

Inter-Docker Cluster Communication Across Different Network Regions Using EVPN

MSc Research Project
Cloud Computing

Ravi Kumar
Student ID: x16132637

School of Computing
National College of Ireland

Supervisor: Vikas Sahni

National College of Ireland
Project Submission Sheet
School of Computing



Student Name:	Ravi Kumar
Student ID:	x16132637
Programme:	Cloud Computing
Year:	2018
Module:	MSc Research Project
Supervisor:	Vikas Sahni
Submission Due Date:	28/01/2019
Project Title:	Inter-Docker Cluster Communication Across Different Network Regions Using EVPN
Word Count:	XXX
Page Count:	33

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	26th January 2019

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Inter-Docker Cluster Communication Across Different Network Regions Using EVPN

Ravi Kumar
x16132637

Abstract

This era is recognized as an information technology era and cloud computing has brought revolution in it. Virtualization is considered as one of the key-enabling technology of cloud. Docker have become quite famous in virtualization because of its desirable features. Initially Docker deployment in cloud was challenge which was solved by Software defined Networking. The deployment of Docker in different geographical area has become necessity to provide redundancy and high availability. But inter-Docker communication leads to the problem of latency. This paper focuses to minimize the latency in inter-Docker communication. We propose an EVPN solution over VX-LAN for the communication between Docker located on different geographical area. This technology will minimize latency by connecting two different ethernet over MPLS.

Contents

1	Introduction	2
2	Theoretical Background	3
2.1	Cloud Computing (Domain)	3
2.2	Virtualization (Area)	3
2.3	Virtual machine (Ongoing Technology)	4
2.4	Docker Containers (Evolving Trend)	4
3	Related Work	4
3.1	Docker - Denial of Service attack	4
3.2	Network Virtualization built on Docker Engine	5
3.3	Distributed Cloud Monitoring built on	6
3.4	Optimization of Docker Containers	7
3.5	CPU scheduler for Docker container	8
3.6	Docker based Emergency communication system	9
3.7	Honeytrap systems built on Docker	10
3.8	Distributed software development tool built on Docker	10
3.9	Discussion	11

4	Methodology	12
4.1	Google cloud platform (GCP)	12
4.2	Eve-Ng tool	12
4.3	Lab setup	12
4.4	EVPN setup	13
5	Design Specification	13
6	Implementation	13
6.1	Google cloud platform (GCP) setup	13
6.2	Eve-NG tool setup	14
6.3	Lab setup	14
6.4	EVPN configuration	15
6.5	Docker configuration	15
7	Evaluation	16
7.1	Discussion	19
8	Conclusion and Future Work	19
9	Appendix - Configuration Manual	22
9.1	GCP setup	22
9.2	EVPN Configuration	22
9.2.1	Nexus Switch 1	22
9.2.2	Nexus Switch 2 Configuration	26
9.2.3	Docker Configuration	31

1 Introduction

Cloud Computing has created boom in the IT industry and evolved as a leading model. Network-centric content and Network-centric computing is demand of the era. (Buyya et al.; 2009) states that Cloud provisions the interconnection between parallel and distributed computing. Simulations are examples of Virtualization which is the method of building virtual from something real (Ageyev et al.; 2018). Virtualization is a beneficial tool in optimizing the resource utilization, logical administration and minimizing the infrastructure running costs.

Docker technology works on operating system virtualization (Wu and Yang; 2018). One or more services can be run on Docker at a time. There can be a single service running on multiple Docker to provide a merged output. The service hosted Docker can be on same or different locations. But the services can be impacted due to latency when the Docker are hosted on different geographical locations.

Research Question: Can latency be enhanced by deploying Docker- container cluster across different network regions via Ethernet Virtual Private Network (EVPN) ?

Docker container is evolving technology which helps in quick resource provisioning, allows flexibility to pack and move programs and allows to run apps on legacy infrastruc-

ture. It had provided an ease to developers, without worrying about the deployment environment they can build the applications also saves time and money for organizations in migrating application in its different environments such as testing, development and deployment.

In this paper, the above stated research question will be assisted and the paper is organized as follows - section 2 provides the theoretical background, section 3 focuses on the related work done in the segment, section 4 proposes the methodology, section 5 provides the design specification, section 6 focuses on implementation, section 7 based on evaluation and section 8 provides the conclusion and the future work.

2 Theoretical Background

This section provides theoretical background for the content of this research paper. The topics covered in this section are Cloud Computing - domain of the paper, Virtualization - area of the paper and Docker containers - topic of the paper. Virtual machine is the ongoing technology in the IT industry which are being replaced by Docker containers.

2.1 Cloud Computing (Domain)

Cloud Computing provides a platform which facilitate the customers to rent IT services like compute, storage etc via internet. As per NIST, its five characteristics are - Broad Network access, Measured service, Rapid elasticity, Resource pooling and On-demand self-service (Swenson; 2011). The consumer rents the IT service instead of buying it. Cloud Computing has four service deployment models (Public, private, hybrid and community) and three delivery models (Software as a service, Infrastructure as a Service and Platform as a service) (Al-Lawati and Al-Badi; 2016).

2.2 Virtualization (Area)

Virtualization is a technology used to abstract the real resources from the user (Ageyev et al.; 2018). Resources are merged or multiplexed in virtualization such as RAID, virtual memory address. Resources located in different geographical area can be pooled up via cluster of servers (Zhang; 2018). Platform and resource virtualization are mainly two types of virtualization. Emulation, Containerisation and Hypervisor usage are the approaches to obtain Platform virtualization (Zhang; 2018). Docker containers are type of Containerization platform.

Hypervisor acts an interface between guest OS and hardware communicating between them. OS, Para and Full virtualization are the techniques to share physical resources. There is no modification required in hardware for full virtualization as well as guest OS has no information about the hardware. Para virtualization comprises with few modifications as well as guest OS is aware of hardware. OS virtualization doesnt consider hypervisor layer and has ability to launch virtual machine above the host (Blenk et al.; 2015).

2.3 Virtual machine (Ongoing Technology)

Virtualization is used to create virtual machines on top of hypervisors by OS disk images above bare metal/hardware (Blair et al.; 2017). The implementation of vm depends on their usage and they are like compute system emulations. Native vm communicates directly with the bare metal. VMs are terminated after completion of process and created by process initialization. Each VM has its separate OS and allocated resources. Now as we discussed Virtual machines, we will move forward with which are getting popular over vms day by day.

2.4 Docker Containers (Evolving Trend)

Docker containers are quite light-weighted virtualization techniques instead of VMs (Nadgowda et al.; 2017). Linux kernel of host is used in to run vms on the host (Kovács; 2017). doesnt emulate the physical resources but act as a wrapper for resource accounting and isolation. Cgroups and Linux namespaces are used by Containers. Network ID, Process ID etc are used in Linux namespaces for isolation and for resource management and accounting Cgroups are used. The performance comparison shows containers are better than vm (Felter et al.; 2015).

The next section provides overview of research work done in the segment till date.

3 Related Work

This section provides insights of relevant research works already done in the Docker segment which are - Docker communication, monitoring, automation, security and performance. Note that Table 1 in section 3.9 Discussion provides deeper insight, comparison and critical analysis of present research works on the basis of their achievements and goals. However, no existing research work directly covers inter-Docker communication but it is required for high availability, fault tolerance and QoS.

3.1 Docker - Denial of Service attack

(Chelladhurai et al.; 2016) proposes solution for Docker container Dos attacks. The motivation behind the solution is that one of the vulnerable part of Docker is Control groups, which can become base of four critical attacks (Chelladhurai et al.; 2016). Host kernel is shared by the containers. An attacker can break from malicious containers to the host machine to perform Dos attacks. Network Bridge is the interface between Containers to send and receive packets. Network Bridge has no packet filtering mechanism, hence arp flooding and mac spoofing can be done. Directories can also be responsible for attacks as they are shared with guest container and Docker host has full rights to make changes. Docker Daemon and host integrity can be also compromised as the Docker image comes from insecure sources.

Best practices and existing security solutions are also proposed in the paper. Security hardening is done by solutions such as AppArmor, GRSEC, Seccomp etc. The secure communication with Docker Daemon is established by Unix and REST API. Docker security policies offered by Selinux can be used by users to start container processes. Mandatory Access Control (MAC) should be implemented so that guest containers cant bypass system security policy.

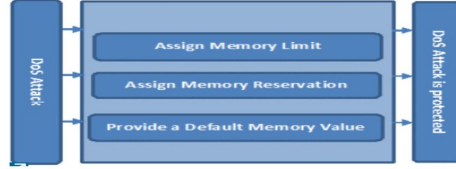


Figure 1: Prevention Mechanism (Chelladhurai et al.; 2016)

The above Figure 1 describes the proposed solution for the DoS attacks which consists three stages in which memory limit is assigned, memory is reserved on advance basis and memory has been given a default value.

The experiment results suffices the aim of the paper but only Dos attacks has been covered in the paper, any other attacks hadn't been addressed. The author also doesnt comment about the communication of Docker which is our research goal.

3.2 Network Virtualization built on Docker Engine

(Xingtao et al.; 2016) proposes Software defined Networking controller platform built on Docker engine for network virtualization. The motivation behind the platform is the deployment, development and testing speed in network virtualization is limited. The platform will help to implement changes on large scale by creating, deploying and modifying virtual resources rapidly according to requirements.

The current SDN controllers available such as Floodlight (Big Switch Networks), Ryu (NTT Japan), Open Contrail (Juniper) and Opendaylight (open source) are restricted to fast, scalable and dynamic requirements.

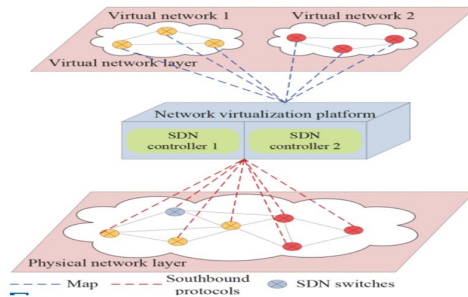


Figure 2: SDN controller based Network Virtualization (Xingtao et al.; 2016)

The above Figure 2 defines the network virtualization logical structure based on SDN controller. Network resources and routing capabilities are managed by SDN controller. There are three modules basic, application and REST module in Docker engine. Event, manage, address, packet and controller are the components of basic module. Router, topology component and configuration is managed by REST application module. Tunnel, arp component, route, switch, forwarding and vlan is managed by Application module. Virtualization platform is first created for network virtualization implementation. The image built for controller is network-specific for Docker-container implementation on hardware platform. The abstraction, virtualization and other activities are controlled and performed by the image on hardware resource. The control and data plane are separated to avoid contradiction in controllers.

The research work covers only routing functionality but leaves other modules of the SDN controller. In addition, the SDN controller was based on but author doesn't provide information about communication between which leaves our research question unanswered.

3.3 Distributed Cloud Monitoring built on

(Dhakate and Godbole; 2015) proposes Distributed cloud monitoring based on Docker for monitoring of cloud services. The motivation behind the solution is cloud portals can be in large numbers as well as located in different geographical regions and monitoring manually is difficult and tiresome process. The customer uses the portal to manage the resources offered by the cloud vendor. The CP nodes are located physically in different locations in consideration to be nearest to the customer for reducing the latency. The manual monitoring process can be lengthy due to large number of portals as well as output containing high ratio of error. There are tools such as OLA available for automation but speed is the issue. The monitoring process of OLA includes spinning of VMs in different locations, which further installs app, libraries, OS and scripts making it slow, resource-consuming and costly.

The next generation virtualization platform Docker is used for solution (Anderson; 2015). The approach used for monitoring locally is distributed monitoring.

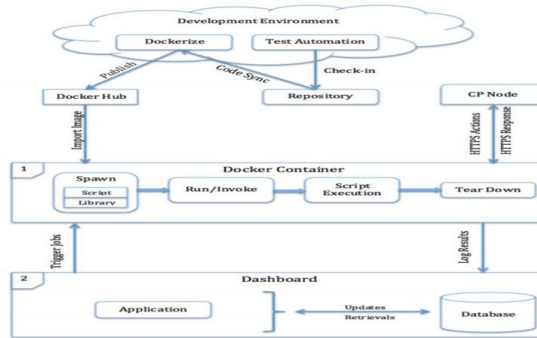


Figure 3: Network Virtualization using SDN controller (Dhakate and Godbole; 2015)

Figure 3 shows the logical diagram of cloud monitoring dashboard modules and its architecture. The local Docker and test suite is spun up by the scheduler. The images are imported from Docker Hub. The output from the logged results captured from the nodes after running the scripts are sent to the Dashboard.

The experiment results verify the aim of the research but fault tolerance capabilities are not addressed by the author and left for the future work. The author also doesn't provide any information for the inter Docker communication which is our research aim.

3.4 Optimization of Docker Containers

(Sureshkumar and Rajesh; 2017) proposes Docker container optimization via energy-aware algorithm and system. The motivation behind the optimization algorithm is due to unmonitored running containers, either they are over-utilized or under which impacts the performance, health of physical devices as well as energy consumption. To cope with the problem author has used simple idea to spin up new instance when required and kill them when requirement decreases.

The algorithm is focused to run maximum containers on their standard limits. The first action of algorithm is to send the under-utilized containers on sleep state after transferring their load to some other less loaded containers. The second step moves idle containers to sleep state and they are moved back to active state when required as per the load. The third step involves re-initiating the containers from sleep to active state when load grows up.

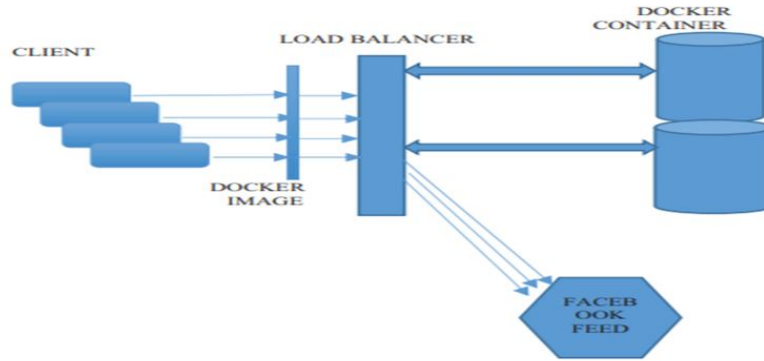


Figure 4: Energy-aware system Architecture (Sureshkumar and Rajesh; 2017)

The above Figure 4 describes the four modules of the solution architecture (Sureshkumar and Rajesh; 2017). CPU and RAM utilization is displayed by container management status which is retrieved by web application module hosted in jboss server. The jobs are allocated to the Docker by energy aware scheduler part of second module. The third module is responsible for migrating containers from sleep to active state after evaluating the requirement as per the load. The configuration of Docker and its image creation is done by the fourth module for the jboss server.

The presented outputs of the experiment proves the successful implementation of the goal. However the solution focuses on overloaded containers but light and medium loaded containers are ignored. In addition the solution is also focused on single system for multiple instances creation but for multiple systems is left for future work. If the will be on multiple systems they need to communicate for the services which is the goal of our research work and remains unanswered.

3.5 CPU scheduler for Docker container

(Wu and Yang; 2018) proposes dynamic scheduling of Docker containers via Flexible Deferrable Server (FDS). Docker container systems run mixed criticality systems consisting real time as well as non-real time applications. The motivation behind cpu scheduler is that non-real time application cpu needs are not static but the cpu allocation done in by default are static. This leads to performance degradation when there is high cpu requirement and to cope up with the problem FDS is proposed.

Upper bound guarantee (UBG) and Weighted fairness (WF) are the two strategies used for cpu allocation in which is known as Completely Fair Scheduler (CFS) (Wu and Yang; 2018).

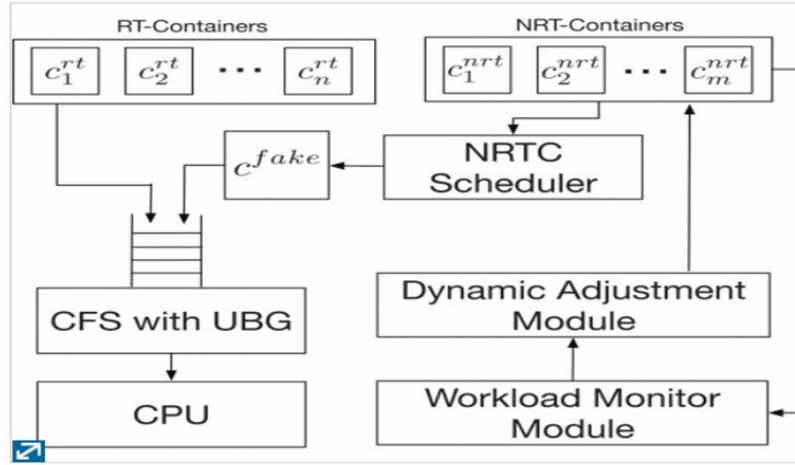


Figure 5: FDS scheduler architecture (Wu and Yang; 2018)

The above Figure 5 defines the FDS scheduler using UBG strategy for CPU scheduling. The real time workload gets allocated cpu by the scheduler to enhance the performance on the first place. Then the scheduler allocate all the remaining CPU to a fake container. On later stage the remaining cpu allocated to fake container is allocated to the non-real applications dynamically as per their requirement. The fake container acts as deferrable server in this scenario.

The experimental results proves the successful implementation of the goal by allocating the resources dynamically as per the need and increasing the performance. However, the author doesnt specify when there is no cpu available situation for allocation. There

is also not any given information about the communication between the which makes our research question unsolved.

3.6 Docker based Emergency communication system

(Pentyala; 2017) proposes communication app using Docker for emergency situations such as natural disasters. The motivation behind the communication app is every year in different part of world people get trapped in natural disasters and communication gets failed. The solution aims to help in management of resources, re-uniting family members, co-ordination in volunteers etc. The basic idea is to launch an editable map using OpenStreetMap (OSM).

The app helps to check uploaded data, sensor data, present locations etc featured on OSM based user interface. The needy person can check the map and know their help as per the uploaded data. These setup will deployed on Docker containers. In experiment, for hardware raspberry pi 3 is used in which 2 interfaces has been allocated to each pi acting as ad-hoc network and access point.

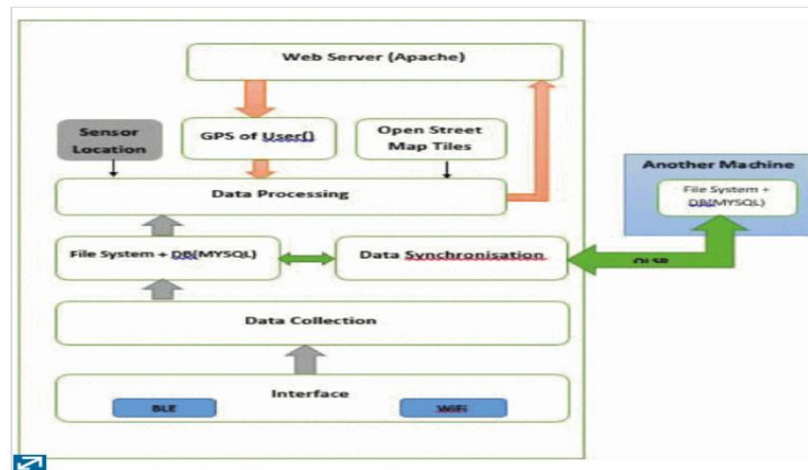


Figure 6: Communication System (Pentyala; 2017)

The above Figure 6 describes the logical architecture of the communication system which has five modules. Data is collected from sensors as wells as users and stored in Data collection module. The data is merged and stored in Openstreet Maps database in the form of files by Data processing module. The syncing of data with respective pi is done by Data synchronization tool. The location of users is accessed by GPS. To avoid failure and provide resiliency, multiple Docker containers have been used.

The author proposed solution serves the goal of the research work. However, author hadnt provided any information for implementation and working of proposed multiple Docker containers for resiliency which is our research question because for that inter-Docker communication will be needed.

3.7 Honeypot systems built on Docker

(Sever and Kišasondi; 2018) proposes Docker based Honeypot infrastructure for evaluating security. The motivation behind this system is that cyber attacks is being common today which impacts various organization services even letting them complete down. The focus of Honeypots are to attract malicious attackers to know vulnerabilities or to distract them from valuable resources. The main challenge is to make Honeypots feel genuine to attract the malicious attackers. Some other challenges are security, flexibility and limited ip address (Chin et al.; 2009).

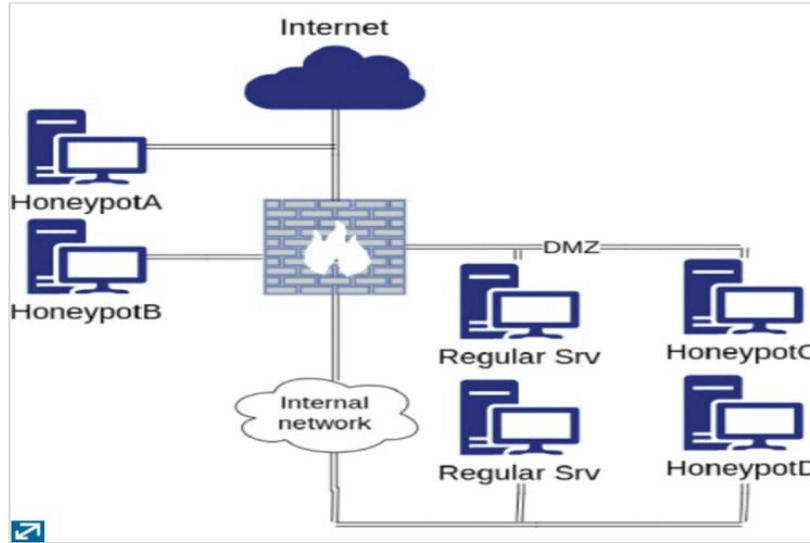


Figure 7: Honeynet model (Sever and Kišasondi; 2018)

The above Figure 7 describes the model architecture of Honeynet. The cluster of Honeypots are also known as Honeynets. Real time report generation, storage and centralized sensor control are features of Honeynet. The virtual Honeypods are better as there is less flexibility, leftover resources after deployment and time-taking process issues in physical honeypots (Sever and Kišasondi; 2018). as virtual Honeypods provide efficient solution from all these concerns.

The logs generated from the experiment shows various attacks such as Docker compromise, denial of service attack etc which serves the goals of creating Honeypods. The best security practices has also been provided in the research work but the author provides no solution for the detected vulnerabilities and leaves the same for future work. However, author talks to make the honeypods fault tolerant but provides no description to perform that as it needs inter-Docker communication which leads our research question still active.

3.8 Distributed software development tool built on Docker

(Naik; 2016) proposes distributed software development process on multiple cloud via virtual system of systems (SoS). The motivation behind the tool is to cope up with the Hybrid cloud scenario as to get benefits of both cloud environments clients are moving

towards Hybrid cloud. This will also ease customers stuck in vendor lock-in and provides high availability. Docker swarm by Docker can be considered as an solution for distributed system development tool but the given solution has a new approach based on multiple tools such as Docker swarm, Mac OSX, nginx, redis, VirtualBox etc (Naik; 2016).

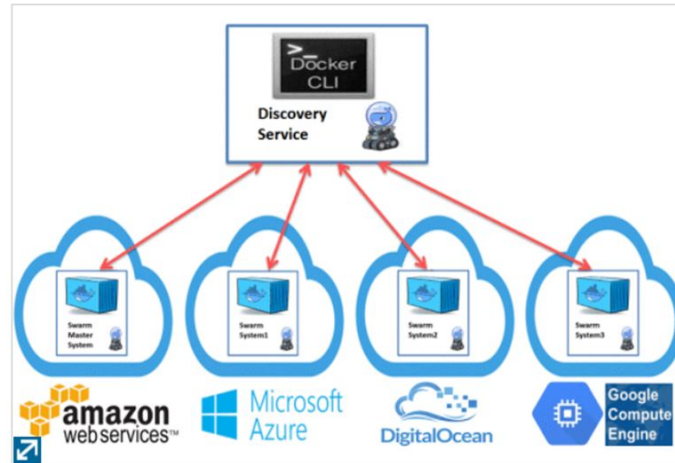


Figure 8: Virtual SoS (Naik; 2016)

The above Figure 8 defines the SoS architecture. Virtual box is used to host four swarm systems cluster to implement virtual SoS which supports any cloud which has Docker support. The nodes are created by creating Docker swarm image. Cluster form neighborhood among themselves via discovery token. After consensus, one node becomes master and rest three becomes slave to serve the request. Nodes can be managed from any cloud irrespective of their creation.

The research work results obtained from the experiment meets the goal. However, built infrastructure is on one physical device which can lead to single point of failure. The research work doesnt provide any details of failure as well as how the Docker nodes in different cloud will communicate which is our research question.

3.9 Discussion

Note that for better clarity, P1 refers to Docker - Denial of Service attack, P2 refers to Network Virtualization built on Docker Engine, P3 refers to Distributed Cloud Monitoring built on , P4 refers to Optimization of Docker Containers, P5 refers to CPU scheduler for Docker container, P6 refers to Docker based Emergency communication system, P7 refers to Honeypot systems built on Docker and P8 refers to Distributed software development tool built on Docker.

The characteristic elements compared are important for Docker Containers but there is no research paper directly on inter-Docker communication although these papers are related to implementation of and inter-Docker communication.

Table 1 : Important elements based on existing research work for inter-Docker communication approach

	Docker				
	Communication	Performance	Security	Automation	Monitoring
P1	x		✓		
P2	x	✓			
P3	x				✓
P4	x	✓			
P5	x	✓			
P6	x			✓	
P7	x		✓		
P8	x			✓	

As per the related work till now, either no approach has been taken for inter-Docker communication or left for the future work. So, we need a new approach for inter-Docker communication located in different geographical are. Hence, the next section.

4 Methodology

This section includes the components involved in lab building and methodology used to implement the research - Google cloud platform, Eve-ng tool, Lab setup, EVPN setup.

4.1 Google cloud platform (GCP)

GCP had been used for the cloud resources(*Google Cloud including GCP & G Suite Try Free*; n.d.). GCP only provides large instances with high compute and memory resources on free tier, so it's been considered for the project to avoid the charges. As per the project requirement, instance had been created on the platform to host the Eve-ng tool. The created instance has been hosted with Ubuntu image which supports nested virtualization.

4.2 Eve-Ng tool

Eve-ng tool is the first clientless network emulation tool which supports multi-vendor network based emulation (*Eve-NG*; n.d.).It's clientless emulator tool which supports a wide range of networking devices, servers etc. The user- interface is accessed via browser. This tool is implemented on the Google cloud platform created instance. Public ip has been assigned to access the user interface from the internet.

4.3 Lab setup

The Lab has been created on the Eve-Ng tool which includes two layer 3 devices acting as routers and 2 servers. In the current scenario Nexus series switches has been used as layer 3 devices and the servers has been used to deploy the Docker container - Portainer.

4.4 EVPN setup

The two layer 3 devices has been configured with routing protocols to share their paths. EVPN is configured on the both devices as well to pass the traffic and extend the vlan id. Dockers hosted on the first server will communicate with the other one hosted on the second server through the EVPN with minimum latency.

5 Design Specification

This section provides the specifications of the tools, os and other software used to implement the research work.

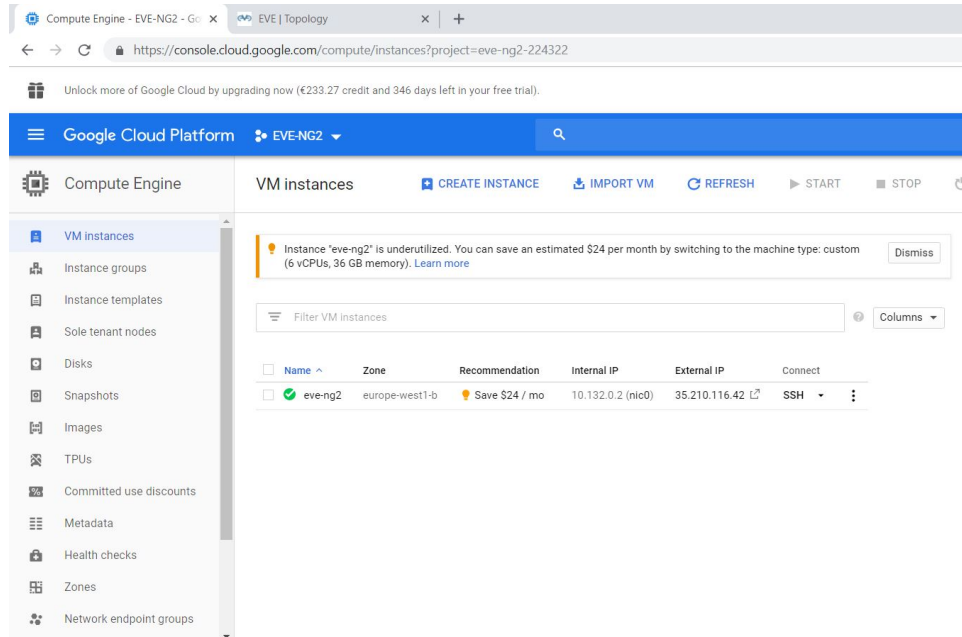
CLOUD VENDOR	Google Cloud Platform
INSTANCE SPECIFICATIONS:	Compute: 8 vCPUs
	RAM: 52 GB
	Boot disk: 300 GB
	OS: Ubuntu 16.04 LTS supporting nested virtualization
TOOL	EVE-NG : Community Edition
LAB SETUP	Layer 3 devices: Nexus 9000 series switches
	Server: Ubuntu 18.04
	Docker: Portainer/Portainer

6 Implementation

The implementation of this research work has been done on five segments in a sequential order - GCP setup, Eve-NG tool setup, Lab setup, EVPN configuration and Docker configuration. It is also focused on cost-optimizing regardless of Qos.

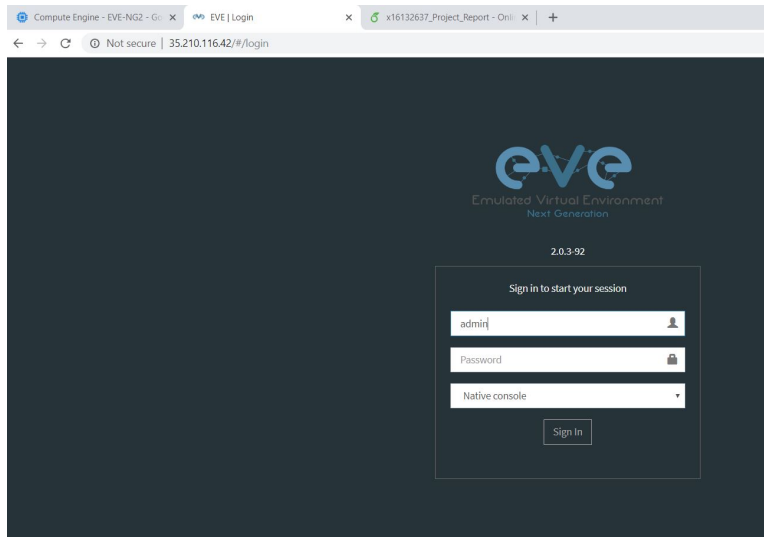
6.1 Google cloud platform (GCP) setup

In this scenario, as Nexus devices uses 16 Gb of memory and 4Gb for each servers and calculating other requirements, an GCP instance has been created with 52 GB RAM, 8 vCPU's and 300 GB boot disk. Eve-NG requires nested virtualization, so the OS uploaded on the instance is ubuntu 16.04 Lts supporting nested virtualization.



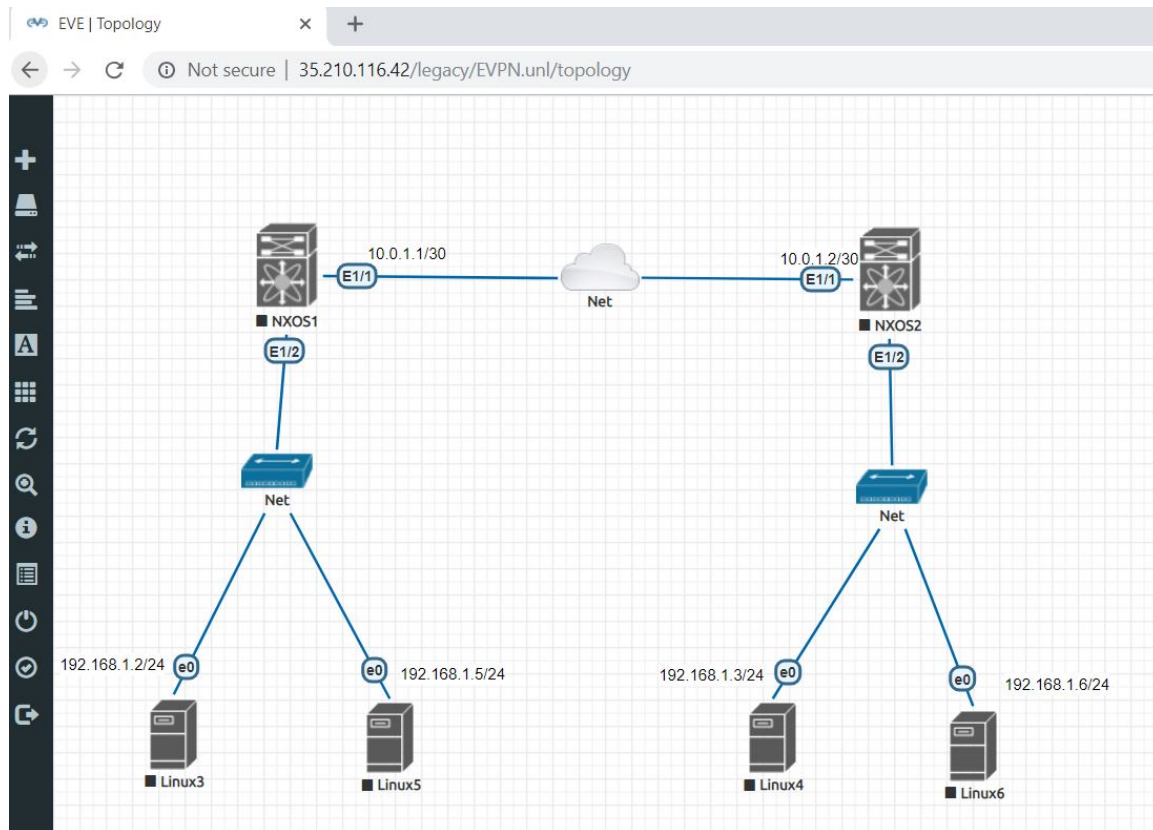
6.2 Eve-NG tool setup

The community edition of the Eve-NG has taken for the research work as its free to use for non-commercial purposes (*Eve-NG*; n.d.). The setup of the file has been downloaded from <http://eve-ng.net/downloads/eve-ng-2> and installed on the GCP instance. The GUI of the tool is then accessed via public ip of GCP instance which can be login through default credential admin as username and eve as password.



6.3 Lab setup

A new lab has been created naming Eve-NG2. The images of the nexus and other devices had been uploaded via Filezilla tool using the public ip of the instance in the Qemu folder of the Eve-NG tool. This allows access to the images and allows to create nodes in the lab. As per the topology, the lab is been created with required nodes.

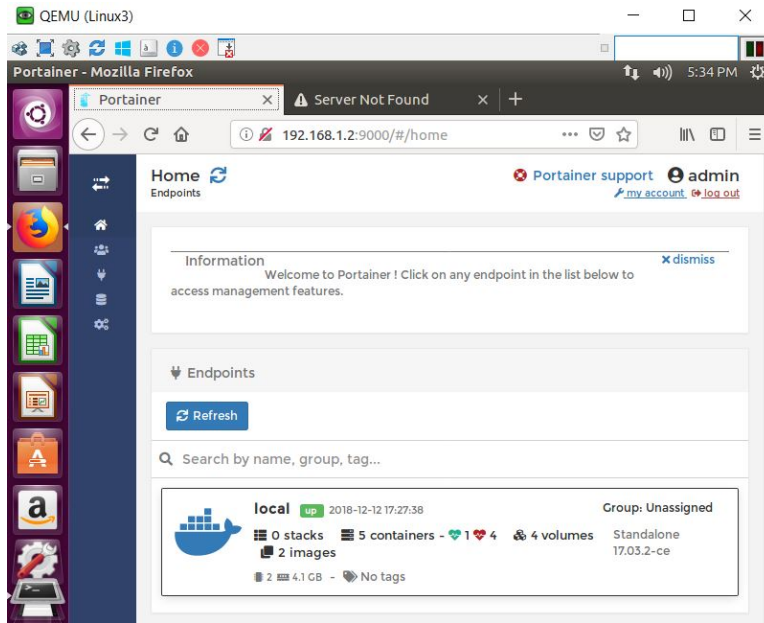


6.4 EVPN configuration

The interface ip of the nexus devices had been configured as per the topology. Then evpn had been configured on both the nexus devices one by one by the script shared in the configuration manual. The ip on server interfaces had also been configured as per the topology.

6.5 Docker configuration

After configuration of server network interfaces, we configured Docker containers on the linux server. To cross-verify the service, docker services status has been checked. Then the portainer Docker image is pulled from the Docker hub and hosted on the server (*portainer/portainer* - *Docker Hub*; n.d.). The communication port configured for the docker is 9000.



7 Evaluation

This section provides the detailed information of evaluation procedure followed and the extracted results from the implementation. The "PING" feature of ubuntu and windows OS has been used to get the latency. Ping is one of the most used commands for troubleshooting purposes. To get the latency, ping command is used to ping the docker hosted on second server(Linux4 - 192.168.1.3) on the other side from the first docker hosted on server(Linux3 - 192.168.1.2).The latency test has been carried in 6 iterations with varied packet size for better analysis.

Iteraion	Packet size
1	108
2	1008
3	10008
4	50008
5	60008
6	64452

The above table shows the different packet size selected for the latency test. Latency has been measured in milliseconds and packet size in bytes.

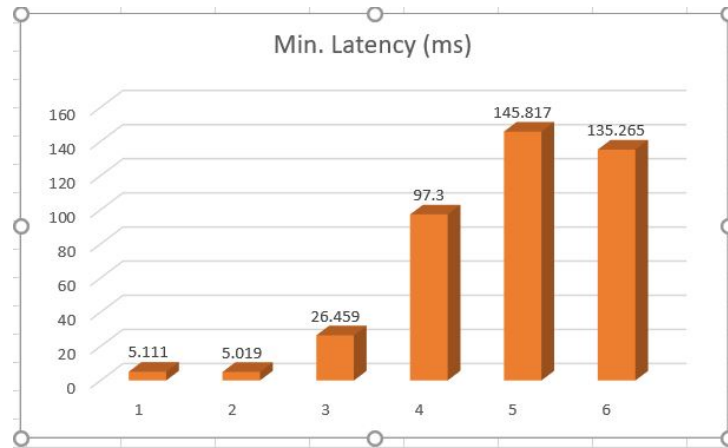


Figure 9: Minimum latency Graph

Figure 9 graph shows the minimum latency recorded in 6 iterations with variable packet size. The minimum latency recorded is 5.111 ms in the first iteration with smallest packet size and greatest of 145.817 ms in 5th iteration.

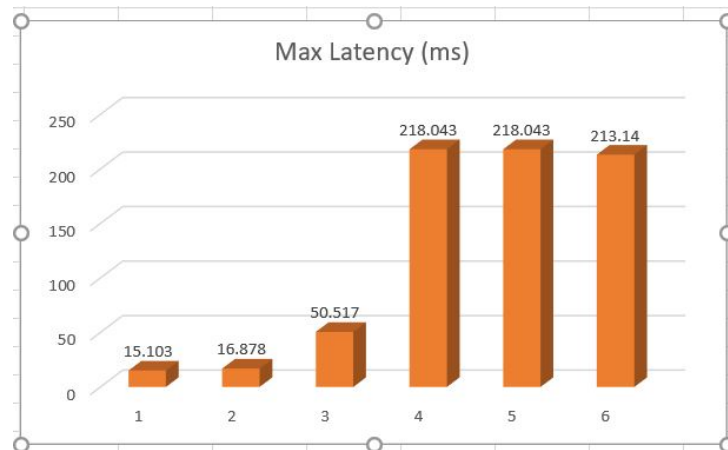


Figure 10: Maximum Latency Graph

Figure 10 graph shows the maximum latency recorded in 6 iterations with variable packet size. The maximum latency recorded is 15.103 ms in the first iteration with least packet size and greatest of 218.043 ms in 4th 5th iteration.

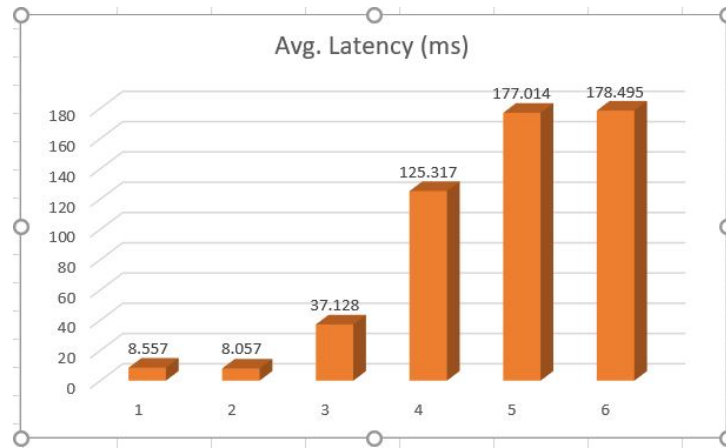


Figure 11: Average Latency Graph

Figure 11 graph shows the average latency recorded in 6 iterations with variable packet size. The average latency recorded is 8.557 ms in the first iteration with smallest packet size and greatest of 178.495 ms in 6th iteration.

As there is no research work done till date with inter-docker communication, to compare the results we had recorded latency in reachability from one end device/server to another server. We had taken my pc, as an end device and recorded the latency to reach the server hosted on Google cloud. Two iterations of result has been taken and compared.

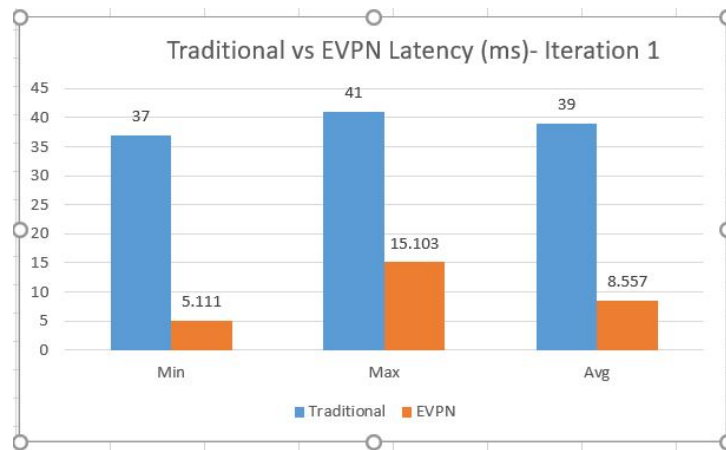


Figure 12: Latency comparison - Iteration 1

Figure 12 graph shows the latency variance in EVPN and normal scenario. The graph shows the min,max and avg latency recorded in iteration 1.

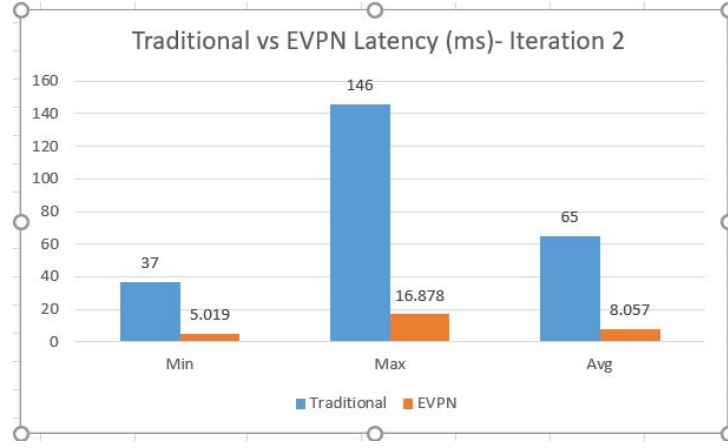


Figure 13: Latency comparison - Iteration 2

Figure 13 graph shows the latency variance in EVPN and normal scenario. The graph shows the min,max and avg latency recorded in iteration 2.

7.1 Discussion

This subsection covers critical analysis of output generated from the implementation of the research work. The above graphs clearly shows that the latency increases when the packet size increases. In both iterations of comparison, the graph clearly states that the latency in EVPN is quite smaller as compared to traditional communication. The min latency difference in both scenarios is approx 32 ms which is quite significant in terms of Qos.

Although evaluation had been done on live scenarios but results may vary on real environment as simulations may cause vary in actual latency. Latency also increases when the distance between location increases.

8 Conclusion and Future Work

This section includes overall insights of the research paper and future work in the segment. The research work concludes two main points. The latency increase with the increase in packet size as well as latency is quite less in EVPN scenario as compared to the traditional technologies. The goal of the research work to enhance the latency in inter-docker communication via EVPN is accomplished and verified by the obtained result. In addition latency can vary as per the distance between the nodes. The results obtained can also vary in real environment has virtual environment have their own limitations.

The future research work can be carried in two directions- to improve the latency to more significant values and security flaws in EVPN.

Acknowledgements

I would like to thank specially my mentor Vikas Sahni, who has been a guiding light during my whole research work till the submission and provided deep insights to proceed the research work in right direction. In addition, I would also thank my friends family who had supported and motivated to complete the research work.

References

- Ageyev, D., Bondarenko, O., Radivilova, T. and Alfroukh, W. (2018). Classification of existing virtualization methods used in telecommunication networks, *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, IEEE.
- Al-Lawati, A. and Al-Badi, A. H. (2016). The impact of cloud computing it departments: A case study of oman’s financial institutions, *Big Data and Smart City (ICBDSC), 2016 3rd MEC International Conference on*, IEEE, pp. 1–10.
- Anderson, C. (2015). Docker [software engineering], *IEEE Software* **32**(3): 102–c3.
- Blair, W., Olmsted, A. and Anderson, P. (2017). Docker vs. kvm: Apache spark application performance and ease of use, *Internet Technology and Secured Transactions (ICITST), 2017 12th International Conference for*, IEEE, pp. 199–201.
- Blenk, A., Basta, A., Reisslein, M. and Kellerer, W. (2015). Survey on network virtualization hypervisors for software defined networking, *arXiv preprint arXiv:1506.07275*.
- Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J. and Brandic, I. (2009). Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility, *Future Generation computer systems* **25**(6): 599–616.
- Chelladhurai, J., Chelliah, P. R. and Kumar, S. A. (2016). Securing docker containers from denial of service (dos) attacks, *Services Computing (SCC), 2016 IEEE International Conference on*, IEEE, pp. 856–859.
- Chin, W., Markatos, E. P., Antonatos, S. and Ioannidis, S. (2009). Honeylab: large-scale honeypot deployment and resource sharing, *Network and System Security, 2009. NSS’09. Third International Conference on*, IEEE, pp. 381–388.
- Dhakate, S. and Godbole, A. (2015). Distributed cloud monitoring using docker as next generation container virtualization technology, *India Conference (INDICON), 2015 Annual IEEE*, IEEE, pp. 1–5.
- Eve-NG* (n.d.).
URL: <http://eve-ng.net/>
- Felter, W., Ferreira, A., Rajamony, R. and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers, *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, IEEE, pp. 171–172.

Google Cloud including GCP & G Suite Try Free (n.d.).

URL: <https://cloud.google.com/>

Kovács, Á. (2017). Comparison of different linux containers, *Telecommunications and Signal Processing (TSP), 2017 40th International Conference on*, IEEE, pp. 47–51.

Nadgowda, S., Suneja, S. and Kanso, A. (2017). Comparing scaling methods for linux containers, *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, IEEE, pp. 266–272.

Naik, N. (2016). Building a virtual system of systems using docker swarm in multiple clouds, *Systems Engineering (ISSE), 2016 IEEE International Symposium on*, IEEE, pp. 1–3.

Pentyala, S. K. (2017). Emergency communication system with docker containers, osm and rsync, *Smart Technologies For Smart Nation (SmartTechCon), 2017 International Conference On*, IEEE, pp. 1064–1069.

portainer/portainer - Docker Hub (n.d.).

URL: <https://hub.docker.com/r/portainer/portainer>

Sever, D. and Kişasondi, T. (2018). Efficiency and security of docker based honeypot systems, *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, IEEE.

Sureshkumar, M. and Rajesh, P. (2017). Optimizing the docker container usage based on load scheduling, *Computing and Communications Technologies (ICCCT), 2017 2nd International Conference on*, IEEE, pp. 165–168.

Swenson, G. (2011). Final Version of NIST Cloud Computing Definition Published.

URL: <https://www.nist.gov/news-events/news/2011/10/final-version-nist-cloud-computing-definition-published>

Wu, J. and Yang, T.-I. (2018). Dynamic cpu allocation for docker containerized mixed-criticality real-time systems, *2018 IEEE International Conference on Applied System Invention (ICASI)*, IEEE, pp. 279–282.

Xingtao, L., Yantao, G., Wei, W., Sanyou, Z. and Jiliang, L. (2016). Network virtualization by using software-defined networking controller based docker, *Information Technology, Networking, Electronic and Automation Control Conference, IEEE*, IEEE, pp. 1112–1115.

Zhang, Y. (2018). Virtualization and Cloud Computing, *Network Function Virtualization: Concepts and Applicability in 5G Networks*, IEEE.

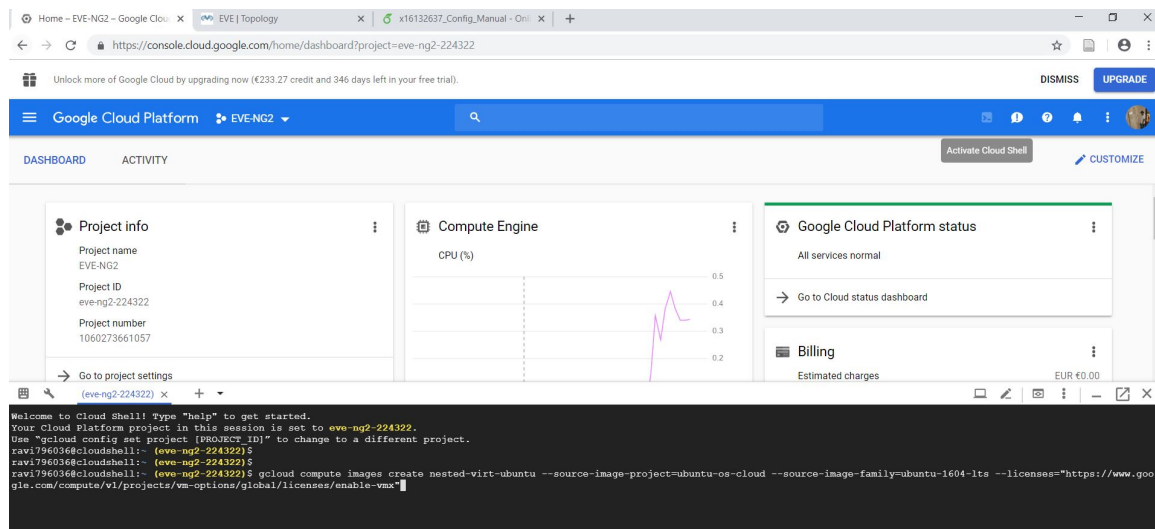
URL: <https://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=8268600>

9 Appendix - Configuration Manual

This section provides the scripts used for implementation of the research work in three segments- GCP setup, EVPN Configuration and Docker configuration.

9.1 GCP setup

The instance is created on GCP (*Google Cloud including GCP & G Suite Try Free*; n.d.) with os specification provided in section: Design specification using below script:



9.2 EVPN Configuration

EVPN had been configured on both the nexus devices one by one which are L3 devices acting as a internet gateway.

9.2.1 Nexus Switch 1

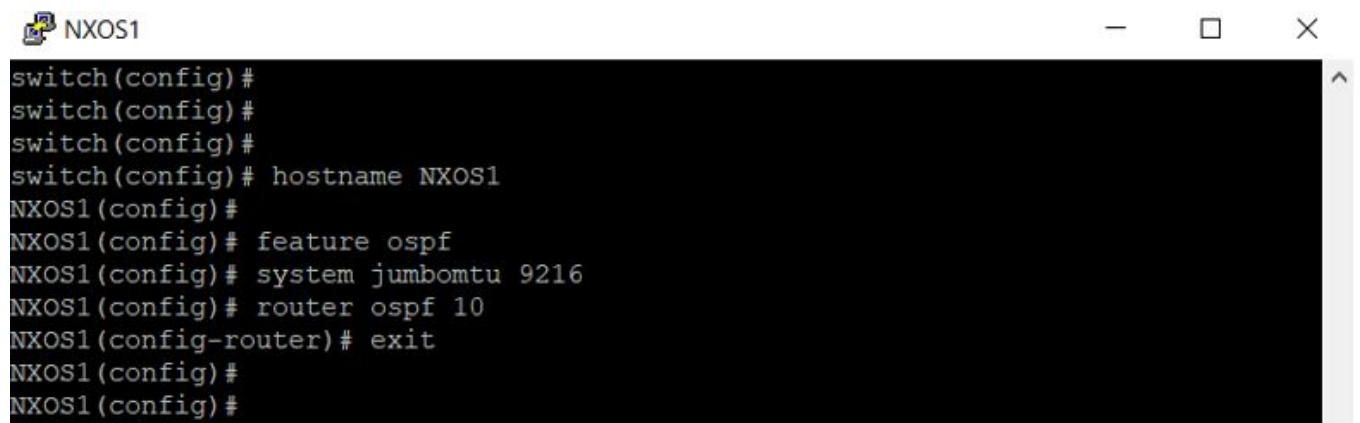


Figure 14:

In above Figure 14 snapshot, hostname of the device and ospf protocol has been configured with enabling jumbo packet.

```
NXOS1(config)#  
NXOS1(config)# interface ethernet 1/1  
NXOS1(config-if)# no switchport  
NXOS1(config-if)# ip address 10.0.1.1/30  
NXOS1(config-if)# ip router ospf 10 area 0  
NXOS1(config-if)# no shutdown  
NXOS1(config-if)# exit  
NXOS1(config)#  
NXOS1(config)#
```

Figure 15:

In Figure 15 snapshot, the ip address is configured for the device reachability.

```
NXOS1(config)#  
NXOS1(config)#  
NXOS1(config)# interface loopback 0  
NXOS1(config-if)# ip address 1.1.1.1/32  
NXOS1(config-if)# ip router ospf 10 area 0  
NXOS1(config-if)# exit  
NXOS1(config)#  
NXOS1(config)#
```

Figure 16:

In Figure 16 snapshot, loopback interface is configured and its ip is being added in the network area 0.

```
NXOS1(config)#  
NXOS1(config)# feature bgp  
NXOS1(config)# feature interface-vlan  
NXOS1(config)# feature vn-segment-vlan-based  
NXOS1(config)# feature nv overlay  
NXOS1(config)# nv overlay evpn  
NXOS1(config)#  
NXOS1(config)# fabric forwarding anycast-gateway-mac 0000.2222.3333  
NXOS1(config)#  
NXOS1(config)#
```

Figure 17:

In Figure 17 snapshot, evpn is configured.

```

NXOS1(config)#
NXOS1(config)# interface nve1
NXOS1(config-if-nve)# no shutdown
NXOS1(config-if-nve)# source-interface loopback 0
NXOS1(config-if-nve)# host-reachability protocol bgp
NXOS1(config-if-nve)#
NXOS1(config-if-nve)#
NXOS1(config-if-nve)# router bgp 65535
NXOS1(config-router)# router-id 1.1.1.1
NXOS1(config-router)# neighbor 2.2.2.2
NXOS1(config-router-neighbor)# remote-as 65535
NXOS1(config-router-neighbor)# update-source loopback 0
NXOS1(config-router-neighbor)# address-family l2vpn evpn
NXOS1(config-router-neighbor-af)# send-community
NXOS1(config-router-neighbor-af)# send-community extended

```

Figure 18:

In Figure 18 snapshot, loopback ip is being added in bgp route.

```

NXOS1(config)#
NXOS1(config)# vlan 101
NXOS1(config-vlan)# vn-segment 900001
NXOS1(config-vlan)# exit

```

Figure 19:

In Figure 19 snapshot, vlan 101 is being created and vn-segment is being assigned.

```

NXOS1(config)#
NXOS1(config)# vrf context Group-1
NXOS1(config-vrf)# vni 900001
NXOS1(config-vrf)# rd auto
NXOS1(config-vrf)# address-family ipv4
ipv4    ipv6
NXOS1(config-vrf)# address-family ipv4 unicast
NXOS1(config-vrf-af-ipv4)# route-target both auto
NXOS1(config-vrf-af-ipv4)# route-target both auto evpn
NXOS1(config-vrf-af-ipv4)# exit
NXOS1(config-vrf)# exit
NXOS1(config)#
NXOS1(config)#

```

Figure 20:

In Figure 20 snapshot, vrf is being created.

```

NXOS1(config)#
NXOS1(config)# interface vlan 101
NXOS1(config-if)# no shutdown
NXOS1(config-if)# vrf member Group-1
Warning: Deleted all L3 config on interface Vlan101
NXOS1(config-if)# ip forward
NXOS1(config-if)# exit
NXOS1(config)#
NXOS1(config)#
NXOS1(config)# interface nve 1
NXOS1(config-if-nve)# member vni 5000
NXOS1(config-if-nve-vni)# suppress-arp
Please configure TCAM region for Ingress ARP-Ether ACL before configuring ARP su
pression.

NXOS1(config-if-nve-vni)# ingress-replication protocol bgp
NXOS1(config-if-nve-vni)# member vni 900001 associate-vrf
NXOS1(config-if-nve)# exit
NXOS1(config)#

```

Figure 21:

In Figure 21 snapshot, vlan 101 is being assigned as member of vrf group-1 and vni id is assigned to the interface.

```

NXOS1(config)# router bgp 65535
NXOS1(config-router)# vrf Group-1
NXOS1(config-router-vrf)# address-family ipv4 unicast
NXOS1(config-router-vrf-af)# advertise l2vpn evpn
NXOS1(config-router-vrf-af)# exit
NXOS1(config-router-vrf)# exit
NXOS1(config-router)# exit
NXOS1(config)#
NXOS1(config)# vlan 1000
NXOS1(config-vlan)# vn-segment 5000
NXOS1(config-vlan)# exit

```

Figure 22:

In Figure 22 snapshot, vrf group is being assigned to bgp and vn-segment to vlan 1000.

```

NXOS1(config)# interface vlan 1000
NXOS1(config-if)# no shutdown
NXOS1(config-if)# vrf member Group-1
Warning: Deleted all L3 config on interface Vlan1000
NXOS1(config-if)# ip address 192.168.1.1/24
NXOS1(config-if)# fabric forwarding mode anycast-gateway -gate2018 Dec 11 23:01:
43 NXOS1 %$ VDC-1 %$ last message repeated 1 time
NXOS1(config-if)# fabric forwarding mode anycast-gateway
NXOS1(config-if)# exit
NXOS1(config)#

```

Figure 23:

In Figure 23 snapshot, layer 3 vlan is being configured.

```

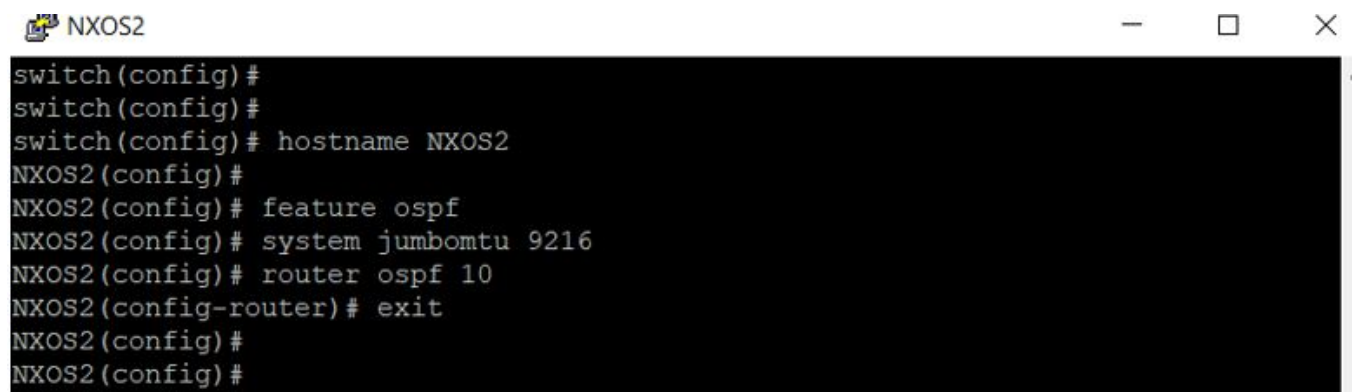
NXOS1(config)# evpn
NXOS1(config-evpn)# vni 5000 12
NXOS1(config-evpn-evi)# rd auto
NXOS1(config-evpn-evi)# route-target import auto
NXOS1(config-evpn-evi)# route-target export auto
NXOS1(config-evpn-evi)# exit
NXOS1(config-evpn)# exit
NXOS1(config)#
NXOS1(config)# interface ethernet 1/2
NXOS1(config-if)# switchport
NXOS1(config-if)# switchport access vlan 1000
NXOS1(config-if)# no shutdown
NXOS1(config-if)# exit
NXOS1(config)#

```

Figure 24:

In Figure 24 snapshot, evpn and switchport is configured.

9.2.2 Nexus Switch 2 Configuration



```

NXOS2
switch(config)#
switch(config)#
switch(config)# hostname NXOS2
NXOS2(config)#
NXOS2(config)# feature ospf
NXOS2(config)# system jumbomtu 9216
NXOS2(config)# router ospf 10
NXOS2(config-router)# exit
NXOS2(config)#
NXOS2(config)#

```

Figure 25:

In above snapshot, nexus second device hostname of the device and ospf protocol has been configured with enabling jumbo packet.

```
NXOS2(config)#  
NXOS2(config)# interface ethernet 1/1  
NXOS2(config-if)# no switchport  
NXOS2(config-if)# ip address 10.0.1.2/30  
NXOS2(config-if)# ip router ospf 10 area 0  
NXOS2(config-if)# no shutdown  
NXOS2(config-if)#  
NXOS2(config-if)#
```

Figure 26:

In Figure 26 snapshot, the ip address is configured for the device reachability.

```
NXOS2(config-if)#  
NXOS2(config-if)# interface loopback 0  
NXOS2(config-if)# ip address 2.2.2.2/32  
NXOS2(config-if)# ip router ospf 10 area 0  
NXOS2(config-if)#  
NXOS2(config-if)#  
NXOS2(config-if)# exit
```

Figure 27:

In Figure 27 snapshot, loopback interface is configured and its ip is being added in the network area 0.

```
NXOS2(config)#  
NXOS2(config)# feature bgp  
NXOS2(config)# feature interface-vlan  
NXOS2(config)# feature vn-segment-vlan-based  
NXOS2(config)# feature nv overlay  
NXOS2(config)# nv overlay evpn  
NXOS2(config)#  
NXOS2(config)#  
NXOS2(config)# fabric forwarding anycast-gateway-mac 0000.2222.3333  
NXOS2(config)#
```

Figure 28:

In Figure 28 snapshot, evpn is configured.

```

NXOS2(config)# int nve 1
NXOS2(config-if-nve)# no shutdown
NXOS2(config-if-nve)# source loopback 0
NXOS2(config-if-nve)# host-reachability protocol bgp
NXOS2(config-if-nve)#
NXOS2(config-if-nve)#
NXOS2(config-if-nve)# router bgp 65535
NXOS2(config-router)# router-id 2.2.2.2
NXOS2(config-router)# neighbor
neighbor          neighbor-down
NXOS2(config-router)# neighbor 1.1.1.1
NXOS2(config-router-neighbor)# remote-as 65535
NXOS2(config-router-neighbor)# update-source loopback 0
NXOS2(config-router-neighbor)# address-family l2vpn evpn
NXOS2(config-router-neighbor-af)# send-community
NXOS2(config-router-neighbor-af)# send-community extended
NXOS2(config-router-neighbor-af)#
NXOS2(config-router-neighbor-af)#

```

Figure 29:

In Figure 29 snapshot, loopback ip is being added in bgp route.

```

NXOS2(config)#
NXOS2(config)# vlan 101
NXOS2(config-vlan)# vn-segment 900001
NXOS2(config-vlan)# vrf context Group-1
Warning: Enable double-wide arp-ether tcam carving if igmp snooping is enabled.
Ignore if tcam carving is already configured.
NXOS2(config-vrf)# vni 900001
NXOS2(config-vrf)# rd auto
NXOS2(config-vrf)# address-family ipv4 unicast
NXOS2(config-vrf-af-ipv4)# route-target both auto
NXOS2(config-vrf-af-ipv4)# route-target both auto evpn
NXOS2(config-vrf-af-ipv4)# exit
NXOS2(config-vrf)# exit
NXOS2(config)#

```

Figure 30:

In Figure 30 snapshot, vlan 101 is being created and vn-segment is being assigned.

```

NXOS2(config)#
NXOS2(config)# interface vlan 101
NXOS2(config-if)# no shutdown
NXOS2(config-if)# vrf member Group-1
Warning: Deleted all L3 config on interface Vlan101
NXOS2(config-if)# ip forward
NXOS2(config-if)# exit
NXOS2(config)#
NXOS2(config)# interface nve 1
NXOS2(config-if-nve)# member vni 5000
NXOS2(config-if-nve-vni)# suppress-arp
Please configure TCAM region for Ingress ARP-Ether ACL before configuring ARP su
pression.

NXOS2(config-if-nve-vni)# ingress-replication protocol bgp
NXOS2(config-if-nve-vni)# member vni 900001 associate-vrf
NXOS2(config-if-nve)# exit
NXOS2(config)#

```

Figure 31:

In Figure 31 snapshot, vrf is being assigned to vlan and nve interface is configured.

```

NXOS2(config)#
NXOS2(config)# router bgp 65535
NXOS2(config-router)# vrf Group-1
NXOS2(config-router-vrf)# address-family ipv4 unicast
NXOS2(config-router-vrf-af)# advertise l2vpn evpn
NXOS2(config-router-vrf-af)# exit
NXOS2(config-router-vrf)# exit
NXOS2(config-router)# exit
NXOS2(config)#
NXOS2(config)# vlan 1000
NXOS2(config-vlan)# vn-segment 5000
NXOS2(config-vlan)# exit

```

Figure 32:

In Figure 32 snapshot, vrf group is being assigned to bgp and vn-segment to vlan 1000.

```

NXOS2(config)# interface vlan 1000
NXOS2(config-if)# no shutdown
NXOS2(config-if)# vrf member Group-1
Warning: Deleted all L3 config on interface Vlan1000
NXOS2(config-if)# ip address 192.168.1.1/24
2018 Dec 11 23:22:21 NXOS2 %$ VDC-1 %
$ last message repeated 1 time

NXOS2(config-if)# ip address 192.168.1.1/24
NXOS2(config-if)# fabric forwarding mode anycast-gateway
NXOS2(config-if)# exit
NXOS2(config)#
NXOS2(config)# evpn
NXOS2(config-evpn)# vni 5000 12
NXOS2(config-evpn-evi)# rd auto
NXOS2(config-evpn-evi)# route-target import auto
NXOS2(config-evpn-evi)# route-target export auto
NXOS2(config-evpn-evi)# exit
NXOS2(config-evpn)# exit
NXOS2(config)#

```

Figure 33:

In Figure 33 snapshot, layer 3 vlan and evpn is being configured.

```

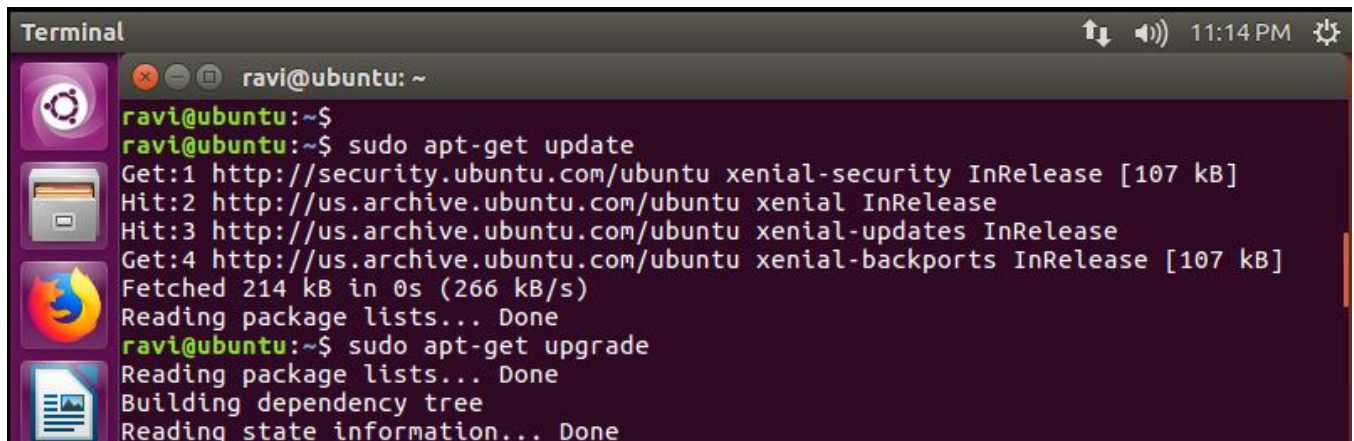
NXOS2(config)#
NXOS2(config)# interface ethernet 1/2
NXOS2(config-if)# switchport access vlan 1000
NXOS2(config-if)# no shutdown
NXOS2(config-if)# exit
NXOS2(config)#
NXOS2(config)#

```

Figure 34:

In Figure 34 snapshot, switchport is configured.

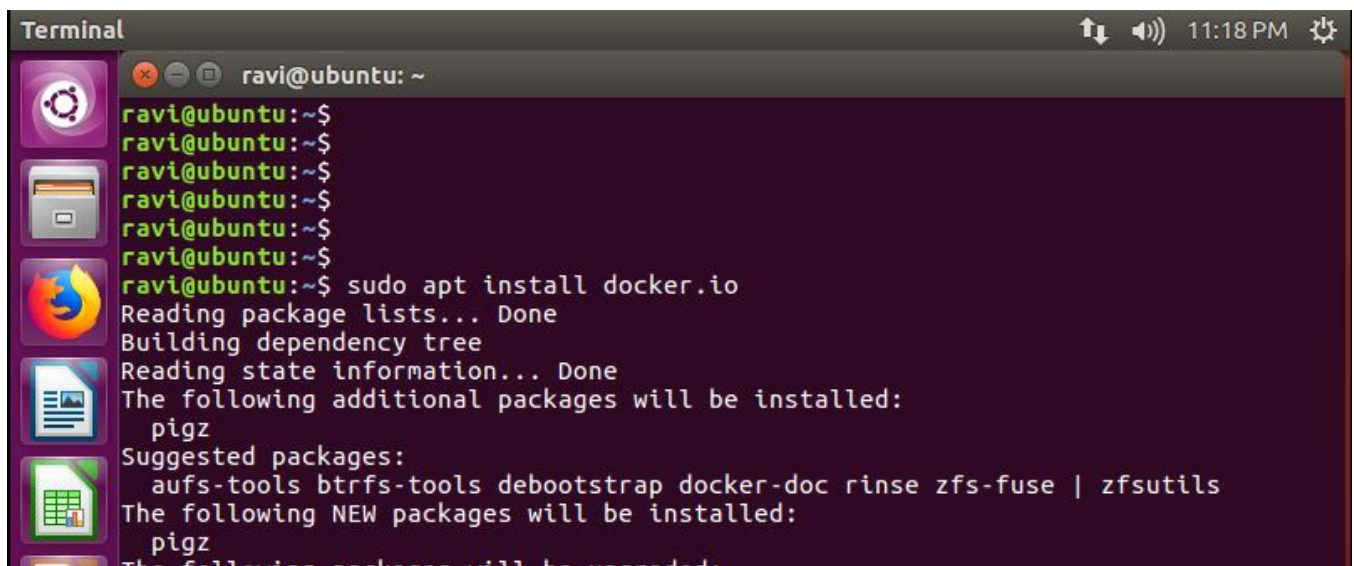
9.2.3 Docker Configuration



```
Terminal
ravi@ubuntu: ~
ravi@ubuntu:~$
ravi@ubuntu:~$ sudo apt-get update
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease [107 kB]
Hit:2 http://us.archive.ubuntu.com/ubuntu xenial InRelease
Hit:3 http://us.archive.ubuntu.com/ubuntu xenial-updates InRelease
Get:4 http://us.archive.ubuntu.com/ubuntu xenial-backports InRelease [107 kB]
Fetched 214 kB in 0s (266 kB/s)
Reading package lists... Done
ravi@ubuntu:~$ sudo apt-get upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figure 35:

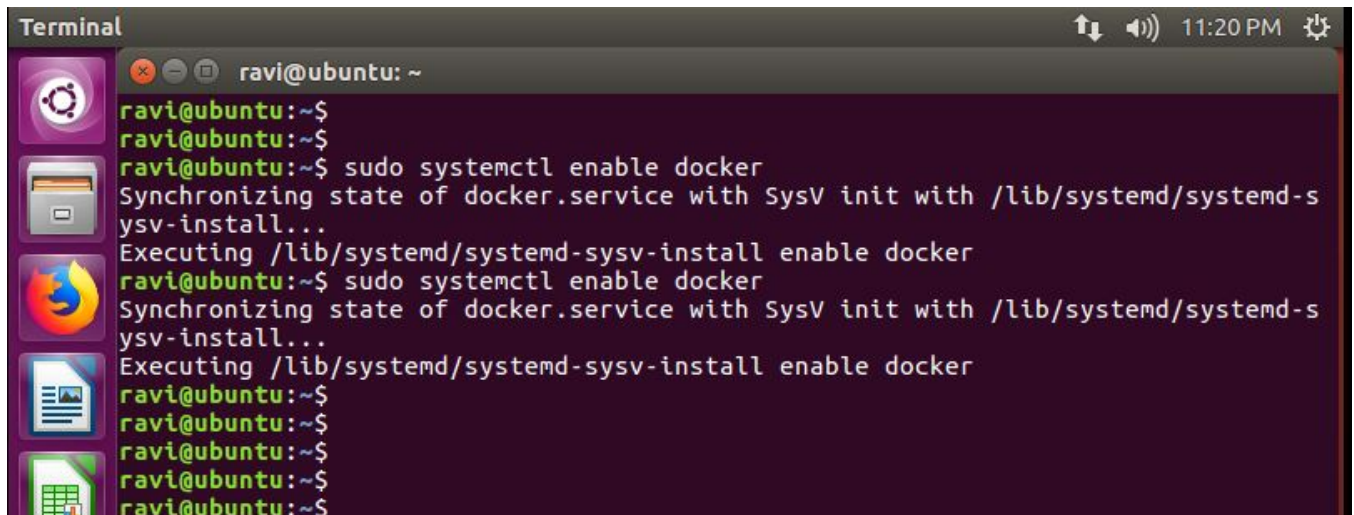
In above Figure 35, system is updated.



```
Terminal
ravi@ubuntu: ~
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$ sudo apt install docker.io
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  pigz
Suggested packages:
  aufs-tools btrfs-tools debootstrap docker-doc rinse zfs-fuse | zfsutils
The following NEW packages will be installed:
  pigz
The following packages will be upgraded:
```

Figure 36:

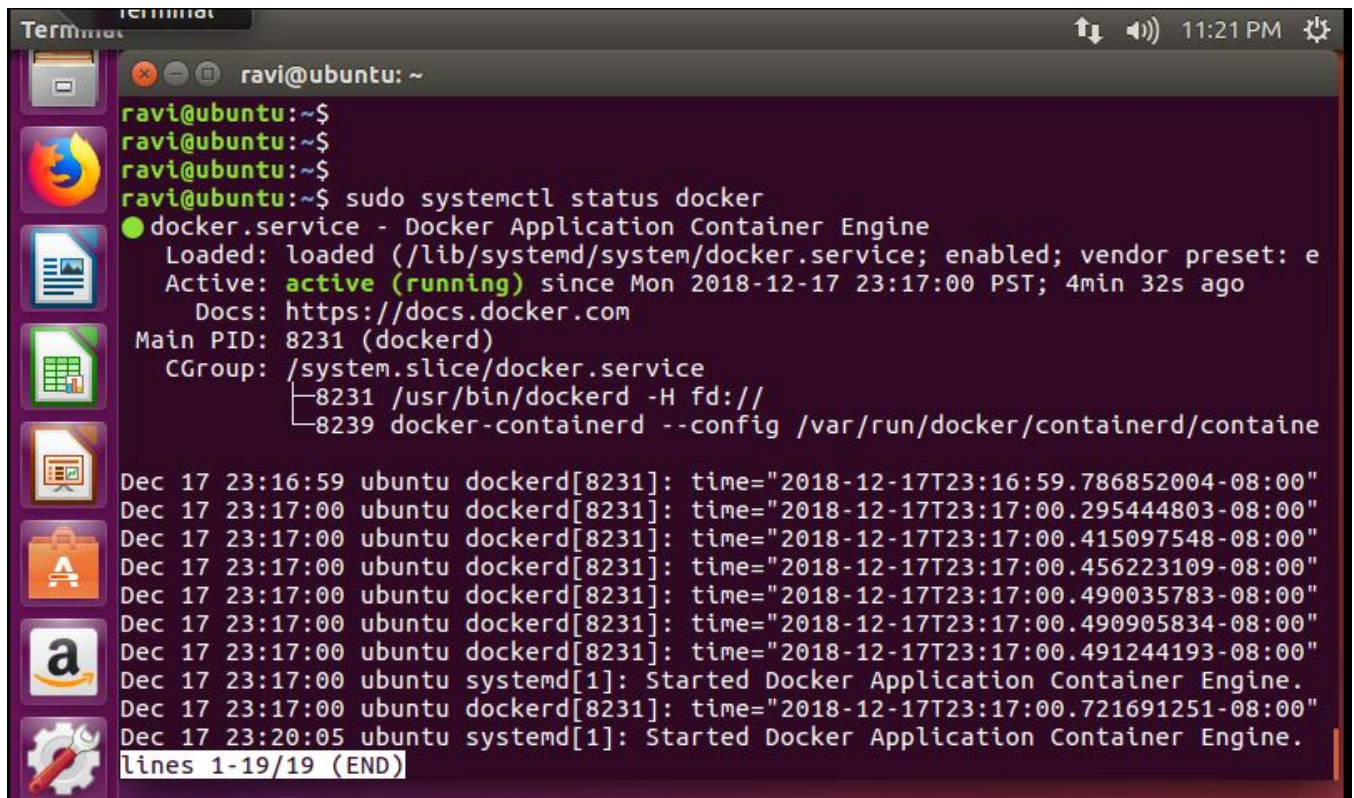
In Figure 36, Docker is installed.

A terminal window titled "Terminal" with a dark background. The user "ravi@ubuntu" is at the prompt. The command "sudo systemctl enable docker" is entered and executed. The output shows the system synchronizing the state of the docker.service with SysV init and then executing the installation script. The command is repeated, and the prompt returns. The left sidebar shows icons for system settings, file manager, Firefox, LibreOffice, and a spreadsheet.

```
Terminal
ravi@ubuntu: ~
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$ sudo systemctl enable docker
Synchronizing state of docker.service with SysV init with /lib/systemd/systemd-sysv-install...
Executing /lib/systemd/systemd-sysv-install enable docker
ravi@ubuntu:~$ sudo systemctl enable docker
Synchronizing state of docker.service with SysV init with /lib/systemd/systemd-sysv-install...
Executing /lib/systemd/systemd-sysv-install enable docker
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
```

Figure 37:

In Figure 37, Docker is enabled and its services has been started.

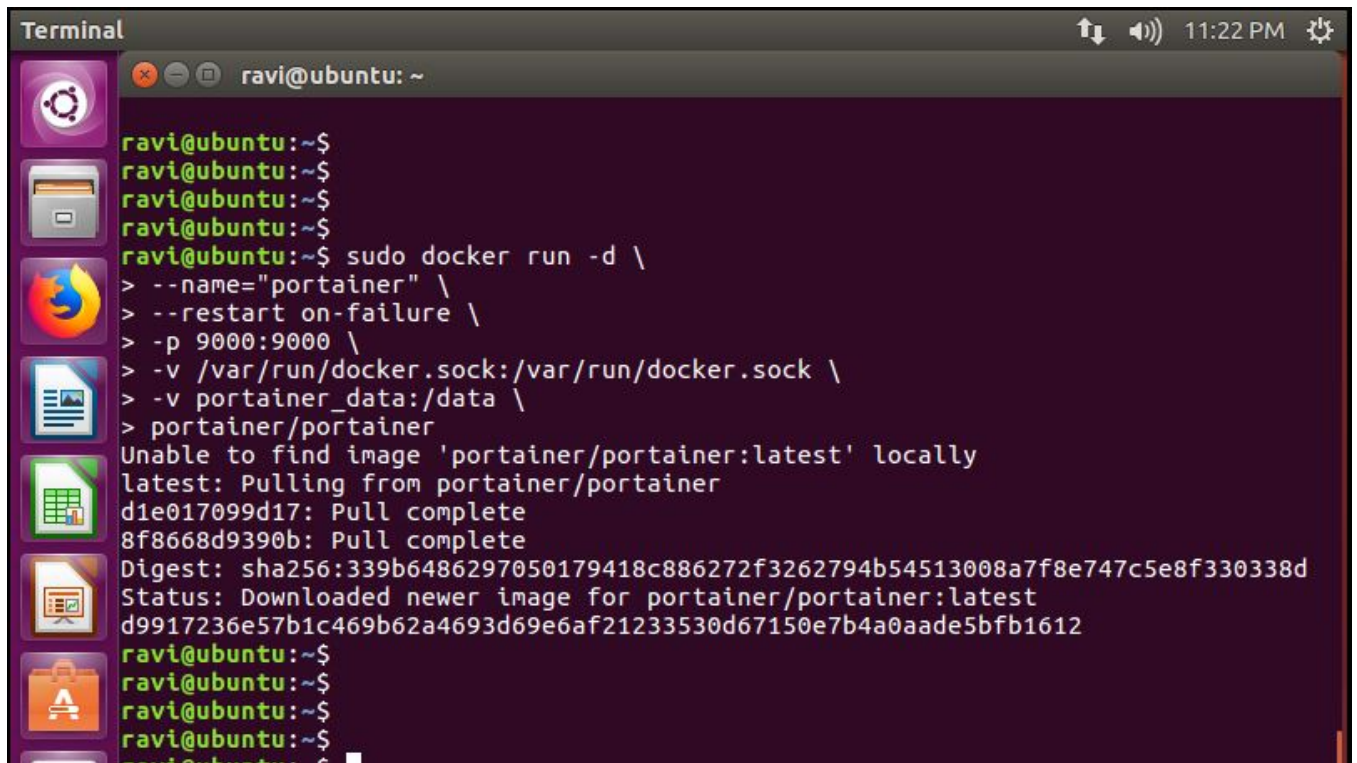
A terminal window titled "Terminal" showing the output of "sudo systemctl status docker". The output indicates that the docker.service is loaded, enabled, and active (running). It shows the main PID as 8231 (dockerd) and lists the CGroup processes. Below this, a series of log entries from the journal are displayed, showing the Docker Application Container Engine starting at 23:17:00. The left sidebar is the same as in Figure 37.

```
Terminal
ravi@ubuntu: ~
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: e
   Active: active (running) since Mon 2018-12-17 23:17:00 PST; 4min 32s ago
     Docs: https://docs.docker.com
    Main PID: 8231 (dockerd)
      CGroup: /system.slice/docker.service
              └─8231 /usr/bin/dockerd -H fd://
                 └─8239 docker-containerd --config /var/run/docker/containerd/containe

Dec 17 23:16:59 ubuntu dockerd[8231]: time="2018-12-17T23:16:59.786852004-08:00"
Dec 17 23:17:00 ubuntu dockerd[8231]: time="2018-12-17T23:17:00.295444803-08:00"
Dec 17 23:17:00 ubuntu dockerd[8231]: time="2018-12-17T23:17:00.415097548-08:00"
Dec 17 23:17:00 ubuntu dockerd[8231]: time="2018-12-17T23:17:00.456223109-08:00"
Dec 17 23:17:00 ubuntu dockerd[8231]: time="2018-12-17T23:17:00.490035783-08:00"
Dec 17 23:17:00 ubuntu dockerd[8231]: time="2018-12-17T23:17:00.490905834-08:00"
Dec 17 23:17:00 ubuntu dockerd[8231]: time="2018-12-17T23:17:00.491244193-08:00"
Dec 17 23:17:00 ubuntu systemd[1]: Started Docker Application Container Engine.
Dec 17 23:17:00 ubuntu dockerd[8231]: time="2018-12-17T23:17:00.721691251-08:00"
Dec 17 23:20:05 ubuntu systemd[1]: Started Docker Application Container Engine.
lines 1-19/19 (END)
```

Figure 38:

In Figure 38, Docker services has been checked as active.



```
Terminal
ravi@ubuntu: ~

ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$ sudo docker run -d \
> --name="portainer" \
> --restart on-failure \
> -p 9000:9000 \
> -v /var/run/docker.sock:/var/run/docker.sock \
> -v portainer_data:/data \
> portainer/portainer
Unable to find image 'portainer/portainer:latest' locally
latest: Pulling from portainer/portainer
die017099d17: Pull complete
8f8668d9390b: Pull complete
Digest: sha256:339b6486297050179418c886272f3262794b54513008a7f8e747c5e8f330338d
Status: Downloaded newer image for portainer/portainer:latest
d9917236e57b1c469b62a4693d69e6af21233530d67150e7b4a0aade5bfb1612
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
ravi@ubuntu:~$
```

Figure 39:

In Figure 39, (*portainer/portainer* - *Docker Hub*; n.d.) Portainer Docker image is being pulled from Docker hub and 9000 port has been assigned for communication.