Network Virtualization: Proof of Concept for Remote Management of Multi-Tenant Infrastructure

Stephen Ugwuanyi
Electrical and Electronic Engineering
University of Strathclyde
Glasgow, United Kingdom
stephen.ugwuanyi@strath.ac.uk

Rameez Asif
Power Networks Demonstration Centre
University of Strathclyde
Glasgow, United Kingdom
rameez.asif@strath.ac.uk

James Irvine
Electrical and Electronic Engineering
University of Strathclyde
Glasgow, United Kingdom
j.m.irvine@strath.ac.uk

***Abstract*—Recently, container and cloud-based orchestration technologies are important requirements for data and functional integration of new control and automation services in distributed networks. The industrial adoption of Docker Swarm and Kubernetes based orchestrators are due to their abilities to support large-scale deployment, reduce capital cost, enable interoperability, introduce flexibility, minimize vendor lock-in and identify scalable means of deploying new functions with minimal performance and monitoring related issues. In this study, we performed a comprehensive study of the requirements and strategies in a Containerized Docker Swarm Orchestration Framework (CDSOF) as an efficient, cost-effective, secure and resilience device management tool for remote administration, configuration and supervision of virtualized large-scale Ethernet connected multi-tenanted assets. In a performance and security comparative analysis to Kubernetes and Apache Mesos, Docker Swarm is the leading solution with support features for overlay networks such as Flannel and Weave. Finally, we established the design and test specifications to guide our substation Virtual Remote Terminal Units (vRTUs) implementation.**

Keywords-IP addressing, internet of things, multi-tenancy, networks, overlay network, security, utility, virtualization

I. Introduction

The establishment of the requirements and strategies for managing, configuring, and supervising large-scale containerized devices remotely is a challenging task in recent years. With the increasing demand for Docker container technology in managing industrial domain applications and services and the evolution of emerging frameworks due to technological advancement, there is a need for studies of the performance of Docker orchestration frameworks. Given that the existing container orchestration solutions have not addressed networking issues such as security [1], IP addresses [2], [3], portability [4], and other performance characteristics in multi-tenant deployments [5], [6], [7], [8], we assessed Docker Swarm strengths, opportunities and weakness. It currently leads to two key roles; Application Manager and Cluster Administrator for developing and managing applications that run in Docker clusters and for customizing the orchestrator framework to fit projects-specific requirements respectively.

Investigating the effectiveness of a virtualization approach in meeting these objectives, this paper outlines the benefits of using Docker container technologies to achieve optimum performance in multi-tenant solutions of distributed computing functions and management commonly related to resource administration, software provisioning, pricing, reconfiguration, supervision and maintenance of Container Orchestration Frameworks (COF). The ever-increasing number of monitored and controlled utility network assets that are multi-tenant in nature will require a stable Docker and overlay and networking tools to handle functions such as security, customizability, IP addressing, and filtering for internal and external communications between containers, hosts and the outside world. The views of current industrial and other domain practices are also identified and the summary of the contributions of this work is as follows:

- Provide a comprehensive review of the requirements and strategies of the Container Docker Swarm Orchestration Frameworks (CDSOF).
- Established design and test specifications to guide the realization of distributed control and automation functions and remote management and configuration of vRTUs.

This paper is divided into eight sections. Section II critically discusses Docker related multi-tenant resource management and control orchestration frameworks. Section III presents Docker virtualization technologies and its networking concepts. Section IV analyzes the load balancing and service discovery tools for Docker orchestration frameworks. The multi-tenant management requirements like the overlay and security were discussed in Section V. Section VI is a critical analysis of the IP addressing scheme as a management technique. Section VII is the critical discussion surrounding the choices for the test phase of the project, and section VIII concludes the paper with key concepts established for future work.

II. Related Literature

Cloud and edge-based infrastructures are leveraging virtualization systems for multi-tenant Software as a Service (SaaS) applications. Resource optimization, cost-efficiency and energy utilization are a few of the reasons that give rise to the use of container orchestration solutions for resource management and administration in Distributed Systems (DS). Distributed multi-tenant systems allow multiple entities to share the same resource contrary to assigning one resource to one entity in a single-tenant system [9]. Multi-tenancy is a cloud computing concept that allows isolated virtual components in the cloud or the host Operating System (OS) Kernel to share resources in the form of (SaaS) on top of another service; Platform as a Service (PaaS). Multi-tenancy approach to a large scale device management means that multiple separate entities (tenants) on one cloud platform will manage shared resources effectively, i.e. sharing more than one application resource or services and operating within different instances [5] based on virtualization and network slicing techniques [9]. Tenants are regarded as a group of roles, users or devices within an organization infrastructure [10]. The resources shared are not enough in all cases and requires orchestration management tools to meet the requirements of the Service Level Agreement (SLA) [11]. To distribute container service instances, a load balancing system is required and service discovery mechanism for efficient communication between containers.

In a multi-tenancy, the failure of one service does not affect other services [9], since each service runs on the separate Linux container, promoting fault tolerance and resilience. But, performing management functions like tenant migration as observed in a multi-tenant business process management system [6], can interrupt both co-located and the migrated tenant services, and also lower the network performance. The migration duration increased with the size of the data and there is no possibility of achieving a zero-downtime in the process. As the workload increases, maintaining the right degree of isolation and performance becomes very difficult to achieve as well as selecting the right multi-tenanted patterns for a multi-tenant content management system. Cgroups, chroot, and Kernel namespaces were identified in [12] as the basic container resource isolation mechanisms. The performance of shared, isolated, and dedicated multi-tenant deployment patterns have been investigated [13]. The dedicated deployment pattern has far less unresolved request errors as the number of tenants increased; achieves a high degree of isolation, but

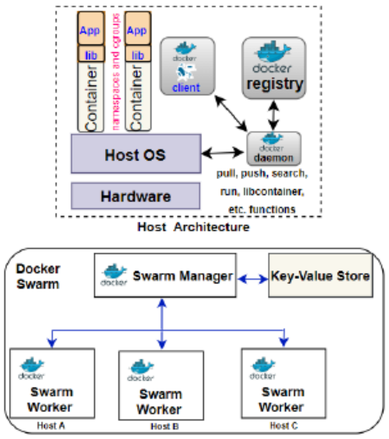


Figure 1. Docker Host and Swarm Architecture

at a high running cost and resources since more hosts will be required. A near-optimal approach has been recommended for providing multi-tenancy isolation based on workload dynamics [8].

For auto-deployment, scaling, and management of a multi-tenant distributed systems, Docker Swarm, Google Kubernetes, and Apache Mesosphere is the commonly used container orchestration frameworks at the time of writing this paper [14], [15], [3]. However, Docker Swarm has gained more industrial adoption for ease of use and implementation for bootstrapping container images [14] with enough resources to quickly develop and manage [7]. Docker has the most common development container image in the technology space today [16] and has provisioned multiple Docker Swarm on the Amazon Web Service (AWS) cloud platform [17]. With Docker orchestrator, automated infrastructure provisioning, upgrade to a new service without interruption, compatibility with overlay storage solutions, auto-scaling of nodes and groups, centralized logging and automatic removal of unused Docker resources are all achievable. In [18], Docker containers automated the control of distributed, cloud-based platforms using structured RESP API. Access Control List (ACL) and hierarchical locks were identified in [5] as the features to support access privileges and conflicts respectively in multi-tenancy solutions. The remaining part of this paper explores the Docker orchestration framework which allows more resource sharing while maintaining high performance, strong security, and low interference level.

III. Docker Container Networking Overview

A multi-tenant management system depends on the virtualization technology employed. Docker is a light-weight virtualized high-level container technology that eliminates features of guest OS in hypervisors, allowing applications to share packages since they are provided by the same host [18]. It is flexible and resource-efficient in integrating the guest application into the OS Kernel, resulting in smaller overheads [19]. Linux-Server, Linux Containers (LXC), Docker, Rocket, OpenVZ are container-based solutions as described in [20] with namespaces and Control Group (cgroup) subsystems as the building blocks [21]. Docker container specifications include images and runtime, Dockerfiles, Docker daemon, Ducker Hub, etc. as shown in Fig. 1. The Dockerfile is an OS package instruction to Docker daemon for building an image [22]. The runtime executes container images within an isolated environment. Docker image is a read-only template that packages applications into a unit file for distribution to containers [23]. The other benefit of using container includes reducing infrastructural costs, increasing the boot process, lowering resource overhead, improving resource utilization, quickening the development cycle [19], [24], and allowing seamless operational and management services such as a software upgrade in the lifecycle management [25]. The impact of such management actions on Docker network is always difficult to measure. On the power utilization of virtualized edge devices, the impact can rise to 13% and should be considered especially when the edge hardware are battery-powered [26].

The Container Network Model (CNM) and Container Network Interface (CNI) are the container networking technologies in use today for container configuration and management purposes. CNM is a networking tool for Docker Swarm while the CNI is for Kubernetes and Apache Mesos [27]. There are different modes of networking Docker containers. The host is the basic default Docker containers networking mode, which is not scalable without making changes to the network model and as such lack isolation and security functions. In host mode, the network performance is classified as bare mental [28] since containers are restricted within a host namespace and no Network Address Translation (NAT) needed.

The bridge mode is the native networking tool for single-host Docker containers that localize network namespace and IP address for host-based communication. The bridge is created by the Docker daemon and used to assign private virtual subnet IP address and port to host containers as soon as they are created. This establishes the first network foundation on which NAT or overlay network services can be integrated if external communication is desirable. Bridge setup IPv4 and IPv6 with the IP tables, handle IP and port forwarding, networking filtering, and built-in Domain Name System (DNS) for the resolution of container names. In bridge mode, Docker daemon connects host containers to the internal network interface. It has issues with dynamic IP addressing and port mapping, and can render network management a challenging task as the container public IP addresses are bound to host IP addresses [27].

To allow hosts access to external communication, the container uses an internal IP address and port mapping feature to prevent address conflict, see Fig. 2. Using Docker Command Line Interface (CLI), a port can be actioned to listen to a particular interface based on the established rules. External traffic uses the host IP address and the exposed port and never the container IP addresses and internal ports. The Virtual Extensible Local Area Network (VXLAN) is the default Docker overlay mode for multi-tenant scenarios [28] where IP addresses, portability and isolation are difficult tasks across hosts. With overlay networks, these issues are resolved in the host and bridge modes but introduce overhead associated with packets encapsulation and decapsulation.

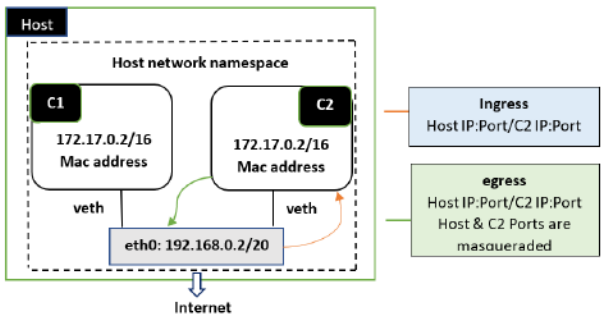


Figure 2. External Access for Containers

IV. Docker Container Orchestration Tools for Multi-tenant deployment

Docker containerization has made the large-scale deployment of microservices implementation straightforward by allowing the use of one Docker images published in Docker hub once to instantiate into many containers with the help of Docker Compose. Orchestration tools like Docker Swarm and Kubernetes (K8s) have been used to manage container dependencies in different modes for a production type of service where performance and reliability are critical requirements. Creating and killing containers, grouping them into cluster nodes and coordinating other application functions are tasks of an orchestrator. But as the complexity of the container network grows, several factors should be considered in choosing an orchestrator framework for a multi-tenant network environment. They include how containers communicate within a host, between different hosts, with the outside world, and how to keep track of what they expose outside the network for security reasons. Others are IP assignment and management, load balancing, security policies, storage management, scaling, etc. A Docker container deployment decision is usually based on the performance, scalability, load balancing, service discovery, application portability, security, resilience, distributed nature, and processing power of the application [16], recognizing that use cases have different orchestration requirements. Large-scale centralized deployment of container would result in a cluster of virtual container network known as Docker Swarm. There are several different multi-tenant networking options for containers in Docker Swarm, Kubernetes, and Mesosphere. However, the Docker Swarm orchestrator platform is preferable because it is free and platform-independent [23], and the management of cluster Docker engines is easier and predictable [26]. It has other advantages of providing cluster self-healing, easy scaling, and high availability of microservices.

A. Load Balancing and Service Discovery

Load balancing in Docker Swarm for SaaS application allows manager nodes to equally distribute scaled service request within-cluster nodes horizontally. Docker Swarm has built-in and external load balancing capabilities [29] for scaling the performance and reliability of service. Services created in Docker cluster are automatically instantiated with a virtual IP address for routing and rerouting traffic. To achieve an effective load balancing, the network capacity of each node when detected are used to assign resources to containers. It is done at the server level with the load balancer solutions like Bamboo, Envoy, NGINX, and HAProxy [30]. Load balancing can also be done at the edge by employing techniques such as graph model where the graph vertex and edge indicate nodes and data dependencies [9]. ZooKeeper, Consul, etcd and Radis are the common microservices provisioning tools in the Docker orchestration implementation for registering and promoting host and its service features [26], see Fig. 3.

1) Zookeeper: Is an open-source Apache Foundation centralized service for configuration and management of a

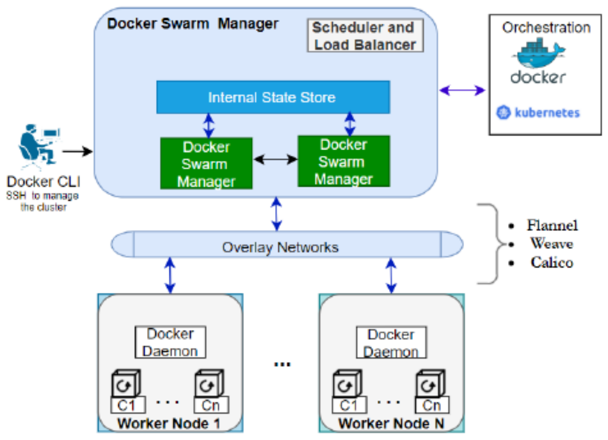


Figure 3. Docker Swarm Orchestration Framework

few clusters of znodes for best performance distribution functions. It uses the leader-follower server setting to allow tenants access to distributed server services and resolve conflict and synchronization problems in distribution systems through the update and commit policies [31]. It offers simple and stable ways of using a consensus algorithm to coordinate and manage distributed resources and tasks without losing any relevant information [32]. Multiple tenants can access ZooKeeper's services concurrently, synchronously or asynchronously [33].

2) Consul: Is an open-source Hashicorp distributed Key Value (KV) store that uses Raft Consensus Algorithm (RCA) to manage and maintain distributed service. Like ZooKeeper, its server runs in follower and leader modes. Consul supports other features such as service discovery, multi-data centres and transport layer security certifications [30].

3) Etcd: Is also an open-source Cloud Native Computing Foundation (CNCF) distributed, reliable KV store for a cluster of machines that uses RCA to consistently register, watch and lookup container instances dynamically [34]. The node clusters are elected based on RCA, giving the network tolerance to accommodate any node failure. Etcd master node based on the TCP protocol supports a port range of between 2379 to 2380 [35]. Consul and etcd are lightweight distributed management tools similar to ZooKeeper in performance. Apache Curator or other higher interfaces should also be considered for large-scale deployment.

V. Containerized Multi-Tenant Management Requirements

An orchestrator is a tool for automating the coordination and management of containerized services and tasks in different nodes. Docker Swarm, Kubernetes, and Apache Mesos are among the common orchestration tools for varying large-scale cluster management. Large-scale containerized multi-tenant deployment would mean scaling out the infrastructure horizontally to accommodate more hosts due to increasing workload and the need for resilience [30]. It is important to note that the requirements for multi-tenant container networking differ in every use case and the network performance and security should be placed at the top of the design and deployment decisions. In a multi-tenant network setting like in utility, the network performance evaluation requires the examinations of bridges and overlay networks, workload distribution, interferences, CPU overhead, and scalability of the network as briefly described in this section of the paper.

A. Overlay Networks

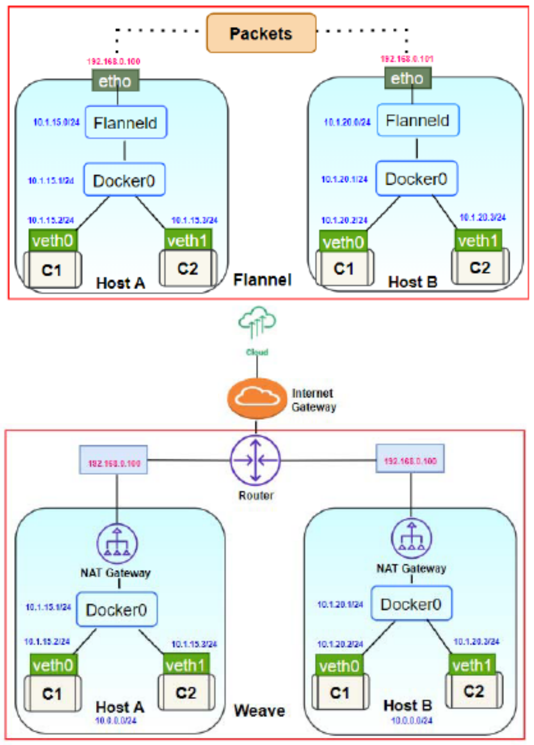
Overlay networks such as Flannel, Weave Net, and Calico are used individually with the native Docker networking features for single-host Docker containers to create Docker Swarm. Weave and Flannel are the current overlay solution for monitoring, networking, and management of Docker containers and microservices. Their TCP and UDP throughput performance in AWS and Azure for different virtual CPUs and RAM spaces in [21] shows Flannel as an optimal choice for high-speed networking in either TCP or UDP. For High-Performance computing in K8s cluster [35], Flannel in UDP type can support a port range of 8472 master and worker nodes.

Table 1. Overlay Network Comparison

Overlay	Bridge	KV store	CPU Service Demand	Lunch Time	Security
Docker	VXLAN bridge br0	yes	Lowest	Slow	no
Weave	Weave bridge	No	High	Fastest	Encryption
Flannel	Docker bridge	Yes	Low	Fast	Encryption

1) Flannel: Is a mature, dynamic container orchestration solution for multi-host and multi-container networking by CoreOS. For communication over a host with Docker, containers rely on port mapping to reach other containers on different hosts since container IP addresses are tied to the host IP address [36]. Flannel allows containers on different hosts to be assigned unique routable IP addresses by restricting containers within hosts to different subnets. Through packet encapsulation, virtual overlay network that spans the entire Docker Swarm is created. Host containers communicate using the native Docker bridge based on the internal addressing while communication between hosts is encapsulated in UDP packets mostly with VXLAN. Using the distributed etcd key-value cluster store as a control plane, Flannel uses the API information to store overlay and host state information (e.g., IP addresses) for accurate packet routing and synchronization of services between containers [28].

2) Weave: Is a networking overlay that provides a virtual connection for containers in multiple hosts. Services



are discovered without the need for port mapping [30].

Figure 4. Flannel and Weave Network Architectures

Weave assigns containers unique IP addresses across the Docker Swarm even in multiple subnets and keeps the networking data in memory and consistent between containers to avoid a single point of failure. In Fig. 4, for the container in C1 on host A to communicate with containers in host B, the containers can find the IP address of each other without the need for NAT by hosting micro DNS server at each node. The Weave router handles both the external and internal UDP containers packets transmissions over the hosts and the internet. The weave is more suitable for Kubernetes and relies on the host-based routers to connect pods [37]. Weave can horizontally extend and complement other orchestration platforms, but its impact on network performance is an area that needs further investigation and depends on the Weave datapath implementation and virtual machine.

B. *Network Start-up Time*

The time taken to establish a connection with a container in a Docker network is a critical requirement for consideration in the networking type selection process. Measured as the average time interval in which a container is created and to respond to a ping request in a running network. In [27], the Weave network lunchtime was 1,605.1 ms faster than the Flannel network while a host took 497.4 ms. In a multi-tenant environment, the encapsulation mechanisms introduced by the overlay network further increase the network latency. Weave utilizing KV store for all address mappings presents a single point of failure and delays container lunchtime. Measures like virtualizing edge devices in a multi-tenant environment will help reduce latency which is higher when compared to multi-tenant cloud-based virtualization without edge computing. To migrate some of the computations towards the nodes, lightweight restful microservices virtual resources like blockchain should be hosted on the edge device. Permission-based Blockchain as a service such as Bluemix can push code into the edge computing device [5].

C. *Scalability*

Docker bridge overlay increases the latency in the multi-tenant networks as the number of container increases. Latency tends to be lower on a single host and scale in 10 folds in multi-tenant communication [27]. To avoid the cost of overlay networks as connectivity increases, we recommended using the IPv6 addressing scheme to increase containers in a host rather than increasing the number of hosts.

D. *Interference and Isolation*

The userspace daemon networking and in-Kernel features are critical parts of container networking for interference and isolation. In-Kernel processes have lower interference as contending containers have higher priority than the third-party overlays. For example, the in-kernel virtual routing in Weave isolates container network services, hence, contributing to a higher throughput under certain interference than in Flannel. To increase isolation between containers, we recommend wrapping containers within a host similar to what was achieved in [28] with the help of hyper-containers at the expense of higher overhead.

E. *CPU Overhead*

The functional services in container networking require CPU resources. It increases with overlay networks as functions like encapsulation consume more CPU resources. Kernel technologies such as the namespaces and cgroups reduce the overhead in Docker containers [38]. Hadoop YARN is a distributed OS for large-scale big data cluster resource management [27]. Based on parallel computing technology, a single task can be performed faster using multiple containers. In [28], the API selective Remote Direct Memory Access (RDMA) showed effectiveness in increasing the container network performance by reducing the CPU roles in data transmission and reception with container isolation as a trade-off. This restricts its use to a single tenant.

F. *Security*

There are ranges of security issues in the Docker ecosystem for both single and multi-host environment. Namespaces and cgroups are Kernel features that single host security relies on. But because containers run on the same OS, its attack surface becomes relatively high by default, though can be reduced through node abstractions introduced when hypervisors are used. This presents a tradeoff between performance and security which depends on resource sharing (workload) and isolation in container networks respectively [27]. In multi-tenancy, the compromise lies between a network with significant overhead due to overlay networking resource with increases flexibility and security in network management or a better performing network with high vulnerabilities.

From existing vulnerability studies, both the official and unofficial Docker images are not secure. In [19], through Docker Image vulnerability analysis and penetration testing, Docker images contained significant vulnerabilities, some of which are inherited from the parent image due to an inadequate update of Docker images. Docker Swarm in standalone mode and Mesos have limited security. However, the Docker Swarm integrated mode is as robust as Kubernetes in terms of containers and nodes cluster security in any orchestration frameworks. Worker nodes can be authenticated with master API, control and application messages are encrypted, external access to service ports are restricted, and tokens for bootstrapping worker nodes are automated. The futuristic security considerations in Docker Swarm for securing containers as identified in [15] will allow functions such as setting Linux capabilities, and Secure Computer Mode (seccomp) profiles per containers.

VI. Containerized Multi-Tenant Management Techniques

Docker Swarm, Kubernetes, and Mesos are the container orchestration solution with scheduling, health checking, autoscaling, upgrade, and service discovery functions. Kubernetes and Mesos are based on CNI networking plug-in while the Docker Swarm uses libnetwork. They all support Docker containers networking functions. Common in all container orchestration frameworks is the master-worker architecture where the master node stores container configurations states, schedules workloads, controls, and monitors worker node run-time states. Replicating master nodes in Docker Swarm ensures higher cluster availability and maintaining an odd number as the limit lies between performance and more fault-tolerant trade-offs even as the partitions increases. RCA is an effective tool for managing master nodes (quorum) within a swarm cluster with management task failure rate of (N-1)/2 and decision rate of (N/2)+2 [39]. Paxos is another consensus algorithm for managing replicated logs but not readily used due to its difficulty in building practical systems [40]. Master nodes use a static IP address for network stability while worker node IP addresses can be assigned dynamically [22]. Master nodes can be installed on-premise, in private infrastructure or the cloud (proprietary solutions) like Google Engine, Microsoft Azure, and Amazon Elastic Containers [41] at a cost. Both options support Docker containers with an API framework to easily manage the life cycle of containers, acquire container image from Docker repository, build the container components, deliver and deploy the application, run management checks, and maintain containers run-time to meet the quality of service mark.

A. *IP Addressing Scheme and Management*

IP addressing is an integral resource and critical part of large-scale communication infrastructural deployment. IPv4 and IPv6 protocol stacks are both supported in Docker orchestrator frameworks to provide addresses to containers via TCP or UDP [3] and in NAT and Proxy standalone mode [26]. When Docker daemon is implemented using the fixed classless interdomain routing flag, IPv4 functions independently or in combination (dual-stack) when it has been initiated using the IPv6 flag since both protocols share the same physical adapter [42]. The IPv6 flag assigns containers with link-local address (fe80::1) and a globally routable address of subnet size of at least MAC address via the classless interdomain routing [43] in the range used within the network. While IPv4 is the default internal Docker container protocol stack for communication with other host containers, IPv6 is more suitable when overlay network software is introduced to support the massive deployment of containers for distributed services. By clearly defining IPv6 subnets, a container can be assigned a global IPv6 address and load-balanced automatically.

Ways of leveraging the IPv6 support capabilities in Docker have been discussed extensively in [42] because containers are ephemeral and their IP address difficult to manage manually. A good IP addressing scheme ensures a secure, stable, and reliable network when multiple containers have to use the same port for external communication. Port-level NAT based on IPv6 Ethernet Virtual Private Network (VPN) and Identification Locator Addressing (ILA) is a novel solution recommended in [24] to handle IP address mobility and exhaustion issues. ILA, Identifier/Locator Network Protocol (ILNP), and Locator ID Separation Protocol (LISP) provides unique IPv6 addresses for containers in the overlay networks and retains the virtual IP address in another node. The first 64 bits are assigned as Locator (Physical Node) and the last 64 bits assigned as an Identifier (endpoint).

Assigning IP address manually to every container is a difficult task and not sustainable in large-scale deployment for IP mobility and management reasons. A container-based overlay network isolates virtual network instances and enforces multi-tenant IP addressing agnostics. In Docker orchestration frameworks as shown in Fig. 3, every container requires a private IP address to access containers on the same host if IP conflicts are to be avoided. For communications across hosts, unique IP addresses have to be assigned to each container across all the hosts to avoid conflicts and permit traffic flow. Other Docker internal networking architectural configurations like Flannel, Weave, Calico, OpenvSwitch is required. Pipework is another solution used for more complex connection of containers to assign static IP addresses to containers over a single host where necessary. In [24], load balancing and Access Control Lists (ACL) were identified as the components of a Network Function Overlay (NFO), a solution currently supported by Cilium [1].

Container IPv6 addresses and port numbers in Docker orchestration are exposed for external communication either by enabling Neighbour Discovery Proxy (NDP) or Docker

IPV6 NAT [2]. Both require extra work of providing public routable IP address range and non-routable Unique Local Addresses (ULA) respectively. To enable DNP, the Kernel proxy is first activated in the Docker daemon configurable file on a per container or subnet basis. Without additional filtering, all the globally routable IP addresses and ports are exposed, source IP address can be rewritten and IPv6 addresses allocated randomly without proper DNS. A disadvantage which is limiting its widespread use for security reasons. A hybrid solution of IPv4 and IPv6 that would require no changes to the Docker image uses NAT to filter unpublished ports for security reasons. Unfortunately, this solution does not support overlay networks and only applies to Docker bridge networking, which is not a viable option for large scale deployment. Docker IPv6 NAT uses ULA range to limit container public IP routing and port forwarding and maintain IPv6 address routing tables rule difficult to achieve in the NDP.

VII. Discussions

To effectively manage the communication of several separate containerized entities on one platform with less error, low latency, less cost, high reliability, and high security, a robust multi-tenant management system with edge computing model is needed [38]. Docker Swarm is an orchestration tool for managing distributed applications with load balancing and service discovery capabilities. A decentralized cloud-based scheduling infrastructure will allow the workload to be distributed among node clusters in the form of containers (microservices). For the same reason, Mesos frameworks are considered best for large-scale cluster deployment and support up to 50,000 nodes [15]. A containerized Docker Swarm based orchestration management platforms can co-exist with other solutions such as Kubernetes. To achieve high performance in such a scenario, parallel programming should be incorporated into Docker designs [16]. For flexible infrastructural lifecycle management, Docker Swarm allows containers images to be updated and the cluster size to be scaled without having to restart the containers [23]. The Docker Swarm must have an adequate scheduling policy for optimal and balanced utilization of swarm resources, reduction of network overhead through localization of resources for fast access, and the removal of a single point of failure.

For auto-scaling, a Docker solution with high availability, the mix of horizontal and vertical scaling is considered more promising for both local and remote distributed adaptations. Scheduling helps to manage memory demand statically or dynamically. Moving some of the computing tasks to the edge is one way of improving operational efficiency, achieving seamless integration with the cloud and lowering the bandwidth demand, and latency due to reduced congestion but at the expense of energy utilization.

The database management models for the multi-tenant system include the combinations of; one tenant in a database (high isolation gives high security but expensive to maintain the many databases); many tenants one database (schema) – reduced costs but difficult to manage due to table sizes; and many tenants share one database (multi-tenancy model saves costs and structure data sets based on tenants' IDs). The tenant ID is used to create separation attributes added to each shared table.

VIII. Conclusions

In this study, we have established the requirements and strategies for employing Docker Swarm orchestration framework as a suitable container networking management and control tool for multi-tenant systems. For a large-scale distributed system as proposed in this study to support multi-tenancy, ways of managing access conflicts in a cloud virtualized setting, how containers are networked, load-balanced, and discovered must be clearly defined by Docker cluster networking algorithms. Ranging from Docker native network features through the overlay to orchestration systems. For instance, the scheduling algorithm allows container orchestration frameworks to compute the node where a container should be placed for effective load balancing. In future work, the following recommendations will inform the implementation phase of a remote administration and configuration of utility assets: (i) Swarm managers will be spread across hosts to avoid singularity in failure; (ii) To select a suitable multi-tenancy pattern, the tenants' size and security requirements will be considered; (iii) To reduce the infrastructural cost and resource maintenance in multi-tenancy, the amount of cloud computing will be reduced through edge virtualization authority.

Acknowledgment

This research is part of the digital substation core research project at the University of Strathclyde's Power Network Demonstration Centre (PNDC) and prepared within the scope of the Nigeria Petroleum Technology Development Fund (PTDF) award number PTDF/ED/PHD/USO/1092/17.

References

[1] Cilium, "API-aware Networking and Security." [Online]. Available: https://cilium.io/. [Accessed: 29-Jun-2020].

[2] Gregoire Pailler, "RPi4 - Docker with IPv6 and Docker Compose," 08-Aug-2019. [Online]. Available: https://gpailler.github.io/2019-10-08-pi4-part3/. [Accessed: 10-Jul-2020].

[3] R. Dua, S. K. Konduri, and V. Kohli, Learning Docker Networking. Birmingham, UK: Packt Publishing, 2016.

[4] P. J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource Management in a Containerized Cloud: Status and Challenges," J. Netw. Syst. Manag., vol. 28, no. 2, pp. 197–246, Apr. 2020.

[5] M. Samaniego and R. Deters, "Supporting IoT Multi-Tenancy on Edge Devices," in Proceedings - 2016 IEEE International Conference on Internet of Things; IEEE Green Computing and Communications; IEEE Cyber, Physical, and Social Computing; IEEE Smart Data, iThings-GreenCom-CPSCom-Smart Data 2016, 2017, pp. 66–73.

[6] G. Rosinosky, C. Labba, V. Ferme, S. Youcef, F. Charoy, and C. Pautasso, "Evaluating Multi-tenant Live Migrations Effects on Performance," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2018, vol. 11229 LNCS, pp. 61–77.

[7] G. Nikol, M. Trager, S. Harrer, and G. Wirtz, "Service-Oriented Multi-tenancy (SO-MT): Enabling multi-tenancy for existing service composition engines with docker," in Proceedings - 2016 IEEE Symposium on Service-Oriented System Engineering, SOSE 2016, 2016, pp. 238–243.

[8] L. C. Ochei, A. Petrovski, and J. M. Bass, "Optimal deployment of components of cloud-hosted application for guaranteeing multitenancy isolation," J. Cloud Comput., vol. 8, no. 1, p. 1, Dec. 2019.

[9] C. H. Hong and B. Varghese, "Resource management in fog/Edge computing: A survey on architectures, infrastructure, and algorithms," ACM Comput. Surv., vol. 52, no. 5, Sep. 2019.

[10] G. Toffetti and T. M. Bohnert, "Cloud Robotics with ROS," in Studies in Computational Intelligence, vol. 831, Springer Verlag, 2020, pp. 119–146.

[11] E. Truyen, A. Jacobs, S. Verreydt, E. H. Beni, B. Lagaisse, and W. Joosen, "Feasibility of container orchestration for adaptive performance isolation in multi-tenant SaaS applications," in Proceedings of the ACM Symposium on Applied Computing, 2020, pp. 162–169.

[12] E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen, "Towards a container-based architecture for multi-tenant SaaS applications," in Proceedings of the 15th international workshop on adaptive and reflective middleware, 2016, pp. 1–6.

[13] A. A. Adewojo and J. M. Bass, "Evaluating the Effect of Multi-Tenancy Patterns in Containerized Cloud-Hosted Content Management System," in Proceedings - 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2018, 2018, pp. 278–282.

[14] K. Nakanishi, F. Suzuki, S. Ohzahata, R. Yamamoto, and T. Kato, "A Container-based Content Delivery Method for Edge Cloud over Wide Area Network," in International Conference on Information Networking, 2020, vol. 2020-Janua, pp. 568–573.

[15] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks," Appl. Sci., vol. 9, no. 5, p. 931, Mar. 2019.

[16] S. Buchanan, J. Rangama, and N. Bellavance, Introducing Azure Kubernetes Service, 1st edition 2020. Berkeley, CA : Apress, 2020.

[17] D. Vohra, Docker Management Design Patterns. Apress, 2017.

[18] A. Slominski, V. Muthusamy, and R. Khalaf, "Building a multi-tenant cloud service from legacy code with docker containers," in Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015, 2015, pp. 394–396.

[19] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem – Vulnerability Analysis," Comput. Commun., vol. 122, pp. 30–43, Jun. 2018.

[20] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015, 2015, pp. 386–393.

[21] R. Bankston and J. Guo, "Performance of Container Network Technologies in Cloud Environments," in IEEE International Conference on Electro Information Technology, 2018, vol. 2018-May, pp. 277–283.

[22] R. Smith, Docker orchestration : a concise, fast-paced guide to orchestrating and deploying scalable services with Docker (eBook, 2017) [WorldCat.org]. Birmingham, UK: Packt Publishing, 2017.

[23] N. Nguyen and D. Bein, "Distributed MPI cluster with Docker Swarm mode," in 2017 IEEE 7th Annual Computing and Communication Workshop and Conference, CCWC 2017, 2017.

[24] Ł. Makowski and P. Grosso, "Evaluation of virtualization and traffic filtering methods for container networks," Futur. Gener. Comput. Syst., vol. 93, pp. 345–357, Apr. 2019.

[25] W. Binder, V. Cortellessa, A. Koziolek, E. Smirni, and M. Poess, Eds., "ICPE 2017, 8th ACM/SPEC International Conference on Performance Engineering, April 22-27, 2017, L'Aquila, Italy," 2017.

[26] T. Mekonnen et al., "Energy Consumption Analysis of Edge Orchestrated Virtualized Wireless Multimedia Sensor Networks," IEEE Access, vol. 6, pp. 5090–5100, Dec. 2017.

[27] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An Analysis and Empirical Study of Container Networks," in Proceedings - IEEE INFOCOM, 2018, vol. 2018-April, pp. 189–197.

[28] U. Abbasi, E. H. Bourhim, M. Dieye, and H. Elbiaze, "A Performance Comparison of Container Networking Alternatives," IEEE Netw., vol. 33, no. 4, pp. 178–185, Jul. 2019.

[29] M. R. A. S. T. M. Mark Church, "Docker - Docker Swarm Reference Architecture: Exploring Scalable, Portable Docker Container Networks," 2019. [Online]. Available: https://success.docker.com/article/networking#ipaddressmanagement. [Accessed: 12-Jul-2020].

[30] M. Hausenblas, Container Networking From Docker to Kubernetes. United State of America: O'REILLY, 2018.

[31] S. Thiruchadai Pandeeswari, S. Padmavathi, and N. Hemamalini, "Engineering Full Stack IoT Systems with Distributed Processing Architecture—Software Engineering Challenges, Architectures and Tools," in Intelligent Systems Reference Library, vol. 185, Springer, Cham, 2020, pp. 71–87.

[32] R. Toasa, C. Aldas, P. Recalde, and R. Coral, "Performance Evaluation of Apache Zookeeper Services in Distributed Systems," in Advances in Intelligent Systems and Computing, 2019, vol. 918, pp. 356–364.

[33] C. Artho et al., "Model-Based API Testing of Apache ZooKeeper," in Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, 2017, pp. 288–298.

[34] etcd, "A distributed, reliable key-value store for the most critical data of a distributed system." [Online]. Available: https://etcd.io/. [Accessed: 08-Jul-2020].

[35] M. E. Piras, L. Pireddu, M. Moro, and G. Zanetti, "Container Orchestration on HPC Clusters," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2019, vol. 11887 LNCS, pp. 25–35.

[36] CoreOS, "flannel Container Networking | Configuring flannel Networking." [Online]. Available: https://coreos.com/flannel/docs/latest/flannel-config.html. [Accessed: 14-Jul-2020].

[37] Weaveworks, "Introducing Weave Net." [Online]. Available: https://www.weave.works/docs/net/latest/overview/. [Accessed: 15-Jul-2020].

[38] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl, "A Containerized Big Data Streaming Architecture for Edge Cloud Computing on Clustered Single-board Devices," in In Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER), 2019, pp. 68–80.

[39] Docker, "docker.github.io/admin_guide.md at v17.06 · docker/docker.github.io · GitHub," 2017. [Online]. Available: https://github.com/docker/docker.github.io/blob/v17.06/engine/swarm/admin_guide.md. [Accessed: 30-Jun-2020].

[40] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in Annual Technical Conference ({USENIX}{ATC}, 2014, pp. 305–319.

[41] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," Concurr. Comput. Pract. Exp., p. e5668, Jan. 2020.

[42] J. Langemak, Docker Networking Cookbook. Birmingham: Packt Publishing, 2016.

[43] Ajeet Raina, "Walkthrough: Enabling IPv6 Functionality for Docker & Docker Compose," 05-Aug-2017. [Online]. Available: https://collabnix.com/enabling-ipv6-functionality-for-docker-and-docker-compose/. [Accessed: 10-Jul-2020].