 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 the near future through its massive adoption by companies and startups. Its merge with the SD-WAN technology is still to happen, making SD-WAN adopters unable to obtain the most out of their investments [16].

1.1. Methodology

In this work, we propose a simple service discovery system that will improve bandwidth usage when accessing containerized services over a SD-WAN environment. This work was performed in three main steps that include: the selection of use cases and design of the SD-WAN topology, a testbed implementation for the observation of data flow that will allow us to identify the required behavior of our service discovery, and results analysis focusing on a user experience approach. The first step takes into account the limited amount of research aiming at the merge of edge

computing, SD-WAN and container orchestration. 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 and all its constituent parts. The envisioned use cases cover scenarios applicable on an enterprise level and the topology is conceived to simulate the Internet (i.e. a distributed network). The second step comprises the simulation of the aforementioned network topology in order to provide the experiment with a real-life WAN environment. This allows the deployment of in-house containerized services and testing of bandwidth usage and request redirection when performing container orchestration. In the final step, the implementation is compared against commercial solutions and validated based on discovery and convergence times, the whole system will be examined looking for problems and limitations that shall be discussed so improvements can be proposed.

1.2. Related work

A few works have somehow dive into service discovery orchestration in SDN networks. In [10] 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 [11] 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 resource orchestration on a data center environment having and underlying SDN network. In [12], Taleb et al. discuss the role of service orchestration in the success of Multi-access Edge Computing environment, but this mainly focuses on the orchestration of networking resources and containerized services orchestration is not explored. This paper has been adapted from [17], which was delivered for the 8th International Conference on Cloud Computing: Services and Architecture (CLOUD 2019). The present document focuses on the discovery of deployed containerized services, indirectly achieving a slight improvement in the usage of network resources performing orchestration between a container orchestrator and the in-house service discovery.

1.3. Structure

This paper is organized as follows: In section 2 we define our use case and the network topology that will be simulated. In section 3, a summary of the implementation is presented as well as of every constituent part of the system. Finally, in section 4 we perform a brief comparison between the constituent parts of Kubernetes and the developed solution, define the parameters for carrying out measurements, the results and their corresponding discussion are also presented.

2. USE CASE

With SD-WAN being the natural extension of SDN and Kubernetes becoming the de-facto container orchestrator, the possible use cases for an Edge Computing case are high in number. In the proposed use case, a Kubernetes master node will be deployed in a company's Central Office (CO) alongside the suggested 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 including the high-level required 5G network for allowing the implementation of local breakout is depicted.

The advent of internet-connected endpoint devices that are commonly not associated with internet and possess a unique identity is known as Internet of Things (IoT). Edge computing and SD-WAN enable the deployment of workloads that can be of an almost infinite variety, all of them focusing on the processing of IoT generated data. The use cases this work will focus comprises (but is not limited to) the following: smart web pages, authentication applications and the already mentioned IoT data processing. Each of these use cases has 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 performs load 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 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.

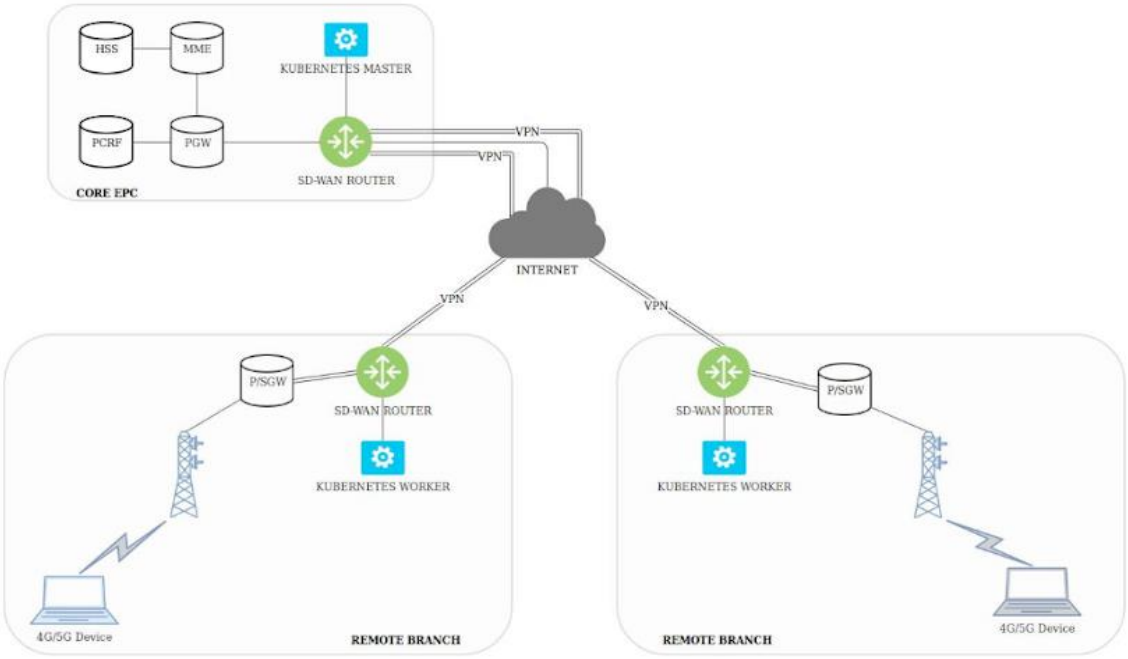

Figure 1. Use case topology

Finally, for IoT data processing use case, data generated by sensors located in a remote branch can be stored in the corresponding Kubernetes worker, while the data processing unit can be deployed in a different worker node. The massive amount of traffic generated by sending and receiving raw data as well as processed data should not be directed through Kubernetes master node in the central office unless it is strictly necessary. The need of dynamic, external service discovery is evident. Although SD-WAN is just starting to be adopted, the integration with Kubernetes will 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 at the moment of writing this document, the proposed paradigm can be further developed as a viable business idea.

3. 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 through a set of Open vSwitch virtual switches that emulate the Internet; the protocol used for the communication between these gateways is BGP, which was implemented by using Quagga [3]. 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 [4], 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. [5]. 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 [4].

The SDN controller runs on the virtual machine and the gateways communicate with it through the main BGP speaker that is located at the main office. 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 by one node [5].

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 solely purpose; the virtual interface located on the SD-WAN virtual machine is loaded to Mininet, therefore simulating a direct connection. 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 conform the master will have one or more containers that are created using Docker, among them, the Container Network Interface (cni). The cni will provide an IP address to every single one of the created pods and is also in charge of the whole networking for the pod network, 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 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 the 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.

Finally, there is 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. Through the use of labels, pods can be scheduled to the worker node, which allows a better resource usage, taking into account that Docker containers are not created in the master node but in the worker node. 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 solely purpose of increasing the isolation between running software, specially conflicts between Kubernetes and Docker. Considering a dockerized version of ONOS is used, any issue affecting the performance of Docker would hinder any effort carried out while building the proof-of-concept testbed.

3.1. Domain name system (DNS)

Kubernetes schedules a DNS pod and service in the master node to 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 both, the client's pod namespace as well as 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 namespace "test-system" must look up for the service with a query for hello-world.kuber-system [6].

This scenario represents the behavior 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 cannot make use of this internal DNS service. For connecting 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 load balancer service type, the service will be exposed either on the internet or limited to a corporate network. Its limitations rely on 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 an unnecessary overhead [7]. 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 nonstandard 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 of posing a security risk for the company when exposing IP addresses of nodes containing vital services [7].

To overcome the aforementioned IP sharing issue, an external DNS service, written in python, was conceived. Every gateway in the SD-WAN virtual machine will run its 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. As an example, hosts located closer to worker1will posses a zone file as shown in Figure 2.

```
# Zones entries for external DNS service
#

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

#
#
```

Figure 2. Custom DNS entries for hosts close to worker1.

By making use of this service, whatever host is connected to the corporate network, close to worker1 will 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 types of traffic. When a host located behind one of the gateways sends a query for certain services, 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 3, 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 sends 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.

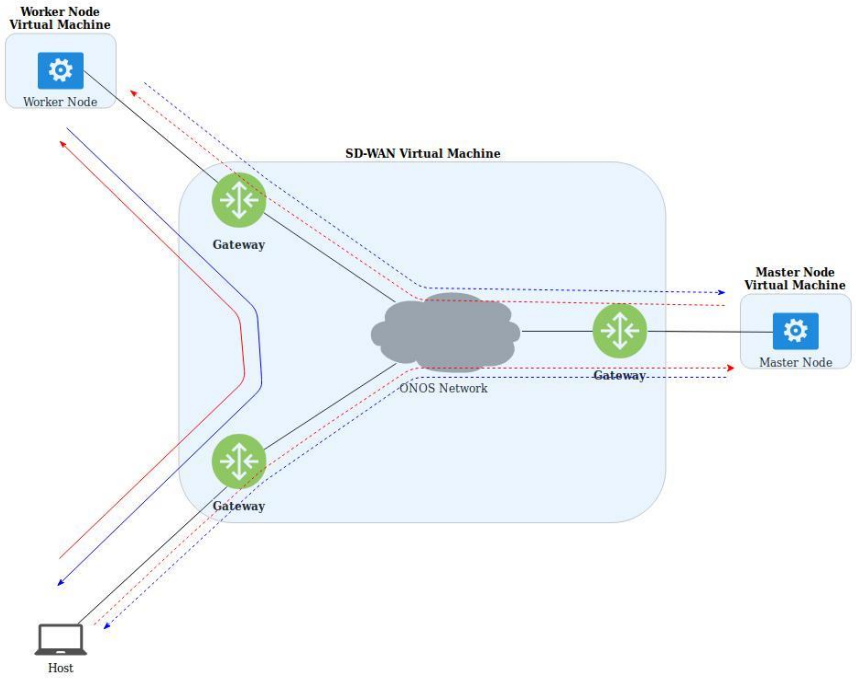

Figure 3. Requests sent by a host.

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 inside a corporate branch and instead use a proxy server. A proxy service is the one attending 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 and has the advantage of enabling load-balancing. Containerized Nginx is the 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 counterparts. The idea behind this implementation is reducing to the minimum the amount of requirements needed for running the worker nodes, 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 to 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 [8] 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’s node 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 HTTP 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 view of worker1, 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 4 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 4. 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 node 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.

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’ containers 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, pods such as calico, Nginx or the 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 routers. 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 the 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 can 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 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 routers, 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. 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 use due to the lack of continuous advertisement. Another reason behind this behavior is the fact that the content of a zone file are mere URLs and their corresponding IP addresses, which are not prone to change, and if they do, this does not happen quite often. In Figure 5, 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 5. Service discovery algorithm.

3.4. Service update system

The updated zones file generated at the master node must be sent to the gateways that are running the external DNS service. For this purpose, a Mosquitto [9] 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 bandwidth consumption. A MQTT publisher was implemented using the paho-mqtt python library, and set up on the CO. This publisher reads the whole 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 an .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 a 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.

The aforementioned constituent parts of the orchestrator are distributed on the underlying network as shown in Figure 6.

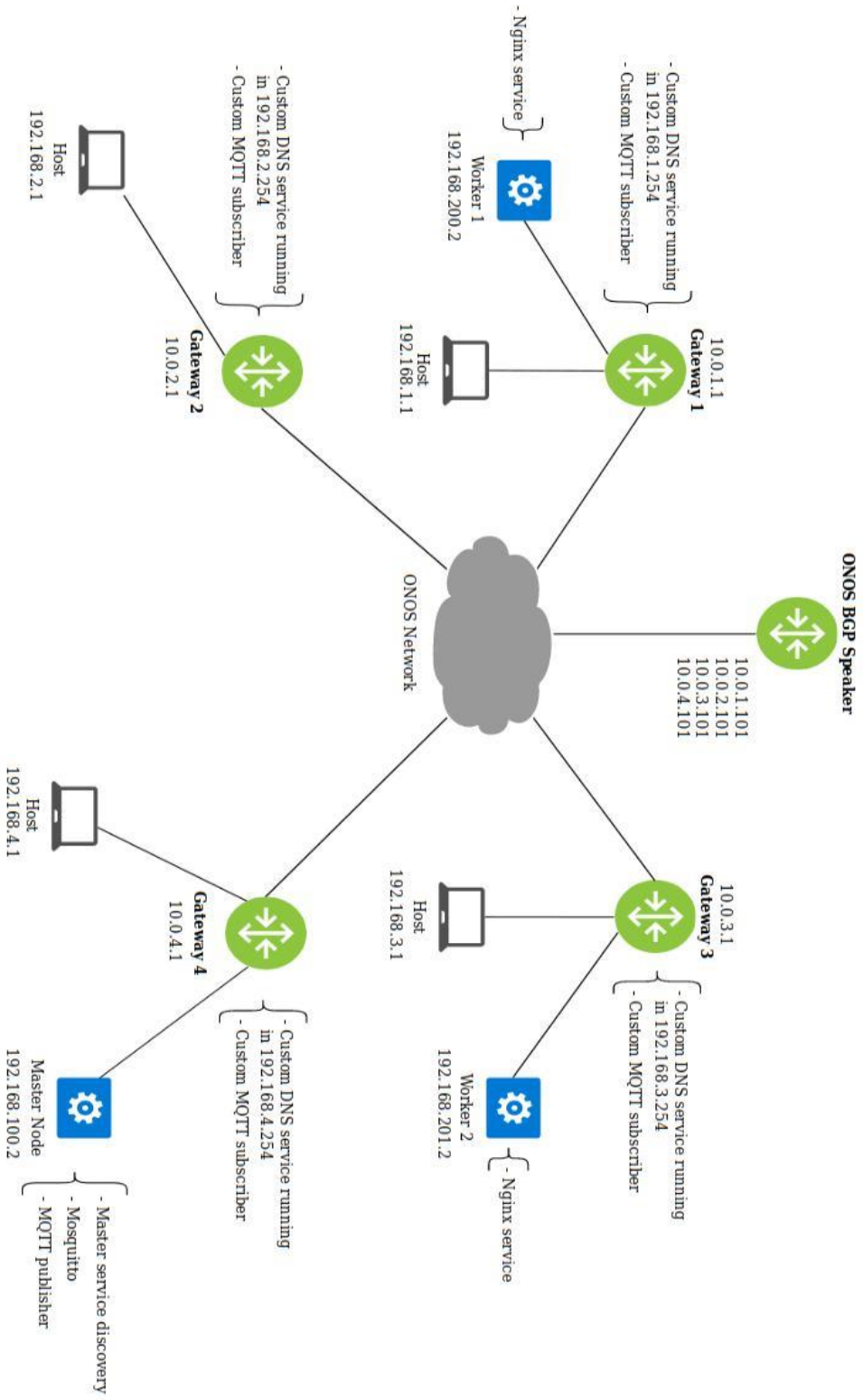

Figure 6. Testing network topology and the services running.

4. COMPARISON AND TESTING

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.

4.1. DNS comparison

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 to offer a lightweight service discovery feature. Enabling DNS based service discovery in a Kubernetes cluster 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 resolve.conf name server 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 also 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 corresponds to a Kubernetes node, neither to 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.

4.2. Proxy comparison

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, 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 no 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. 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.

Kubernetes is a quite complex system, although some networking tasks such as load balancing or specifying some packet rules are directly performed at user space level. The internal proxy will often have to switch between the user space and kernel space 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 services and their respective pods. Next, if a new service is created, kube proxy will open a random port on the node to forward any incoming connection on the port towards the corresponding pod. The pod that will be used is chosen based on the Session Affinity 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.

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, 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 to achieve the desired behavior.

4.3. Service discovery comparison

As explained previously, 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 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.

4.4. Testing

The testing topology as well as all running services and their respective location, can be observed in Figure 6. Under the precedent 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, it hinders 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. This means that 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 resolve.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> they are being loaded as global network configuration. Taking into account that this work is based on the dynamic discovery of updated, newly created and deleted services, the measurements carried out will be 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.

It must be taken into account that reloading the required services in the orchestrator are carried out almost simultaneously; thus, the time for a service to be available for the end-user, as shown in Figure 7, is the time required to reload the DNS and Nginx (~5 seconds). We can infer that at the moment of creating a service, the number of worker nodes available does not influence the overall time. One reason for this might be the fact that Docker containers backing those services are created only in the scheduled worker nodes, therefore eliminating any possible queuing time. Of course, times regarding the download of images required to run certain services are not taken into account. On the other hand, in Figure 8 we can note a slight difference in the files creation function when two worker nodes are present. The reason behind this behavior might be the fact that Kubernetes master node also must delete the corresponding API object for every deleted service.

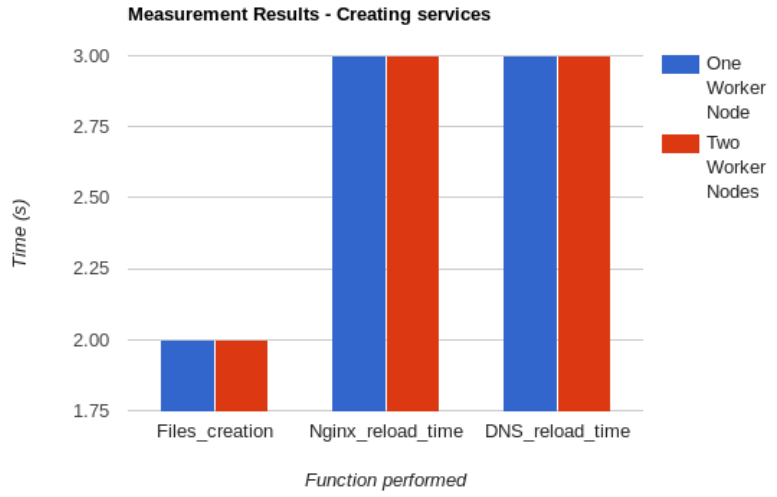

Figure 7. Time required for the orchestrator to perform the necessary tasks in order to fully discover a newly created service.

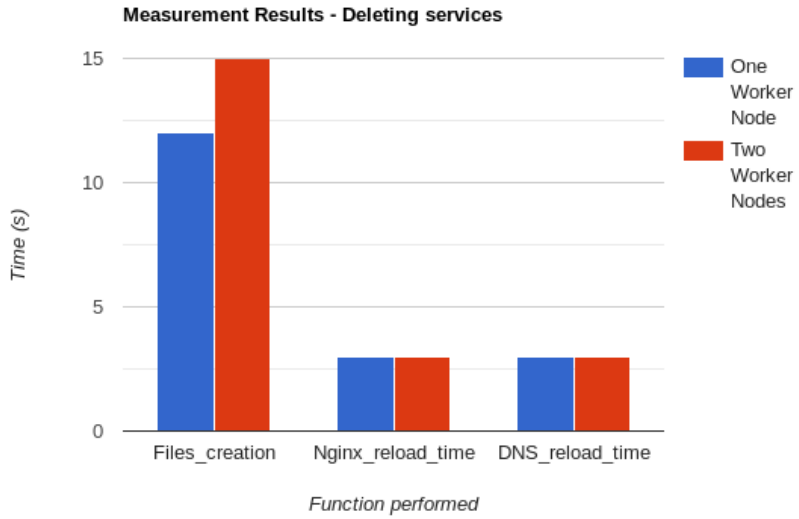

Figure 8. The time required for the orchestrator to perform the necessary tasks in order to eliminate a recently deleted service.

From Figures 7 and 8 it can be inferred that the time of reloading the external Nginx and DNS services does not vary regarding whether a service is being discovered or deleted. This comes from the fact that Nginx and DNS reloading only occurs after the service has been discovered, a task that does not take a considerable amount of time. On the other hand, service deletion takes a much higher time due to pods being granted a grace time 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 required time 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 to ensure a proper deletion. When a service is deleted and its back-end pods stay in the terminating phase for a determined period 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.

Convergence times can be altered to modify the amount of traffic generated by the service discovery as well as its effectiveness when discovering new services. However, the alteration of delays will impact the efficiency of the service discovery and must be analyzed according to the situation. As an example, if services are added on a regular basis one after the other with just a couple of seconds of difference, then a small delay will be required in the master service discovery. 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. Nevertheless, the time spent at reloading Nginx and DNS does not really impact the user experience as it takes no more than 3 seconds.

5. CONCLUSION

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 can 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 identify it.

5.1. Future research

Future work will focus on effectively improving the delivery of the required configuration files, so the right gateways will receive the right information. Taking into account that hosts must be served by the closest available worker node, future work will focus on efficiently measuring the geographical distance between worker nodes as well as a telemetry system for real-time measurement using data plane programmability.

5.1.1 Node distance computing function

A way to dynamically obtain and update the distance between the 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 belief states that latency is an acceptably accurate tool for this purpose, which of course is not a valid assumption. IP addresses cannot 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.

TS Eugene et al have proposed in [13] a method that can be used for the implementation of this feature and 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 allow a distance prediction based on the result of a distance function run over the host's coordinates. This approach works particularly well on a 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 due to the constant changes in network topology, distance estimations will be affected.

5.1.2 Load balancing and telemetry

Data plane programmability can be considered as the natural evolution of SDN, as it enables much flexible networking when compared with a normal control plane based programmable network. Programming Protocol-independent Packet Processors, also known as P4, is the de-facto language for data-plane programmability. It allows several features extension of SDN networks as well as a dynamic configuration of actions that go 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. The proof-of-concept system takes advantage of ONOS's P4 support by implementing some novel features that will improve the experience and manageability of the system and 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.

In [14], Miao Rui et al, demonstrated that it is possible to implement a fully functional 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 higher performance, lower delay and the relief of the MetalLB related pods in all the running worker nodes while decreasing the chances 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 [15] by Changhoon et al. 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 that contain the switch ID and the specific time spent in a determined switch.

REFERENCES

[1] 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 https://www.researchgate.net/publication/264884756_VIRTUALIZATION_TECHNIQUES_TECHNOLOGIES_STATE-OF-THE-ART.

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

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

[4] 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/

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

[6] 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/

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

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

[9] 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

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

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

[12] Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S., & Sabella, D. (2017). On Multi-Access Edge Computing:A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. IEEE Communications Surveys and Tutorials, 19(3), 1657-1681. [7931566]. https://doi.org/10.1109/COMST.2017.2705720

[13] 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

[14] 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

[15] 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

[16] 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

[17] Rodriguez Yaguache F., Ahola K. Enabling Edge Computing Using Container Orchestration and Software Defined Wide Area Networks. 9th International Conference on Computer Science, Engineering and Applications (CCSEA 2019), 353-372. ISBN: 978-1-925953-05-3. Available at: http://aircconline.com/csit/papers/vol9/csit90930.pdf

AUTHORS

Felipe Andres Rodriguez Yaguache is currently finishing his master studies in Communication Engineering at Aalto University (Finland), and is working at the Technical Research Centre of Finland (VTT) as a Master Thesis Worker. His interests include edge computing, SDN, networking and data plane programmability.

Kimmo Ahola currently works as a Senior Scientist at VTT Technical Research Centre of Finland. His research interests include Computer Communications (Networks), Software Defined Networking, Edge Computing, Network Functions Virtualisation, Computer Security and Reliability and Operating Systems.