Towards Osmotic Computing: Analyzing Overlay Network Solutions to Optimize the Deployment of Container-Based Microservices in Fog, Edge and IoT Environments

Alina Buzachis*, Antonino Galletta*, *Student Member, IEEE*, Lorenzo Carnevale*, *Student Member, IEEE*
Antonio Celesti*†, *Member, IEEE*, Maria Fazio*† and Massimo Villari*†, *Member, IEEE*

* MIFT Department, University of Messina, Contrada di Dio 1, 98166, Messina, Italy
{abuzachis, angalletta, lcarnevale, acelesti, mfazio, mvillari}@unime.it

† Alma Digit S.R.L., Contrada di Dio 1, 98166, Messina, Italy
{a.celesti, m.fazio, m.villari}@almadigit.com

***Abstract*—In recent years, the rapid growth of new Cloud technologies acted as an enabling factor for the adoption of microservices based architecture that leverages container virtualization in order to build modular and robust systems. As the number of containers running on hosts increases, it becomes essential to have tools to manage them in a simple, straightforward manner and with a high level of abstraction. Osmotic Computing is an emerging research field that studies the migration, deployment and optimization of microservices from the Cloud to Fog, Edge, and Internet of Things (IoT) environments. However, in order to achieve Osmotic Computing environments, connectivity issues have to be addressed. This paper investigates these connectivity issues leveraging different network overlays. In particular, we analyze the performance of four network overlays that are OVN, Calico, Weave, and Flannel. Our results give a concrete overview in terms of overhead and performances for each proposed overlay solution, helping us to understand which the best overlay solution is. Specifically, we deployed CoAP and FTP microservices which helped us to carry out these benchmarks and collect the results in terms of transfer times.**

***Index Terms*—Osmotic Computing, Cloud Computing, Fog computing, Edge Computing, IoT, virtualization, container, network overlay, orchestration, microservice.**

I. Introduction

Nowadays, with the advent of emerging Cloud/Edge/Fog computing and modern Internet of Things (IoT) technologies, we are observing a revolution in different domains (e.g economy, politics, society, and so on). In this context, cutting-edge Information and Communication Technology (ICT) solutions are changing both production and consumption rates, automating various real-life processes and leading them to mainly deal with control functions.

According to a recent survey of Gartner[1], the total number of connected objects in 2020 will be 25 billion. This proliferation of objects and associated data streams has put significant stress on the network and edge layer leading to the emergence of new problems in terms of data confidentiality and network performance respectively. In order to tackle those problems, in recent years, the ongoing evolution of Cloud Computing and the adoption of the microservices-based architecture led a drastic change in application building, deployment and delivery.

Microservices-based architecture is tailored to the needs of todays software which requires high degree of flexibility and dynamism. Therefore, it combines many sources of information, devices, applications, services and microservices into a flexible architecture where applications extend over multiple endpoint devices and coordinate with each other to produce a continuous digital experience.

Moreover, managing an IT infrastructure is not an easy task, especially when it is necessary to configure and manage a large number of machines. For this reason, recently automated devices are gaining popularity in the field of Internet of Things and embedded systems. Today's complex ecosystem includes an Edge/Fog Computing scenario with numerous devices (on which microservices are spread) geographically dispersed (and therefore belonging to different domains). Hence, the problem that most affects the various components of this ecosystem concerns the implementation of a networking infrastructure able to interconnect them. Since 2016 [2], a new promising paradigm for the integration between a centralized Cloud layer and Edge/IoT layers has been proposed. The Osmotic Computing, borrowing the term from chemistry, aims to overcome the microservices elastic management, because of deployment and migration strategies depend on the requirements of infrastructures, such as load balancing, reliability, availability, and applications, such as anomalies detection, awareness of the context, proximity, QoS. Furthermore, in order to overcome the heterogeneity of IoT resources, the microservices abstraction allows us to support a virtual environment that can be adapted according to the available hardware equipment.

In this complex scenario, this scientific work tackles the problem of connectivity in a distributed microservices-based scenario, and proposes a system able to deal with it. In particular, we implemented our system using a Kubernetes cluster organized in three tiers: Cloud, Edge and Fog. In order to connect the cluster components, Kubernetes relies on third-party plug-ins. In fact, we tackled this problem by proposing several network solutions. Thus, on each cluster

978-1-5386-6488-9/18/$31.00 ©2018 IEEE

component belonging to a specific tier we attached several network overlays (OVN, Calico, Weave and Flannel).

We deployed two different distributed microservices based respectively on FTP and CoAP. Then, in order to quantify the network performances of our implemented system, we realized a benchmark based on the transfer times registered using each implemented microservice. Furthermore, in order to quantify the overhead introduced by each network plug-in we also configured the system's networking without any overlay (mapping directly container to host ports).

Thus, analyzing the deployed system's performances, we noticed that OVN based solution is better than others. Indeed, it presents lower execution time and offers more functionalities for managing microservices geographically distributed. Furthermore, independently of their performance analysis, the choice of the best network overlay is not an easy task because it mostly depends not only on the user's needs but also on the scenario specifications.

The remainder of this paper is organized as follows. We present the related works in Section II. An overview of the deployed system is presented in Section III. We detail the system's implementation in Section IV. Performances analysis are presented in Section V and, finally, conclusions and light to the future are discussed in Section VI.

II. Related Work

In recent years, several scientific works have been proposed regarding service orchestration in Cloud computing. A method for automated provisioning of Cloud business applications able to manage the whole application's lifecycle is proposed in [3]. Such a method is based on Cloud orchestration tools able to manage the deployment of the required software components and dependencies. In particular, a Linked Unified Service Description Language (LUSD) aimed at describing services matching user's requirements is proposed. Designing an edge Cloud network implies first determining where to install facilities among the available sites, then assigning sets of access points, while supporting VM orchestration and considering partial user mobility information, as well as the satisfaction of service-level agreements. In this context, a link-path formulations supported by heuristics to compute solutions in reasonable time is discussed in [4]. In particular, the advantage in considering mobility for both users and VMs is discussed. A Cloud4IoT platform offering automatic deployment, orchestration and dynamic configuration of IoT support software components and data-intensive applications for data processing and analytics is proposed in [5]. It enables plug-and-play integration of new sensor objects and dynamic workload scalability. In addition, the concept of Infrastructure as Code in the IoT context that empowers IoT operations with the flexibility and elasticity of Cloud services is proposed. Furthermore it shifts traditionally centralized Cloud architectures towards a more distributed and decentralized computation paradigm, as required by IoT technologies, bridging the gap between Cloud Computing and IoT ecosystems. An approach that facilitates the evaluation of different application topologies by enabling Cloud users to easily provision and evaluate different Topology and Orchestration Specification for Cloud Applications (TOSCA) options based on performance and cost metrics is proposed in [6]. In particular, the technical feasibility of the approach is evaluated. A set of ontologies that semantically represent Cloud resources is proposed in [7]. Specifically, three Cloud resource description standards are considered, that are: TOSCA, Open Cloud Computing Interface (OCCI), and Cloud Infrastructure Management Interface (CIMI). The proposed framework promotes the creation of a common semantic knowledge base of Cloud resources described using these different standards. A multitenancy ad-hoc service orchestrator for federated OpenStack Clouds is proposed in [8]. In particular, an orchestration broker based on OpenStack-based Heat Orchestration Template (HOT) service manifest is able to produce several HOT microservice manifests including deployment instructions for involved federated Clouds. Accordingly, users can formalize advanced deployment constrains in terms of data export control, networking, and disaster recovery. Experiments prove the goodness of the proposed system in terms of performance. A deployment orchestration of microservices with geographical constraints for Edge computing is proposed in [9]. Assuming that a distributed service is composed of several microservices, users, by means of geolocation deployment constrains can select regions in which microservices will be deployed. Specifically, an Orchestration Broker that based on a HOT service manifest of an Edge computing distributed service produces several HOT microservice manifests including the the deployment instruction for each involved Fog computing node. A recommendation-based approach for Cloud service brokerage in public administration scenario that helps European public administration Clouds to choose Service Flow Lots (SFLs) in order to create specific administration service workflows is proposed in [10].

Regarding, the orchestration of networking services, a control orchestration protocol is proposed in [11]. It provides a transport application programming interface solution for joint cloud/network orchestration, allowing interworking of heterogeneous control planes to provide provisioning and recovery of quality of service (QoS)-Aware end-To-end services. In the domain of future 5G network applications, a Software Defined Networking / Network Function Virtualization (SDN/NFV) orchestrator that automatically serves Mobile Network Operators (MNOs) capacity requests by computing and allocating virtual backhaul tenants is discussed in [12]. Such backhaul tenants are built over a common physical aggregation network, formed by heterogeneous technologies (e.g., packet and optical) that may be owned by different infrastructure providers.

In the context of orchestration of energy-aware services, a novel energy-aware resource orchestration framework for distributed Cloud infrastructures is presented in [13]. In order to manage both network and IT resources in a typical optical backbone. A high-level overview of the system architecture is provided by focusing on the definition of the different layers of the whole infrastructure, and introducing the Path

Computation Element, which is the key component of the proposed architecture.

In [14] and [15] authors proposed a hybrid approach for modeling and simulating IoT systems in terms of multiagent systems (MASs) and network simulator in order to investigate communication issues related to SO deployment in IoT systems.

In [16] authors aim at developing a platform to facilitate dynamic resource-provisioning based on Kubernetes. The platform has three features:

1) comprehensive monitoring mechanism;
2) dynamic resource-provisioning operation
3) control loop which can be applied to all the running applications.

Differently from the scientific works focusing on service Orchestration in Cloud/Edge/Fog computing and Internet of Things (IoT), in this paper we focus on microservice orchestration considering the emerging Osmotic computing paradigm.

III. Overview and basic assumptions

In this section, we give an overview on our proposed system, including container technology, applications deployment and networking solutions. In this paper, we envision an ecosystem composed of a centralized VM - master, which represents the ”brain” of the system, and several worker nodes, on which are scheduled the developed applications. In this context, as above mentioned, we deployed two different distributed microservices based respectively on FTP and CoAP. These ones run into Docker containers orchestrated by Kubernetes. Containers, that are spread on the various cluster components belonging to different tiers, i.e on both virtual machines and Edge/IoT devices. They can be interconnected by means of different network overlays: OVN, Calico, Weave and Flannel. Figure 1 shows an high level representation of our system. In the following paragraphs, we briefly detail the main technologies necessary for the system's implementation.

Docker[17] is an open source platform for developers and system administrators that automates application deployment by leveraging container virtualization, such as cgroups and namespaces. Docker can encapsulate and execute an application in an isolated environment called container.

A. Kubernetes

Nowadays, users require 24-hour applications. The frequent development of new versions of such applications and the need to upgrade them can lead to their inactivity. Another important requirement is to intelligently scale them as demand grows. Kubernetes[18], is a platform designed as a highly available cluster whose purpose is to meet those requirements. Kubernetes allows to deploy and scale applications that, in order to decouple them from the underlying host - must be containerized. This approach is different from the previous one where deployed applications were installed on specific machines as deeply integrated packages in the host. Basically, as defined on the official website, *”Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications”* [19]. From a networking point of view, the components are connected as shown in the following figure 2. It also facilitates the load-balancing within Pods and ensures the use of a proper number of containers for our workloads. Kubernetes relies on external, third-party technologies for load balancing and networking.

Analyzing the Figure 2, the components inside the cluster communicate accordingly the following flow:

1) Kubernetes master interacts with etcd using HTTPS or HTTP in order to store all data regarding the cluster operations. It also contacts Flannel in order to have access to the containerized application;
2) Kubernetes nodes contact the Kubernetes master via HTTPS or HTTP in order to receive commands and report their actual status;

To understand how Kubernetes works, it is necessary to introduce its fundamental operation unit: the **Pod**. *”A Pod is a group of one or more application containers (such as Docker or rkt) and includes shared storage (volumes), IP address and information about how to run them”* [20]. In other words, a Pod encapsulates a group of containers that are scheduled on the same host. It represents the smallest planning, distribution and horizontal/vertical scaling unit that can be created and managed in Kubernetes. Containers placed in the same Pod can reach each other via local addresses and furthermore can communicate between them using interprocess communications such as SystemV semaphores or POSIX shared memory. The same is not valid for containers located in separate pods that should use the network.

From a networking perspective, in Kubernetes, it is possible to identify three communication modes between the various components:

a) Pod-to-Pod Communication: In a Kubernetes cluster, to each Pod is assigned an IP address in a flat shared networking namespace that can be accessed by all other Pods and hosts within the cluster. This approach allows us to consider Pods as physical or virtual hosts.

b) Pod-to-Service Communication: The IP addresses assigned to each service belong to a pool of addresses configured in the Kubernetes API Server through the `service-ip-range` flag. Basically, services are deployed using virtual IP addresses accessible by various clients and are intercepted by a kube-proxy process, running on all hosts, in order to route the traffic to the correct Pod. A Service is an abstraction that defines a logic set of Pods and a policy to access them (for example, load balancing). Services are critical to managing a Kubernetes cluster, the group of pods covered by the service is specified using a label selector.

c) External-to-Internal Communication: Previous communication modes allow to access a Pod or a service within the cluster. In order to access it from outside the cluster it is necessary to adopt an external load balancer. Indeed, the load balancer is able to recognize and route each request to the appropriate Pod by means of the kube-proxy service.

Coordinating dynamically allocated ports is very difficult task. Mainly, to each application is associated a port number

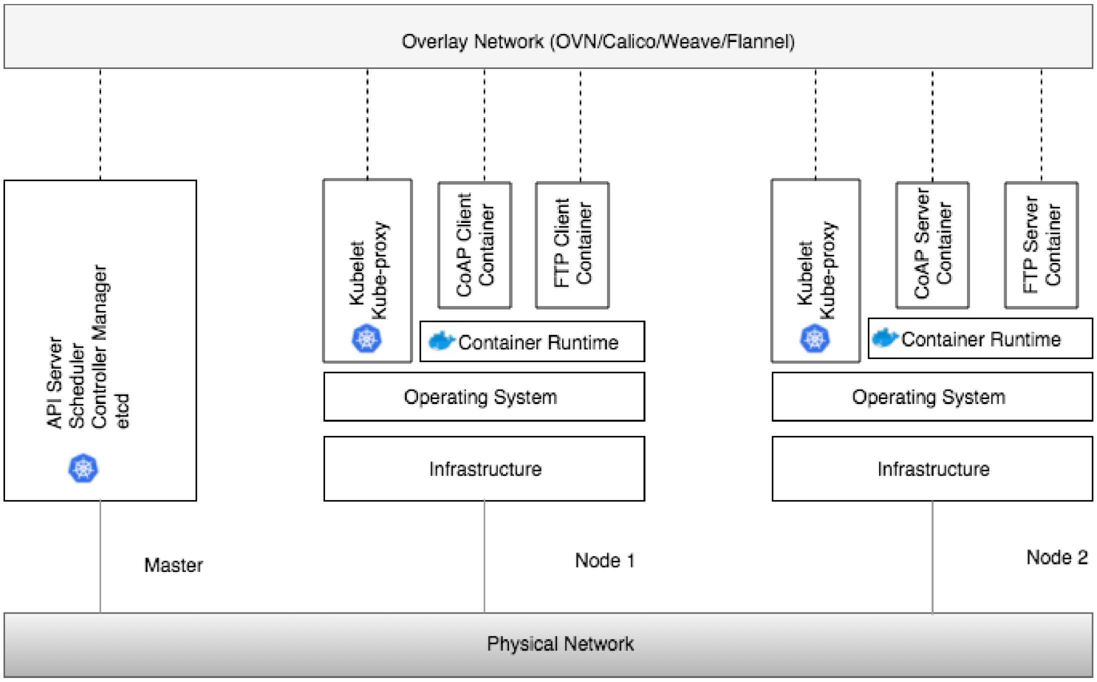

Fig. 1: Kubernetes cluster high level architecture.

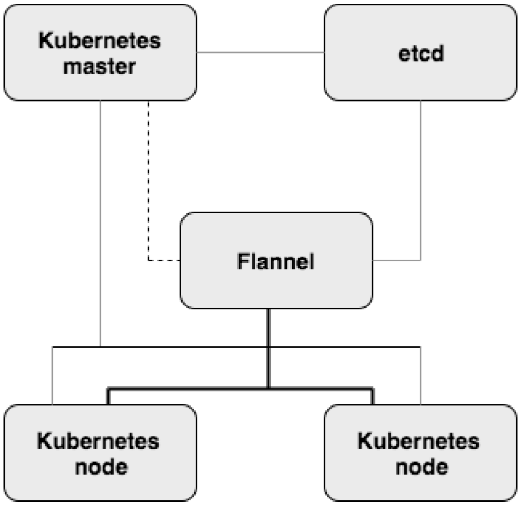

Fig. 2: Kubernetes components

and therefore the API server must insert into the configuration bocks a mapping between the dynamic allocated port and the respective application. In this way, as the number of applications increases, this becomes very challenging. To overcome these disadvantages, Kubernetes uses a different approach that requires each implementation to meet certain requirements:

1) all containers can communicate with each other without NAT;
2) all nodes can communicate with all containers (and vice-versa) without NAT;
3) the IP address of all containers belonging to the same pod are the same.

These assumptions allow to affirm that all containers within a Pod share the same namespace and are reachable each others by means of localhost address. *”This model is not only less overall complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers.”* [21] This does imply that containers within a Pod must coordinate port usage, but this is not different than processes in a VM. We call this the IP-per-pod model. In order to implement this networking model, Kubernetes does not natively provide any solution for multi-host networking. It relies on third-party plug-ins: Flannel, Weave, Calico and OVN. In the following, a brief description of each overlay is provided.

B. Flannel

Flannel[22] is a networking solution for Kubernetes developed by CoreOS. It can be used as an alternative to existing software defined networking solutions. Basically, Flannel assigns to each host an IP subnet and, using this subnet, Docker allocates IPs to each container.

Flannel has two different network models:

a) UDP backend: which is a simple IP-over-IP solution. It uses a TUN device to encapsulate each IP fragment in an UDP packet, thus forming an overlay network. The UDP encapsulation adds 50 extra bytes to the frame.

b) VxLAN: is faster than the UDP backend. VXLAN extends the data link layer (OSI Layer 2) by encapsulating MAC-based layer 2 Ethernet frames with network layer (OSI Layer 4) UDP packets. From an encapsulation overhead perspective, VXLAN is more efficient when compared to UDP.

C. Weave Net

Weave Net[23] is a powerful Cloud native networking tool that creates virtual networks for connecting Docker containers among multiple hosts. It enables also the automatic resources service discovery. Weave Net has two different connection modes:

a) sleeve: mode implements an UDP channel to traverse IP packets to containers. The main difference between Weave Net sleeve mode and Flannel UDP backend modes is that, Weave Net treats all the containers packets as an unique total packet and transfers them via an UDP channel, so Weave Net sleeve is technically faster than the UDP backend mode;

b) fastdp: mode that implements also a solution based on VxLAN.

D. Calico

Project Calico[24] plug-in, introduced from the 1.0 release of Kubernetes, implements a pure Layer 3 routed networking for all Kubernetes Pods present into the cluster. Basically, Calico offers a simple, scalable and secure virtual networking. It uses BGP, allowing at the same time to seamlessly integrate the Kubernetes cluster with existing data centers infrastructures. The pure Layer 3 networking avoids the packet encapsulation present in the Layer 2 solution in order to simplify diagnostics, reduce transport overhead and improve performances. The result is a high scalable performance networking that is simple to deploy, diagnose and operate.

Calico creates a highly scalable network with high throughput rates since in contrast to most virtual networks solutions, it provides a flat IP network that runs without encapsulation (no overlays). Whether is preferred an overlay network, Calico's network includes the ability to route packets using a stateless IP-in-IP overlay.

E. Open Virtual Network

Open Virtual Network (OVN)[25], a sub-project within OVS, was announced in early 2015 and has just recently released the first production ready version, version 2.6. OVN enables support for virtual network abstraction by adding native OVS support for L2 and L3 overlays and extends the existing OVS capabilities by adding native support for virtual network abstractions and security groups. Actually OVN can be used as plug-in for OpenStack[26], Docker and Kubernetes. OVN features include:

- virtual networking abstraction for OVS, implemented using L2 and L3 overlays, but can also manage connectivity to physical networks;
- flexible ACLs implemented using flows;
- software-based L2 gateways;
- networking for both VMs and containers, without the need of a second layer of overlay networking;
- native support for NAT and load balancing;
- native support for distributed L3 routing using OVS flows, with support for both IPv4 and IPv6;
- L3 gateways from logical to physical networks.

In terms of L3 features, OVN provides a so called distributed logical routing. L3 features provided by OVN are not centralized, so this means that each host executes L3 function autonomously. Mentioned plug-ins are installed on top of a tunnel-based (VXLAN, NVGRE, Geneve, STT, IPsec) overlay network. Furthermore, OVN also implements software gateways to realize the physical-logical network integration. In L2 configuration, OVN provides a logical switch. Specifically, OVN creates a L2-over-L3 overlay tunnel among hosts automatically. OVN uses Geneve as default encapsulation mode.

It can be configured in two different working ways: underlay mode and overlay mode.

a) Underlay mode: OVN's underlay mode requires an OpenStack setup in order to provide container networking. In this mode, thus, one can create logical networks and can have containers running inside VMs, standalone VMs and physical machines connected to the same logical network. This is a multi-tenant, multi-host solution.

b) Overlay mode: Unlike the underlay mode, in this mode, OVN creates a logical network amongst the containers running on multiple hosts. In this operating mode, OVN programs the Open vSwitch instances running inside VMs or directly on bare metal hosts.

IV. Workflow

In this section we present the methodology behind the implementation of our proposed system. To support such a flexible environment a public Italian Cloud provider (GARR, that is based on OpenStack) acts as an IaaS layer, while Kubernetes acts as a container orchestration tool. This configuration allows us to achieve applications and services orchestration in a lightweight and flexible manner. In Figure 4 we present the prototypical scheme of our deployed system. In particular, we discuss the main steps needed for Kubernetes cluster configuration and respectively for microservices design and deployment. Specifically, we suppose to realize and test the implemented microservices (FTP and CoAP) in order to perform the benchmarks illustrated in the Section V.

A. Achieving Microservices with Containers

Therefore, in order to implement our microservices, the starting point is represented by the creation of the Docker images.

In order to accomplish this task, as shown in the Fig. 3, for each level of a specific microservice, a Dockerfile has been created. This one contains all the commands needed in order to automatically create the relative Docker images. In the listing 1 is shown a completed Dockerfile snippet necessary for the image creation of the CoAP server.

Listing 1: Dockerfile snippet - CoAP Server.

```
FROM centos
MAINTAINER Alina Buzachis
ADD CoAPthon-master /dir
ADD files /dir
WORKDIR dir
RUN python setup.py install
CMD ["python", "coapserver.py", "-i", "127.0.0.1",
    "-p", "5683"]
```

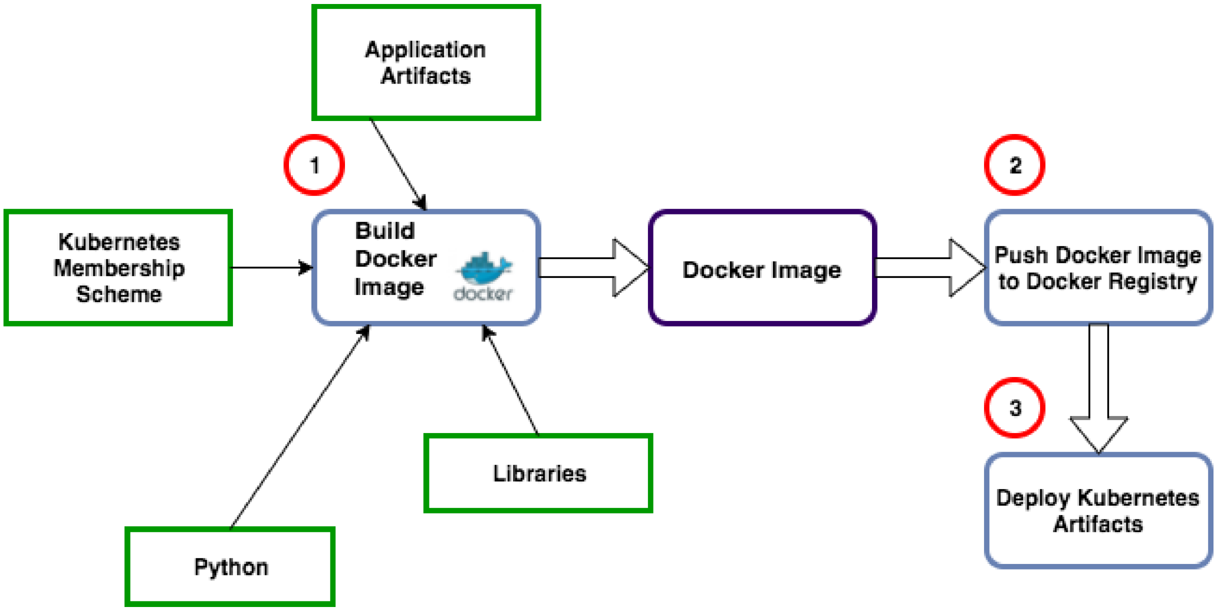
Application Artifacts
1
2
Kubernetes Membership Scheme
Build Docker Image
docker
Docker Image
Push Docker Image to Docker Registry
3
Deploy Kubernetes Artifacts
Libraries
Python

Fig. 3: Microservices deployment flow.

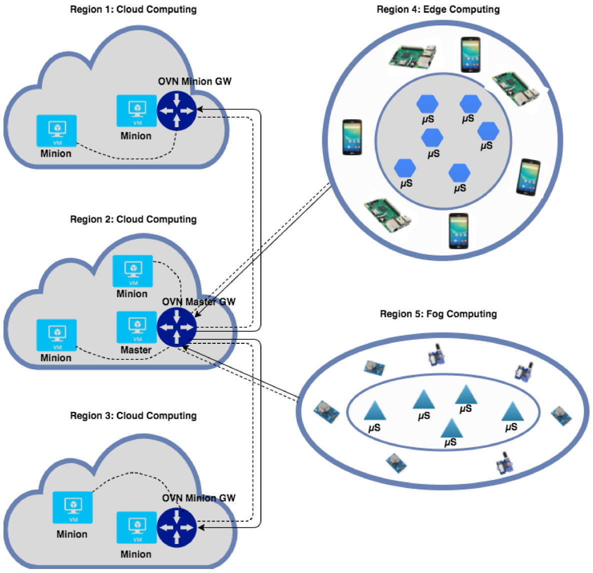

Fig. 4: Kubernetes cluster with OVN in L3 configuration.

After executing the Dockerfile, we can execute the resulting image, having a real running container. Therefore, the image must be loaded on the Google Container Registry through the `push` command, enabling recovery, scalable and simple development. Thus, the different images will be available in the Container Registry and ready to be orchestrated by Kubernetes.

Listing 2: CoAP pod YAML model.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: coapserver
  labels:
    apps: coapserver
spec:
  containers:
  - name: coapserver
    image: alinab/coapserver:alfa0.1
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 5683
  nodeSelector:
    name: minion1-amd64
---
```

Listing 3: CoAP Service YAML model

```
---
apiVersion: v1
kind: Service
metadata:
  name: coapserver
  labels:
    apps: coapserver
spec:
  type: NodePort
  ports:
  - protocol: UDP
    port: 5683
  selector:
    apps: coapserver
---
```

As above mentioned, the main Kubernetes component unit is represented by the Pod. It hosts our applications by encapsulating one or more containers relatively tightly coupled. Although each Pod has an unique IP address which is not exposed outside the cluster, it is necessary to introduce an additional abstraction - the Service. A Kubernetes Service defines a logical set of Pods and a policy by which to access them. The set of Pods targeted by a Service is determined by a Label Selector, e.g `apps: coapserver`. Services allow our applications to receive traffic. The NodePort type makes the Service accessible from outside the cluster exposing it on the same port of the selected Node in the cluster using NAT.

In order to create the Pod and Service objects, it is necessary to apply the `kubectl create -f` command to the respective files.

Furthermore, analyzing the Listing 2, we notice the CoAP server Pod is scheduled on the `minion1-amd64` node. In order to access the CoAP service externally, thanks to the `NodePort` mapping defined in the Listing 3, the `5683` port is mapped externally.

B. Cluster Configuration

Once the containers can be accessed and managed by our platform, we should instantiate the Kubernetes cluster composed by a master and several worker nodes, and therefore they are distributed on the various Pods. Also the creation of the cluster, thanks to the simplicity of using Kubernetes, can be done using Vagrant by executing the command: `$ vagrant up` At the end of this procedure, we will have a Kubernetes cluster consisting of a master and Workers communicating each others through a proposed network overlay. On the latter, using the `kubectl` command line interface, it is possible to deploy and test the relative microservices. In this context, we developed five different scenarios with or without the adoption of the network overlay.

In the first scenario we interconnected two container without the adoption of any overlay, but simply mapping them directly to host ports. In the other scenarios, we configured the system's network by applying the following network plug-ins: Flannel, Open Virtual Network (OVN), Weave Net and Calico. Among them, OVN is of particular interest, indeed it is able to create L3 gateways. In order to have this kind of component, it is necessary to configure on each cluster node a gateway (as shown in the Figure 4). In this way, it is possible to pin the Pod's subnet traffic to go out of a particular gateway. Additionally, the master's gateway is configured to control the Pods belonging to a specific subnet. In this way, the flow is directed to go out through master's gateway. In order to do this, we provide `--rampout-ip-subnets= "10.10.2.0/24,10.10.3.0/24"` option to the `gateway-init` command as shown is Listing 4 during the gateway's initialization phase. Analyzing the gateway's initialization script in Listing 4, it is possible to notice that we share a single network interface for both management traffic (e.g. ssh) as well as for the cluster's North-South traffic. Specifically, we attach the physical interface to the OVS bridge. Indeed, if we choose `enp0s9` as primary interface, with the IP address `PUBLIC_IP`, we then create a bridge called `br-enp0s9`, add `enp0s9` as port and move the `PUBLIC_IP` address to `br-enp0s9`. Then, we add the subnets that this gateway should manage and a default gateway's IP address `GW_IP`.

Listing 4: Gateway initialization script.

```
sudo ovn-k8s-overlay gateway-init \
  --cluster-ip-subnet="10.10.0.0/16" \
  --bridge-interface br-enp0s9 \
  --physical-ip "$PUBLIC_IP"/"$MASK" \
  --node-name="kube-gateway-master" \
  --rampout-ip-subnets="10.10.2.0,10.10.3.0" \
  --default-gw "$GW_IP"
```

This is a very important feature because it represents the starting point at the basis of the implementation of a network flow segmentation, object of future work.

V. Experiments

In this section, we analyze the time performances of our deployed system. We executed two different kinds of experiments focused on the evaluation of network performances in terms of transfer time registered using the proposed networking solutions. As previously mentioned, we created two different services respectively based on FTP and CoAP. These experiments have the purpose both to calculate the overhead introduced by overlays and to make a comparison among different solutions. Unfortunately, we did not find any similar work in the scientific community in order to make a comparison among our system and others. In order to evaluate the performances, we collected 30 subsequent experiments and calculated mean time and confidence interval of 95% for both deployed services in five different configurations with increasing payloads.

A. Testbed Configuration

The whole system is deployed on GARR platform (an Italian operator that provides Public Cloud infrastructure like Amazon). Our testbed is composed by three VMs (a master and two workers) with the following HW/SW configurations: 4 VCPUs @ 3.2 GHz, RAM 8GB, OS Ubuntu Xenial 16.04 with Linux Kernel `4.4.0-98-generic` The whole cluster is deployed using Vagrant `1.9.1` and is composed by three VMs having Ubuntu 16.04 Xenial, 2 GB of RAM and 2 CPUs, while the virtualization part is provided by VirtualBox `5.0.40_Ubuntur115130`. For Kubernetes cluster configuration we used Kubernetes `1.7.5` version while for containers deployment we used Docker version `17.05.0-ce, build 89658be`. In our experiments, in terms of overlays, in order to apply and implement them we used:

- OVN based on the Open vSwitch `2.7.0-1` version - overlay mode, geneve tunnel type;
- Calico `v2.5`;
- Weave `v2.1.5` - pcap mode (sleeve);
- Flannel - `v0.7.1-amd64`- vxlan mode.

B. First Scenario: FTP

The purpose of our first analysis regards the scalability test for any configuration. Specifically, we increased the payload starting from 1MB up to 1GB as shown in the x axis of the figure 5.

In the same figure, the FTP experiment results are illustrated. We can observe that Flannel, compared to the case without overlay, introduces a higher overhead compared to

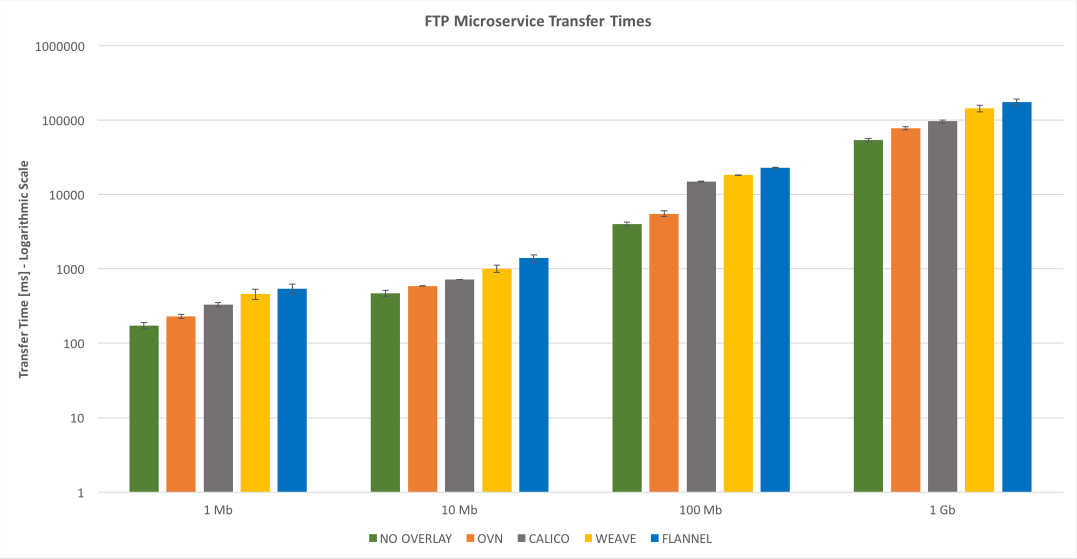

Fig. 5: FTP Microservice Transfer Times using OVN/Calico/Weave/Flannel overlays.

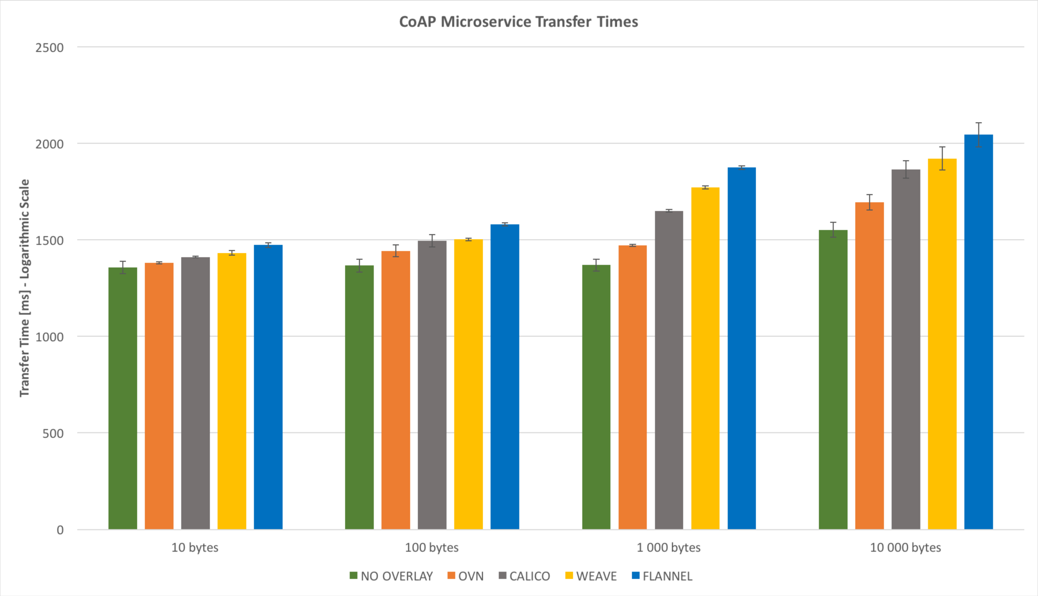

Fig. 6: CoAP Microservice Transfer Times using OVN/Calico/Weave/Flannel overlays.

other network solutions. Indeed, the solution based on Flannel presents slower transfer times for each payload. This overhead is a critical issue as it increases latency, such as compromising the quality of a real time session. A further improvement can be seen by using Weave, which, compared to Flannel, introduces a lower overhead. Calico, instead, presents time performances comparable to Weave but, the best solution, is represented by OVN. Indeed, it introduces less overhead in all configurations.

C. Second Scenario: CoAP

Also for the CoAP experiment, we performed a scalability test for each configuration. In these experiments, due to the nature of CoAP, which is not suitable for transferring large files but only for exchanging simple messages among hosts, we considered as payload 10, 100, 1000 and 10000 bytes files.

Figure 6 shows the behavior of the deployed service. Similar to the first case, Flannel introduces a bigger overhead compared to the case without any network overlay. Weave and Calico, in the experiments with the payload equal to 10 and 100 bytes present very similar time performances, but, the overhead introduced by Weave is bigger in other configurations. OVN, as in the FTP scenario, present better time performances in all configurations. It is important to notice that transfer times for experiments with payload equals to 10, 100 and 1000 bytes are very similar, this behavior is due to the size of the CoAP packet payload that is equal to

1024 bytes.

D. Experiment Summary

The overhead introduced by overlay networks is a critical issue especially in real time application. At first impact, as illustrated in the figures 5 and 6, we observe a considerable overhead introduced by Flannel. This could lead to problems in terms of *synchronization, response time, slow connections* and so on. These issues are key elements in the proper functioning of today's software. Taking the example of FTP microservices, a not negligible latency means introducing a quality deterioration, for example in a real-time session. While this aspect, in the case of the CoAP protocol, results in a lack of synchronism and subsequently failure to perform any quantifiable operation by the sensors present within a network. This could have a disastrous impact in situations where the change (through code injection) of a certain variable is required. In conclusion, from an implementation point of view, Flannel, Weave and Calico are easier and more intuitive than OVN, but that does not mean that the choice easier to implement is also the best. In fact, in order to choose the best overlay network we must evaluate the scenario type and complexity and respectively the design requirements, i.e. in a system where the implementation of gateways or ACLs is non required, we should exclude OVN and use an easier one, as well as in a scenario where specification on bandwidth usage or fast connections are not required, we also can use a simple one. This also applies vice-versa. Despite its complexity, OVN is the enabling technology for the realization of a dynamic osmotic network. Exploiting the OVN's features like L3 gateways and logic flows we are able to address a crucial network problem: the segmentation. We can select the path that must follow a particular type of packet and differentiate the streams: mice flows and elephant flows. By doing so, we obtain a colored network that allows us to optimize bandwidth consumption and limit latency. Furthermore, there is the possibility to dynamically change the route to be followed by certain packages, in this case congestion is limited.

VI. Conclusions and Future Work

In this scientific work we investigated an innovative solution to deploy distributed microservices among Cloud, Edge and IoT devices. In particular, we deployed two different services based respectively on FTP and CoAP inside Docker containers and orchestrated them by means of Kubernetes. In order to test our deployed system we made scalability analysis using four different network overlays.

Our tests show, that OVN is the best solution, indeed it introduces a smaller overhead compared to the case without any overlay. Despite its complexity, OVN is the enabling technology for the realization of a dynamic osmotic network, object of our future works. Indeed, by exploiting the OVN's features like L3 gateways and logic flows we are able to address a crucial network problem: the segmentation. We can select the path that a particular type of packet must follow and differentiate the streams: mice flows and elephant flows. By doing so, we obtain a colored network that allows us to optimize bandwidth consumption and limit latency. Furthermore, there is also the possibility to dynamically change the route to be followed by certain packages, in this case congestion is limited.

Acknowledgment

This work has been partially supported by FP7 Project the Cloud for Europe, grant agreement number FP7-610650.

References

[1] Gartner. https://www.gartner.com/newsroom/id/2970017.

[2] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, pp. 76–83, nov 2016.

[3] H. Benfenatki, C. Ferreira Da Silva, G. Kemp, A.-N. Benharkat, P. Ghodous, and Z. Maamar, "Madona: a method for automated provisioning of cloud-based component-oriented business applications," *Service Oriented Computing and Applications*, vol. 11, no. 1, pp. 87–100, 2017.

[4] A. Ceselli, M. Premoli, and S. Secci, "Mobile edge cloud network design optimization," *IEEE/ACM Transactions on Networking*, 2017.

[5] D. Pizzolli, G. Cossu, D. Santoro, L. Capra, C. Dupont, D. Charalampos, F. De Pellegrini, F. Antonelli, and S. Cretti, "Cloud4iot: A heterogeneous, distributed and autonomic cloud platform for the iot," in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, pp. 476–479, 2017.

[6] A. Sampaio, T. Rolim, N. Mendona, and M. Cunha, "An approach for evaluating cloud application topologies based on tosca," in *IEEE International Conference on Cloud Computing, CLOUD*, pp. 407–414, 2017.

[7] K. Yongsiriwit, M. Sellami, and W. Gaaloul, "A semantic framework supporting cloud resource descriptions interoperability," in *IEEE International Conference on Cloud Computing, CLOUD*, pp. 585–592, 2017.

[8] A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari, "Boss: A multitenancy ad-hoc service orchestrator for federated openstack clouds," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 351–357, Aug 2017.

[9] M. Villari, A. Celesti, G. Tricomi, A. Galletta, and M. Fazio, "Deployment orchestration of microservices with geographical constraints for edge computing," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, pp. 633–638, July 2017.

[10] A. Galletta, O. Ardo, A. Celesti, P. Kissa, and M. Villari, "A recommendation-based approach for cloud service brokerage: A case study in public administration," in *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pp. 227–234, Oct 2017.

[11] A. Mayoral, R. Vilalta, R. Munoz, R. Casellas, R. Martinez, M. Moreolo, J. Fabrega, A. Aguado, S. Yan, D. Simeonidou, J. Gran, V. Lopez, P. Kaczmarek, R. Szwedowski, T. Szyrkowiec, A. Autenrieth, N. Yoshikane, T. Tsuritani, I. Morita, M. Shiraiwa, N. Wada, M. Nishihara, T. Tanaka, T. Takahara, J. Rasmussen, Y. Yoshida, and K.-I. Kitayama, "Control orchestration protocol: Unified transport api for distributed cloud and network orchestration," *Journal of Optical Communications and Networking*, vol. 9, no. 2, pp. A216–A222, 2017.

[12] R. Martinez, A. Mayoral, R. Vilalta, R. Casellas, R. Munoz, S. Pachnicke, T. Szyrkowiec, and A. Autenrieth, "Integrated sdn/nfv orchestration for the dynamic deployment of mobile virtual backhaul networks over a multilayer (packet/optical) aggregation infrastructure," *Journal of Optical Communications and Networking*, vol. 9, no. 2, pp. A135–A142, 2017.

[13] G. Fioccola, P. Donadio, R. Canonico, and G. Ventre, "Dynamic routing and virtual machine consolidation in green clouds," in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, pp. 590–595, 2017.

[14] G. Fortino, R. Gravina, W. Russo, and C. Savaglio, "Modeling and simulating internet-of-things systems: A hybrid agent-oriented approach," *Computing in Science Engineering*, vol. 19, no. 5, pp. 68–76, 2017.

[15] G. Fortino, W. Russo, C. Savaglio, W. Shen, and M. Zhou, “Agent-oriented cooperative smart objects: From iot system design to implementation,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. PP, no. 99, pp. 1–18, 2017.

[16] C. C. Chang, S. R. Yang, E. H. Yeh, P. Lin, and J. Y. Jeng, “A kubernetes-based monitoring platform for dynamic cloud resource provisioning,” in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6, Dec 2017.

[17] “Docker.” http://docker.io/.

[18] “Kubernetes.” http://kubernetes.io/.

[19] Kubernetes, “Kubernetes,” 2017.

[20] “Viewing pods and nodes,” 2017.

[21] Kubernetes, “Cluster networking,” 2017.

[22] “Flannel.” http://coreos.com/flannel/.

[23] “Weave net.” http://www.weave.works/oss/net/.

[24] “Calico.” http://www.calicolabs.com/.

[25] “Open virtual network.” http://www.sdxcentral.com/projects/ovn-open-virtual-network/.

[26] “Openstack.” http://www.openstack.org/.