International Journal of Future Generation Communication and Networking
Vol. 10, No. 12 (2017), pp.35-46
http://dx.doi.org/10.14257/ijfgcn.2017.10.12.04

DockFlex: A Docker-based SDN Distributed Control Plane Architecture

Othmane Blial and Mouad Ben Mamoun

ANISSE, Faculty of Sciences, Mohammed V University in Rabat, Avenue des Nations Unies, Agdal, 10000 Rabat, Morocco
blial.othmane@gmail.com, ben_mamoun@fsr.ac.ma

Abstract

Software Defined Networking is changing entirely the network architecture and moving it forward to make it more flexible, scalable, fast, and highly available. Recently, the idea of containerization is becoming more present in the industry as well as in the academia and has managed to solve many issues linked to virtual machines. SDN has begun with a centralized architecture with a single controller that is responsible for the entire underlying network, thereafter, this solution has shown many overall performance related problems.

After that, the need has raised for distributed SDN networks to obstruct the limitation of performance, security, and scalability. In this paper, we try to take advantage of both notions already addressed, the containerization concept, as well as the control plane distributed SDN networks concept and propose DockFlex, a distributed control plane Docker-based SDN architecture that inherits all the features of a classic SDN architecture and, furthermore, enables a highly manageable and orchestrated multi-controller SDN network that can be established easily via the cloud. Moreover, we suggest a highly customizable implementation of the DockFlex architecture, which uses native Docker-based tools and third-party solutions.

Keywords*: SDN, distributed control plane architectures, Docker*

1. Introduction

Software-defined networking (SDN) [1] is a new concept that aims to make the network more agile, flexible, and highly adaptable to quickly changing business requirements. It is based on the idea of separating between the control plane and the data plane that are joined in traditional networks. It also enables programmability usage to innovate and automate the control plane entirely.

On the other hand, Network function virtualization (NFV) [2] is an emerging technology which has a goal of replacing current physical components of the network like switches, routers, and firewalls, with software that runs on servers that are usually on the cloud. This means more elasticity, less time-consuming for service activation, creation and monitoring and more advantages brought by the cloud.

SDN and NFV highly depend on each other, SDN adds to the network the smartest device, which is the controller, which needs to be physically distributed to respond to scalability and high availability requirements of today's networks. While NFV adds what we can call virtual switches, or virtual firewalls, which can be purely software installed on many servers for high availability, on the cloud.

Almost all SDN solutions that support multiple controllers, like ONOS [3], OpenDayLight [4], and Contrail [5], usually install their controllers on Linux-based

Received (August 15, 2017), Review Result (October 31, 2017), Accepted (November 13, 2017)

ISSN: 2233-7857 IJFGCN

virtual machines, using local servers or on the cloud. Virtual machines until now are performing very well and delivering good quality. However, the cloud recently knows a big revolution by the introduction of emerging new cloud products like Docker that use the notion of containerization, which has managed to solve many problems related to VMs.

Before digging deeper in the world of containerization, let's first understand why we need to abandon virtual machines (VMs), while they are currently the most used solution in production. VMs have succeeded to provide virtualized operating systems (OSs), however, each time we create a new VM, we need to go through installing an OS, making it ready to work with by installing and configuring a bunch of drivers and licenses then deploying an application that we want to work with on the top of it. Bottom line, we go through many steps to finally deploy an application, which is our main goal from the beginning, these repeated steps represent the main problem with VMs, in addition to the overhead brought by the VM's OS, by consuming the CPU, the Disk space and the Memory of the physical machine, where the hypervisor resides. Moreover, VMs needs usually minutes to be started, migrated or destroyed.

On the bright side, Containers are very lightweight and very fast in comparison to VMs, because they remove the overhead brought by the OS, which means they use less memory, CPU and disk space. Containers try to reach the goal directly by deploying the application wanted without installing or configuring the OS, simply by bringing the minimal essential dependencies to work with that targeted application. Another reason why containers are faster than VMs is the fact that they share the same kernel of the physical machine.

It might occur an interesting question, which is why we need containers, why ce cannot just install as many applications as we need on the top of the machine we have. The answer is simply we won't have for each application a separated networking stack, a shared library stack, or any of the others stacks, which define a unique Linux environment.

In order to provide each container with its own private stacks, we use kernel namespaces [6], which allow each container to have an isolated private networking stack as well as a process stack, a user space stack, *etc*,. Another great feature that permits creating containers is the cgroups [6] feature, which permits the physical machine to fully control resources that are shared among the different containers.

Now that we have an idea about containerization, let's introduce Docker [7], a cloud based platform, which is a containerization implementation. Docker is an open source project used to pack, ship and run any Linux application or distribution as a highly manageable and lightweight container, built using a Docker image.

A Docker container can run any service or program by loading a filesystem that has everything needed to run. In other words, it is an isolated process working on an isolated filesystem. This container is very lightweight since it uses less RAM and CPU because it shares the same system kernel of the machine. Additionally, it is not tied to any type of infrastructure. A Docker container also has the power to instantiate, destroy instances immediately, and remove environment inconsistencies.

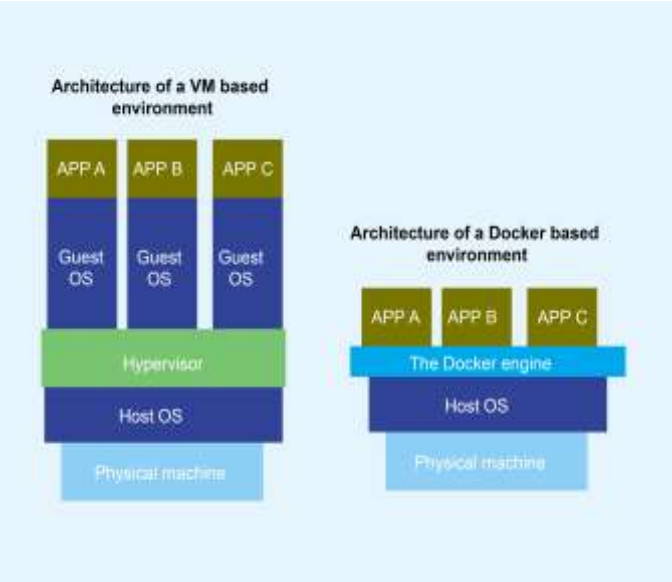

Figure 1. Comparing a VM Based Environment to A Docker Based Environment

In the SDN context, we can instantiate an SDN controller as a container. To build a container, we need to use a Dockerfile or a starting point image, an Ubuntu server Docker image from the Docker hub for example, and install on it the SDN controller related files. In the following of this paper, we are going to note a Controller-Container as a CC.

Using Docker to instantiate a new CC can bring all the benefits of containerization, like the speed offered to provision, destroy or migrate a CC, as well as less overhead compared to VMs, smooth backing up, easy maintenance and optimized automation process.

Few research papers have covered Docker in the SDN context. For example, a paper [8], highlights the advantages of Docker containers in SDN networks. It explains why VMs based SDN networks have a lack of speed regarding deployment, migration, integration, and scalability as well as the complexity of network management, then it shows the power of Docker over VMs. Moreover, this paper shows how relying on Docker to build a virtualized SDN network led to rapid deployability and good extensibility.

Another paper [9] has handled Docker in an SDN context proposed a Docker-based framework that allows to automatically install and uninstall an application at the edge switches according to the user needs. Likewise, this framework can efficiently manage the storage, the computing, and the networking resources of the switch.

Using Docker within an SDN network can be very beneficial, however, the issue appears when we instantiate many CCs, there is a lack of a management and orchestration architecture that allows us to have an up and running distributed control plane with multiple CCs on the top of a Docker-based cloud environment.

To remediate this problem, we propose, DockFlex, a distributed control plane architecture for a Docker-based SDN environment. DockFlex enables smart management and interactive orchestration, which can also provide high scalability and high performance to ensure an emerging distributed control plane.

The remainder of this paper is organized as follows. In section 2, we give a background about Docker and Docker Swarm. In section 3, we describe the DockFlex architecture, then we provide the implementation of DockFlex that we

propose to build the DockFlex distributed control plane. In Section 4, we give some simulation results. Finally, in Section 5, we give perspectives and conclusions.

2. Docker and Docker Swarm: Background

In this section, we give a general background about Docker, as well as Docker Swarm, the native Docker clustering solution.

2.1. Docker Background

Docker [7] is a fast-emerging cloud-based technology solution and platform, which can be considered as the most recognized and used containerization implementation currently. Docker resides on three essential components, the Docker engine, which is the agent that must be installed on the machine we are working on, to take advantage of Docker. The Docker engine also called the Docker runtime. It provides us with a Docker-based environment that permits accessing all the Docker services. By developing and deploying an application on the Docker runtime grants having a portable application that can run smoothly, on a laptop, on a data-center, a cloud-based server, *etc*. The second component is the Docker images, which are used to lunch a Docker container. They are built usually using the Dockerfile which is a list of commands and instructions to pull all the minimal dependencies and make the essential configurations to build a certain application. The third component is the Docker container, which is the most important component of the Docker environment. It is launched based on an image that we have locally, or that we can pull down from the Docker official web store.

2.2. Docker Swarm Background

Docker can be integrated with many native solutions and third-party solutions. Docker swarm [10] is one of the most useful Docker-based solutions to make clustering very smooth. It is considered as a clustering solution used to control different containers in a Docker-enabled environment. Docker swarm needs two major components to work. First, the discovery service, which is a key-value store that is responsible for maintaining the state of the cluster and its configuration. Second, the swarm manager, which controls the entire cluster by receiving Docker commands and sending commands to a particular node in the cluster. Both of these services should be duplicated for high availability purposes.

The discovery services that Docker Swarm supports are :

- Zookeeper [11], which is a centralized service used to maintain configuration, provide information and synchronize data.

- The etcd tool [12], which is a shared configuration solution for distributed environments using key-value stores.

- Consul [13], which is a service discovery and configuration solution also based on value-key stores, which is more adaptable and easy to configure. It supports failure detection and datacenter-awareness.

Instantiating new containers within a Docker swarm based network can be managed manually or automatically. In order to instantiate a new container manually, Docker swarm offers the possibility to choose the node we want based on its different attributes, like the location, the Linux distribution, the available resources or even using custom attribute added by the administrator. While in order to instantiate new containers automatically, we can choose among three options, the Random, the Spread and the Binpack option.

- The Random option is the less intelligent option. It distributes containers randomly on the nodes, it doesn't care about any attribute, it may seem have less overhead, we don't need to compare between the nodes current resources, but it is not efficient in a working environment.

- The Spread option, which is the most reasonable option, which inspects the node that has the least number of containers to add a new one, this option helps to balance between nodes, and it is very useful in a productive environment.

- The Binpack option, which is based on releasing the most of the infrastructure we can, which means, we add new containers to the same node until it is full, then we can go to the next node. This method is also very useful and efficient for real-world environments.

3. The DockFlex Proposition: Architecture and Implementation

In this section, we describe the architecture of the DockFlex proposition, then we suggest an implementation for this architecture.

3.1. The Dockflex Architecture

The DockFlex architecture that we propose aims to take advantage of a containerization based environment, to enhance a distributed control plane SDN network overall performance.

It is composed of three layers since it inherits the characteristics of the classic SDN architecture, an infrastructure layer, a distributed control layer and a management layer.

In the first place, we have the infrastructure layer, which has the different network devices that are connected to the control layer via an SDN southbound API.

While the distributed control layer or plane is where we have a cluster of nodes, and on each node, we have one or more SDN controllers as containers, each one is connected to the infrastructure layer using an SDN southbound API and to the management layer using an SDN northbound API.

Finally, the management layer, which has many components that we propose, which characterize the DockFlex architecture, in addition to all applications that can reside within this layer:

The discovery service node: this component maintains the state of the cluster and its configuration, assuring its smooth functionality.

The cluster manager node: this component that controls and manage the entire cluster, by sending or receiving the command to any particular node in the cluster.

The cluster logs collector store node: this component gathers different types of logs from all the nodes in the cluster using logs forwarders engines installed within each node, then saves them inside. These logs can be used to retrieve all types of information about each CC within each node.

The cluster monitor node: this component is responsible for real-time monitoring of resources consumption as well as overall performance.

The following illustration shows the DockFlex architecture.

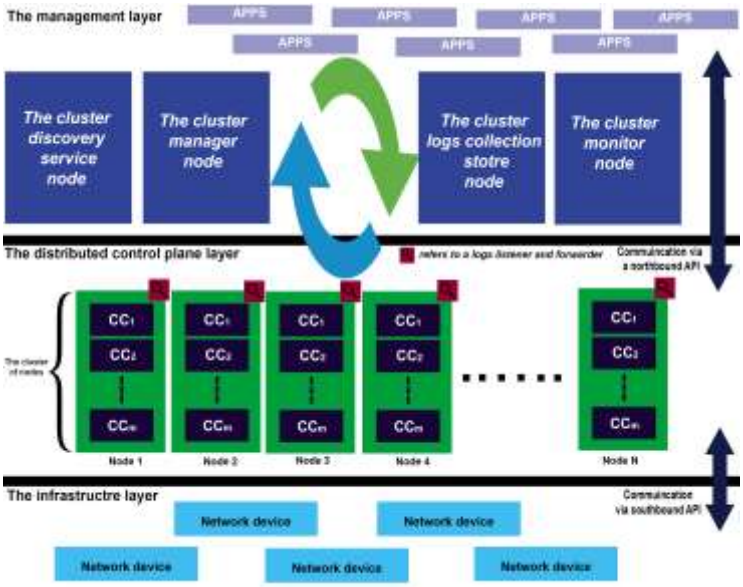

Figure 2. The Architecture of DockFlex

The DockFlex architecture can be easily extended, by adding new components or also, removing existing components, as well as changing the role of a component within the architecture.

3.2. The Dockflex Implementation

The DockFlex architecture that we have presented previously, can be implemented using any containerization solution and using all types of solutions, concerning clustering discovery, clustering management, logging collection, logging storing and monitoring.

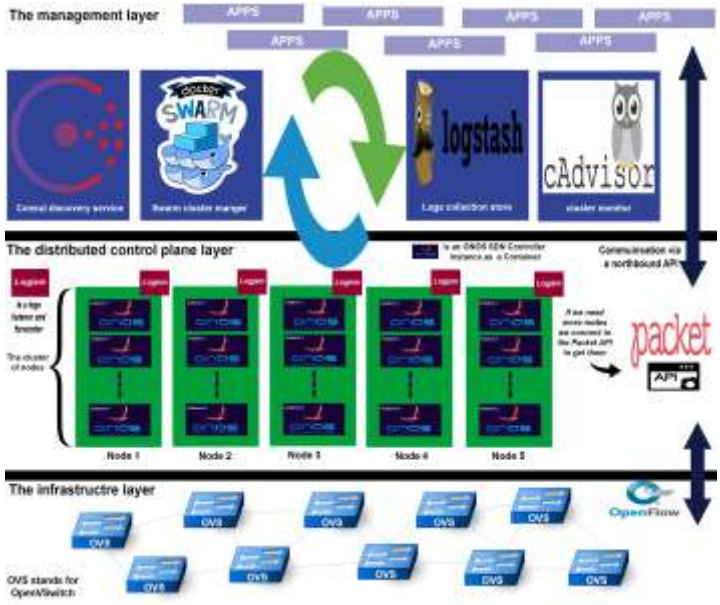

Figure 3. The Proposed Implementation of DockFlex

Our DockFlex implementation, like we see the above figure, uses Docker, as containerization solution, because it is the most used containerization solution, not

to mention, it has a very large community supporting it. Our implementation tries to orchestrate and manage a cluster of CCs, for this reason, we choose the Docker Swarm, a native clustering solution to Docker.

In order to make an up and running cluster of CCs, we need a swarm manager, which is a node that is responsible for executing different commands inside each node within the cluster. This manager can be used to provision a new CC, migrate, shutdown or destroy an existing CC, as well as many other things.

Additionally, we need a discovery service, which responsible for maintaining the state of the cluster and its configuration. We choose Consul as a discovery service, which is used in production environments by many companies.

Each CC inside each node in the cluster, will generate events and different type of outputs, that can all be regarded as logs, these logs need to be stored in a centralized store, likewise, any other CC, can reach them, and take advantage of them. For this reason, we use Logjam [18], is a log forwarder, which is designed to listen on the docker0 interface within each node in the cluster, and receive over UDP, all the log entries, received from the different CCs inside that particular node, and then send them to a centralized log collection store node.

Concerning this last mentioned, we use Logstash [19], which is an open source tool that collects, parse, and store logs for future use.

To enable active monitoring within our DockFlex implementation, we use cAdvisor [20], which will analyze in real-time, the different characteristics and resources consumptions for all the CCs, within each node in the control plane.

Our implementation suggests the use of high availability, which means each node in the orchestration layer should be duplicated. Additionally, we can use all the different components we talked about as containers.

Each CC inside each node is an SDN ONOS controller instance as a container. ONOS is The Open Network Operating System, an OS designed for SDN networks, which has high scalability, high performance and designed for Data Center based networks, as well as service provider networks.

We suggest for this implementation the use of OVS (OpenVSwitch), in the infrastructure layer, because they have the particularity of being full software switches, and they support OpenFlow, which is the Southbound API (SB-API) used for communication between the OVS witches and a particular CC. OpenFlow is the most used and recognized SB-API, which happens to be always linked to SDN.

The DockFlex implementation is directly connected to the Packet.Net, which is a cloud service that permits to provision full Bare Metal servers on the cloud. In case we are running out of node inside the cluster, we can easily provision a new node (server), and install the different dependencies we need while provisioning, thanks to the optional user data, offered by this cloud service.

This proposed implementation can be extended to use all sort of open source tools, which can be integrated to enhance the overall functionality of DockFlex.

4. A Simulation of the DockFlex Core

As we have seen, the implementation of DockFlex, requires the combination of many Docker native tools as well as third-party solutions, and since this is a work in progress, we haven't been able yet to simulate the DockFlex implementation, for this reason, we will present a simulation of the core of DockFlex, which is Docker, and at the same time, show the superiority of a Docker-based SDN network in comparison with to a VM-based SDN network.

4.1. Testbed Description

In this testbed, we are going to use a physical machine from the packet.net [14] Bare Metal service, of type 0 (1 x Intel® Atom™ C2550, 4 cores, 2.4 GHz and 8GB of DDR3 RAM).

Within this machine, we are going to install VirtualBox [15], which will allow us to deploy VMs as well as Docker, which will permit us to create containers.

We are going to use ONOS [3], an emerging and rapidly growing SDN controller, by building it as a container, as well as a VM. Then we are going to connect a first ONOS controller to a simulated Mininet [16] network, which is a fat tree topology, which has one OVS [17] switch, connected to three other OVS switches, where each one of those three is connected to three simulated Mininet hosts. We also connect the same topology to a second ONOS controller.

The following figure summarizes the previously explained testbed:

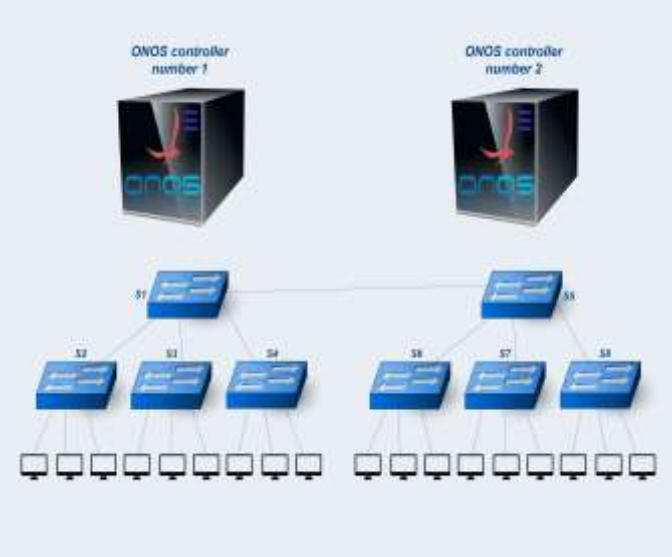

Figure 4. The Testbed Description

4.2. Performances Indices

To measure the performance of this use-case implementation, we measure two different performance indices. First, the CPU consumption, which is an important index to see how much the controller and its underlying networks are consuming the processor resources. Second, we measure the memory consumption, which also relevant for the overall resources available. These two indexes are very significant for the cloud environment to show us how much our implementation handle the available resources.

5. Simulation Description, Results and Analysis

Within this experiment, we measure the CPU and memory consumption, by building up the underlying network, which will construct the switches, the hosts as well as the links between switches and hosts. After that, we assure the command PingaAll, within the Mininet simulated network, so the simulated hosts can discover each other.

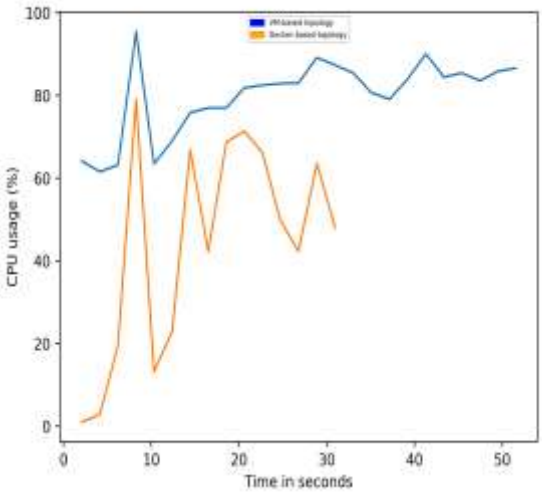

Figure 5. CPU Usage for a VM based versus a Docker Based

Figure 5 shows that the CPU consumption goes up to about 79%, for a Docker-based SDN network, and up to about 95% for a VM based SDN network, within the switch building phase. In this phase, many switch events are exchanged at the same time, it consumes a lot of CPU resources.

While, within the phase when the Mininet hosts ping each other, the CPU decreases for the VM-based, but still higher than the Docker-based SDN network.

The difference of CPU usage, it simply due to the fact that when using a VM, we are bringing the overhead brought by the guest OS, which is Ubuntu in this use case. In contrast, when using Docker, we are using the ONOS instance as a container directly, which means lower overhead, likewise, lower CPU usage.

Another thing to point out is the fact that when we issue the command PingaAll on Mininet, it takes about 20 more seconds to finish pinging all the Mininet hosts. Which means, that the Docker-based environment is faster.

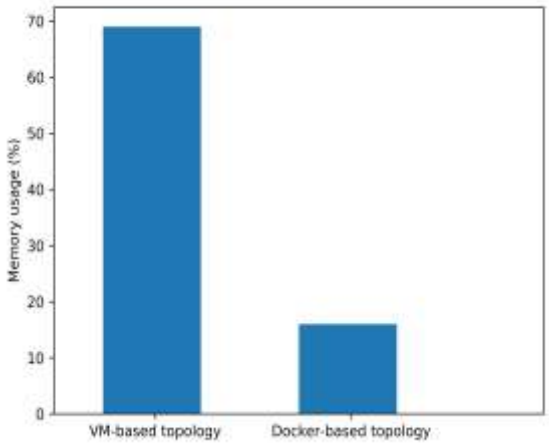

Figure 6. Memory Usage for a VM based Versus a Docker Based

Figure 6 shows the memory usage for both, a VM-based and Docker-based topologies. For the VM-based environment, we consume a significantly high

memory, up to 69% of the global memory. By comparison, the Docker-based environment is consuming much less memory, which is about 16%.

In other words, we can say that the Docker environment has managed to save up to 53% of the available memory.

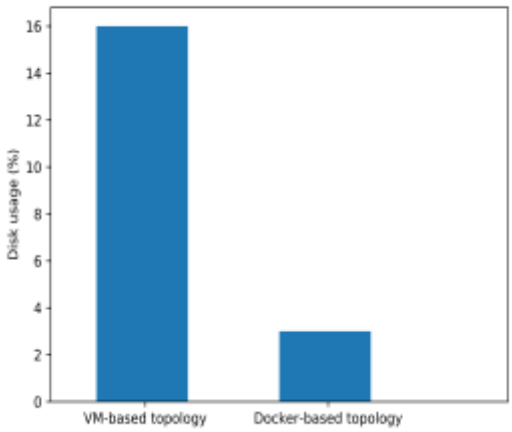

Figure 7. Disk Usage for a VM based Versus A Docker Based

Figure 7 shows that the disk space consumed was 16% when using a VM-based topology, then it decreases down to just 3%, using the Docker-based environment.

All the resources measurements that we have observed during the lunching of the Mininet based topology, have demonstrated the superiority of the Docker-based environment in comparison to the VM-based environment.

5. Conclusion and Perspectives

In this paper, we take advantage of the concept that has become very popular in the recent years, which is containerization, which came to solve many problems related to virtual machines, in the context of SDN networks. More precisely, we use containerization for distributed SDN networks, since distribution has become a must to remove many limitations in terms of performance. Based on these two concepts, which are SDN distributed networks and containerization, we propose DockFlex, which is an orchestrated container-based SDN distributed architecture, which resides in the cloud.

We have presented and described this architecture, then we have suggested an implementation based on this architecture, which uses Docker as a containerization environment while combining many native Docker tools and other third-party solutions, to enable many important operations within modern networks, like communication, logging, and monitoring.

Concerning the next steps of this research paper, we plan to build the DockFlex implementation and deploy it within a real testbed on the cloud to show its performance in production environments. We suggest the use of multiple WAN network topologies for the underlying networks, to be able to watch the behavior of the DockFlex based network, within different topologies, each one with a different number of underlying nodes.

Furthermore, we look forward to discussing and analyzing the scalability opportunities within a DockFlex implementation, as well as cohabitation between DockFlex with other types of containerization environments.

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-defined networking: a comprehensive survey", Proceedings of the IEEE, vol. 103, no. 1, **(2015)**, pp. 14–76.
[2] J. Gil Herrera and J. Botero Vega, "Network Functions Virtualization: A Survey", IEEE Latin America Transactions, vol. 14, no. 2, **(2016)**, pp.983-997.
[3] ONOS, "Open Network Operating System", [online] Available at: http://onosproject.org/, **(2017)**.
[4] "The OpenDaylight Platform | OpenDaylight", Opendaylight.org, 2016. [Online]. Available: https://www.opendaylight.org/.
[5] Juniper.net, "Contrail SD-WAN and Open SDN NFV Solutions - Juniper Networks", [online] Available at: http://www.juniper.net/us/en/products-services/sdn/contrail/, **(2017)**.
[6] R. Rosen, [online] Available at: http://Resource management: Linux kernel Namespaces and cgroups – Haifux, **(2017)**.
[7] Docker, Docker. [online] Available at: https://www.docker.com/, **(2017)**.
[8] L. Xingtao, G. Yantao, W. Wei, Z. Sanyou and L. Jiliang, "Network virtualization by using software-defined networking controller based Docker", 2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference, Chongqing, **(2016)**, pp. 1112-1115.
[9] Y. Xu, V. Mahendran and S. Radhakrishnan, "SDN docker: Enabling application auto-docking/undocking in edge switch", 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), San Francisco, CA, **(2016)**, pp. 864-869.
[10] GitHub, docker/swarm. [online] Available at: https://github.com/docker/swarm, **(2017)**.
[11] Zookeeper.apache.org. (2017). Apache ZooKeeper - Home. [online] Available at: https://zookeeper.apache.org/.
[12] GitHub. coreos/etcd. [online] Available at: https://github.com/coreos/etcd, **(2017)**.
[13] Consul by HashiCorp. Consul by HashiCorp. [online] Available at: https://www.consul.io/, **(2017)**.
[14] Premium Bare Metal Servers and Container Hosting – Packet, Premium Bare Metal Servers and Container Hosting - Packet. [online] Available at: http://packet.net, **(2017)**.
[15] Virtualbox.org. Oracle VM VirtualBox. [online] Available at: https://www.virtualbox.org, **(2017)**.
[16] M. Team, Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet. [online] Mininet.org. Available at: http://mininet.org/, **(2017)**.
[17] Openvswitch.org. Open vSwitch. [online] Available at: http://openvswitch.org/, **(2017)**.
[18] GitHub. gocardless/logjam. [online] Available at: https://github.com/gocardless/logjam, **(2017)**.
[19] Elastic.co., Logstash: Collect, Parse, Transform Logs | Elastic. [online] Available at: https://www.elastic.co/products/logstash, **(2017)**.
[20] GitHub. google/cadvisor. [online] Available at: https://github.com/google/cadvisor, **(2017)**.

