University of Nice Sophia Antipolis/Universite Cote d'Azur

Ecole Polytechnique Nice-Sophia

An Analysis and Empirical Study of Docker Networking

A final report of M.Sc. (UBINET) Internship

at

i3s/CNRS Research Labs - Sophia Antipolis, France

Student:
Yusuf Haruna

Supervisors:
Prof. Guillaume Urvoy-Keller
Ass. Prof. Dino Lopez-Pacheco

August 2019

Abstract

The improvements in the performance brought by lightweight virtualization such as containerization cannot be overemphasize when compared with the performance of traditional virtualization (heavy virtualization). Containerization provides the possibility to consolidate more containers in a machine. Due to their lightweight nature, containers fit well with new applications' design. To benefit from these improvements many applications are now running in containers. As a result of their multi-tier/distributed nature, the running applications need to communicate. Network virtualization enables to create a dedicated network overlays spanning over several VMs or physical hosts to interconnect application fragments, hence there is a need to understand their performance. In this work, we use a variety of applications to benchmark the performance of different container interconnection solutions. We picked them similarly to [1] as they represent a variety of applications in terms of needed resources (CPU, RAM, networking). Experiments were carried out with three applications viz. Memcached, Nginx and PostgreSQL. We further added *iperf3* to this list. Each of these applications was installed inside a container in one VM and their corresponding benchmarks (test client) in a separate container in another VM in order to benchmark the performance of the applications. The VMs were interconnected using four modes namely: host, NAT, Docker default overlay (VXLAN) and weave. In most of the cases, host mode recorded the best performance, followed by NAT and the overlay networks (VXLAN and Weave) have the least performance due to packet encapsulation. In each case, *sar* was used to monitor the CPU utilization. We were able to reduced the overhead of the two overlay networks using RPS technique because they brought solutions to some of the problems faced when connecting containers using host and NAT modes in the cloud.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Virtualization technology is one of the great achievement in the field of computer science, as it enables users to create virtual (rather than physical) [2] hardware resources for applications deployment. However, the overhead of this technology remains a critical concern due to the fact that, traditional virtual machines (VMs) behave like a full-fledged Operating System (OS). This introduces limitations on how many VMs can be consolidated on one physical machine. In addition, the VMs long startup times makes it difficult to host short-lived applications on the machines [3].

These problems were solved by a nice improvement to the virtualization technology, known as container-based virtualization which works by sharing the kernel and the libraries of the original (host) operating system among the running applications. It works by running them in an isolated namespace called container. A namespace is a way of logically separating processes along different dimensions: Network, IPC, User, PID, Mount or UTS namespace. A container is a kind of lightweight virtualization environment compared to traditional VMs. It enables partitioning of hardware resources to users in order to speedup deployment of applications [3]. Some of the advantages are: containers provide lightweight virtualization hence the possibility to consolidate more containers in a machine, containers fit well with new application design where applications are structured into micro-services interconnected with one another and so on. The most popular containerization solutions currently in the market is Docker [4, 5]. Figure 1.1 shows a comparison between traditional virtualization which uses a hypervisor for creating the virtual machines (guest OSs) where each of them has its own separate libraries and binary files. On the other hand, lightweight virtualization using Docker which allows running of applications in containers. The containers share the kernel and other files of the same OS.

However, for an efficient computation to take place, the running applications within the containers need to communicate with each other either within the same host or between multiple hosts. These communications are of critical concern due the fact that,

the performance of a system is significantly affected by the characteristics of the communication. Hence, the need to study and analyze different container networking models in order to make good choices of the right model to use on a given system.

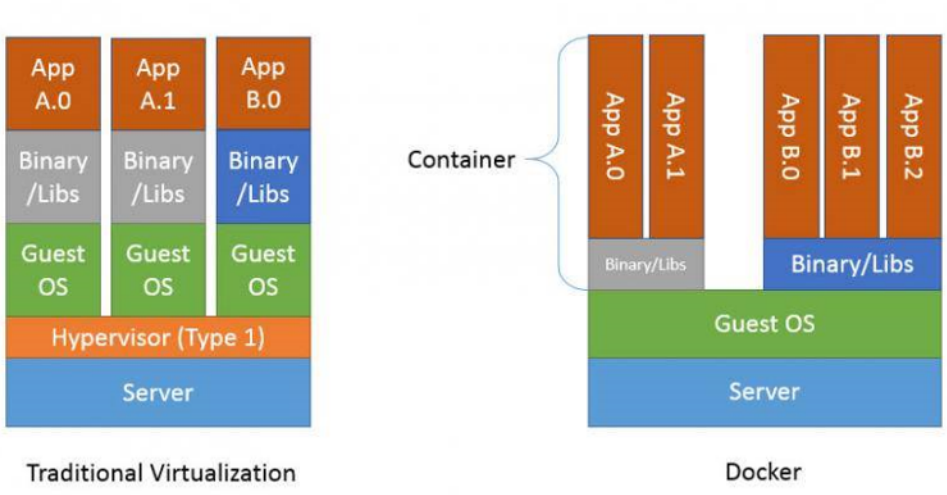


Figure 1.1: Traditional Virtualization Vs Lightweight Virtualization

Source: https://www.docker.com/

1.1.1 Motivations

Many applications are being deployed in the cloud nowadays, which brought about quite a number of improvements to modern days computing. This enables companies to focus more on their core business for better satisfaction of their clients instead of spending more resources on the computing infrastructure and their maintenance. As of today, a significant fraction of companies use cloud computing solutions for their work. They might use a third party public cloud solutions like Amazon web services, Microsoft Azure etc. or even private cloud solutions for example with Openstack to achieve their business target. These developments are possible because of the virtualization technology. Container-based virtualization being a lightweight virtualization (which behaves by partitioning the hardware recourses) has even more advantages than traditional virtualization (which behaves like a full Operating System). These advantages and many other make computing easier and cheaper nowadays. However, this development cannot be possible without good performance of communication between containers, hence, it is of great importance to study and analyze the container networking performance.

An emblematic example is how search engines use container-based virtualization, for instance google search engine launches almost 7,000 containers every second [3] and these containers communicate with each other in order to deliver the result of the google searches. This obviously raises the need for good networking performance.

1.1.2 Objectives

Initially, the project started as a PFE project where we reproduced the result of a paper [3] published in the year 2018 at INFOCOM which is an IEEE conference. We tested the performance of Docker container networking in single Virtual Machine (VM) and multple VM scenarios. We used a number of benchmarks in carrying out the analysis including those used in the paper like *Sparkyfish* [6], *Sockperf* [7] and a new one that was not used in the paper which is *iperf3* [8]. In this work, we intended to do perform the analysis similarly to what was done in [1]. The following are the objectives of this work:

- Understand the performance of different Docker Networking solutions. We focused on four modes (host, NAT, Docker default overlay and weave) of connecting containers in the cloud which were explained in section 3.2 of this report.
- Build a realistic testbed by selecting three popular cloud applications (and their benchmarks as explained in Chapter 4) and deploying them into Docker containers so as to benchmark their performance in the four modes. We also performed similar analysis with *iperf3* which performs an active measurement.
- Obtain some results by testing our testbed and monitor system level performance with sar (System Activity Reporting) which is a Unix System V-derived system monitor command.
- We would like to also use statistical tools like boxplot and standard deviation in order understand the level of variability of the results (thirty samples in each case) in order to get statistical significance.
- Check if we can reduce the overhead of the overlay Networks using OS/hardware support.

1.1.3 Challenges

The main challenge of this project was to tune the testbed and perform some tests with RPS/RFS (Receive Packet Steering/Recive Flow Scaling). They are Linux kernel support in Linux networking stack to increase parallelism and improve performance for multi-processor systems. These techniques can be used to reduce the overhead of the overlay networks used in connecting Virtual Machines (VMs) in the cloud.

1.2 Metrics of our Evaluation

1.2.1 Network performance metrics

Throughput

Is defined as the number of items processed per unit of time, for example bits transmitted per second, HTTP operation in a day or millions of instructions per second (MIPS) [9].

In this work we considered network throughput which is measured in bits per second (bps). The higher the throughput the better the performance.

Latency

Is define as the time between making a request and beginning to receive a result. It is measured in units of time such as second, millisecond, nanosecond and so on [9]. In this work we considered network latency and presented all the results in millisecond (ms). The lower the latency the better the performance.

1.2.2 System level metrics

For monitoring the usage of the system, we focused on the following metrics reported by *sar* [10] related to CPU consumption.

User: Is the amount of time the CPU spent when executing in the user space [11].

System: Is the amount of time the system CPU was busy executing in the kernel space [11].

I/O wait: Is the amount of time the CPU spent waiting for an I/O operation to finish and the CPU can not be used for other things [11].

Chapter 2

State of the Art

Docker is an open-source software, which allows automation of applications deployment into containers. This software was produced by a team at Docker Inc. It was designed in such a way that application deployment engine is added on top of a virtualized environment that allows execution of containers [4, 5].

Containers are such an environment that can host several processes and each process can have its network stack. Containers incur less overhead when compared with virtual machines. A huge number of applications that are frequently launched and terminated within a second can be deployed using containers. There are different modes to connect containers that are organized in different scenarios in order to enable communications between the running applications either in a single VM or between multiple VMs [3]. Indeed, in a lot of deployment scenarios, containers are deployed within VM and not directly on the host OS. The main reason behind this is security as containers do share the host kernel and any breach in the containerization engine might compromise the whole machine. In our work, we consider only the case of containers running inside VMs. The performance of the network connections between containers is of paramount importance. A number of studies were carried out in order to understand how container-based virtualization works but most of them did not study the problem from networking point of view. For example [12] tried to evaluate the performance on high performance computing environments. They conducted the study on Computing Performance, Memory Performance, Disk Performance, Network Performance (on a very narrow perspective), Performance Overhead in HPC (High-performance computing) Applications and Isolation. Another study is [13] which focused on the networking aspect but yet does not study the problem in general, as it restricts only on IoT devices. It studied the impact of container virtualization on network performance of IoT Devices.

A nice and comprehensive study on this problem is [3] which studied and analyzed different modes of connecting containers on both single VM and multiple VMs cases. This can help users on their decision making process when deciding which mode to use when connecting their containers. This study seems to be the first of its kind. The authors

studied and analyzed the performance of Docker container networking in two scenarios. The first scenario is when the containers are within the same VM. They carried out the study using bridge mode, container mode and host mode relative to without container mode with a number of benchmarks that perform an active test. The second scenario is when the containers are running in different VMs. The modes used in connecting the containers in this scenario are host mode, NAT and overlay networks (Docker default overlay, weave [14], flannel [15] and calico [16]). Simple Benchmarks (bulk transfer tools similar to iperf) that perform active test were also used. They observed that the overlay networks used in the multiple VM case have some significant overhead on the performance of the network due to packet encapsulation (explained in section 3.2 of this report). Nevertheless, these overlay networks have brought solutions to some of the problems (for instance port contention, scalability problem and so on) faced when connecting containers using host and NAT modes, hence, the overlay networks are highly used in networking containers in the cloud. Yet this study did not take into account on how to find ways to reduce the overhead of the overlay networks despite their importance. A clear weakness of the the study was that it was conducted without deploying some applications into the containers when carrying out the experiments. This will constitute a starting point for our work.

A recent study [1] published at USENIX conference in the first quarter of this year tried to reduce the overhead of one of the overlay networks (weave). They used OS kernel support to reach this goal by deploying some applications into containers and evaluating the performance of the network.

In our work we use an approach similar to what was done in [1]. We picked and deployed some popular cloud applications similar to what they deployed in order to understand the performance of Docker networking modes. As compared to [1], we used Network Address Translation (NAT) mode of container networking in addition to what they used. Moreover, since in their study they tested with only one overlay network which is weave, while we tested also another overlay network which is Docker default overlay network which uses virtual extensible LAN (VXLAN) tunnel in connecting containers. We built a testbed in order to achieve this target. We understood the performance of the four modes (Host, NAT, Docker default overlay (VXLAN) and weave) by deploying applications like Memcached, Nginx and PostgreSQL together with their benchmarks. We also also carried out experiments with iperf3. Finally, we were able to reduced the overhead of the two overlay networks using RPS (explained in section 6.1 of this report) by testing on one of the three applications which is PostgreSQL.

Chapter 3

Experimental settings and Container Networking modes

3.1 Experimental settings

3.1.1 Hardware

We carried out the experiments on a HP machine which has 12GB memory, Intel(R) core(TM) i7-6500U CPU @ 2.50GHz (4 CPUs) approximately 2.6GHz processor, and WDC WD10JPVX-60JC3T0 1TB hard disk.

3.1.2 Software

We used Ubuntu 16.04 and Linux kernel 4.15.0-45-generic as both host and guest OS. The hypervisor was KVM version 2.5.0 where the VMs were assigned with the virtio NIC driver, 2vCPUs and 4GB RAM each. Docker version was 18.09.2 Community Edition and weave version was 2.5.0.

3.1.3 How our testbed works on the VMs

For each test, a container was created using Docker [4] in one Virtual machine (VM) where the application was installed and the corresponding benchmark was installed in another container in a different virtual machine. When the test is going on, sar (system activity reporting) also runs in parallel in order to monitor the performance of the system (CPU utilization in our case), a draft of this configuration is shown in Figure 3.1. The two VMs were connected using one of the four modes described in section 3.2 i.e, host, NAT, VXLAN and Weave modes. There are two ways of deploying containers either on a VM because of security for example by cloud provider or on a physical machine for example by Google. In this study we carried out the experiments by deploying the containers in VMs.

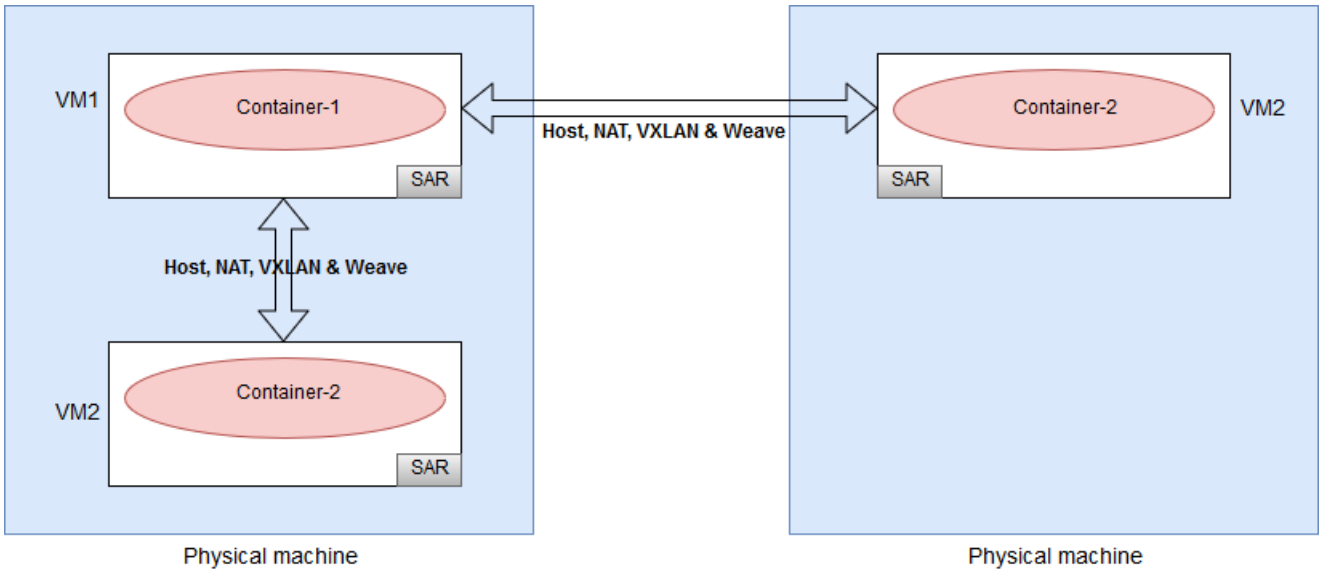


Figure 3.1: The Virtual Machines configuration in the physical machine

3.2 Container Networking modes

3.2.1 Host mode

In this mode, the containers share the namespace of the host OS. A Namespace is a way of logically separating processes along different dimensions: Network, IPC, User, PID, Mount or UTS namespace. Since the containers share the stack of the host, this allows communication between containers in multiple hosts using the IP addresses of the host machines [3]. Figure 3.2 shows how containers can be connected in this mode within a host that has an IP address of 10.1.1.1. Any external host can connect with the containers using their respective port number plus the IP address of the host. For instance, the first container can be reach using 10.1.1.1:80, the second can be connected using 10.1.1.1:81 and so on. If many containers listen on the same port number, host mode won't work as only one container is allowed to bind to a particular port number. However, one drawback of this mode is that, for two containers on different hosts to talk, they need host ip and port number. If that host died, the containers are restarted on another host and redo the entire process and this leads to *port contention* which is a problem to the connectivity of the containers.

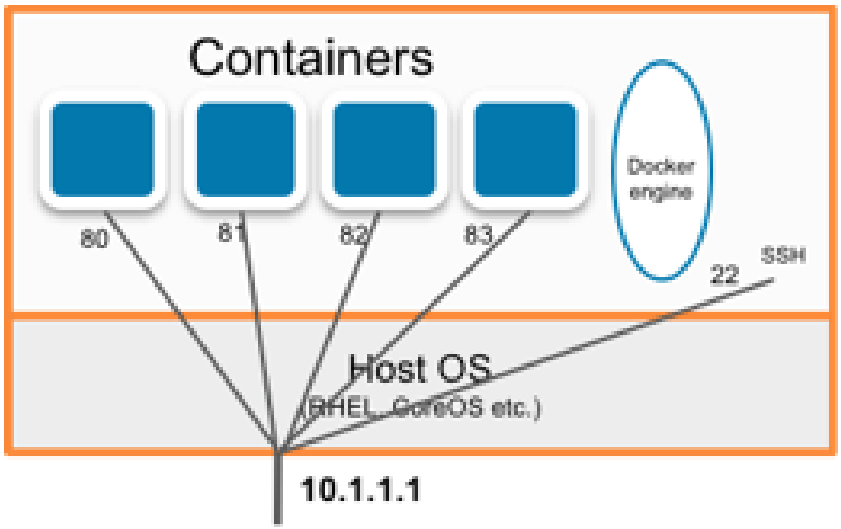


Figure 3.2: Host Mode Networking

Source: https://www.onug.net/blog/container-networking-easy-button-for-app-teams-heartburn-for-networking-and-security-teams/

3.2.2 Network Address Translation (NAT)

NAT is a process of remapping a particular Internet Protocol (IP) address space into another by changing network address information in the IP header of packets as they move across a traffic routing device [17]. It allows containers from different machines to be able to communicate using the public IP address of the host machine and port number of the container. NAT maps the private address of a container to its port number in a NAT table. For the containers to communicate, the public IP address of their host machine is used plus a port number to identify a particular container [3]. In Linux NAT is configured using *iptables*, which is a user-space utility that allows a system administrator to configure tables. The table can either be a filter table, **NAT table** (which was used in this context) or mangle table. Figure 3.3 shows a simple NAT design that can be configured with iptables. Let us say the three boxes in the figure are containers, and their private IP addresses are 192.168.10.100, 192.168.10.101 and 192.168.10.102 while the host IP address is 192.168.10.1. To identity a particular container from the external world (internet or another host), this IP address plus a port a number are used and the mapping between this pair and the private IP address of the containers is stored in a NAT table [18, 19].

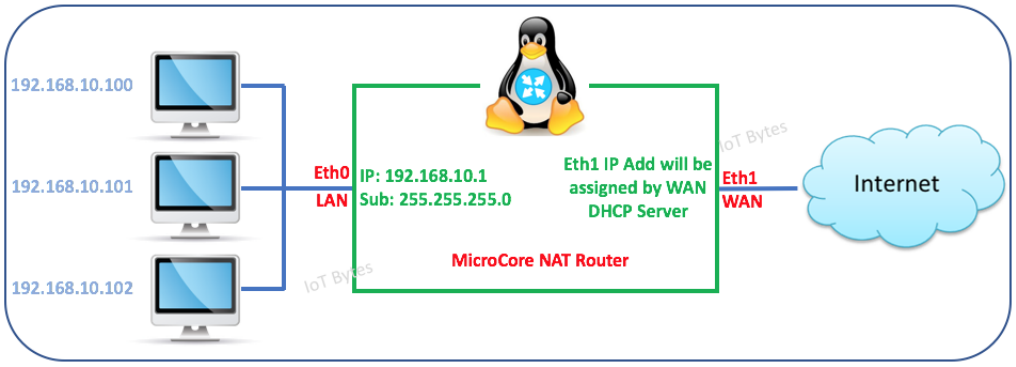


Figure 3.3: NAT configuration in Linux with iptables

Source: https://iotbytes.wordpress.com/
configure-microcore-tiny-linux-as-nat-p-nat-router-using-iptables/

3.2.3 Docker default overlay (VXLAN)

An overlay network runs on top of another network to build virtual links between nodes, Docker default overlay uses Virtual Extensible LAN (VXLAN) to create a virtual link between multiple hosts. VXLAN is a technology in network visualization that solved the scalability issue related to large deployment in the cloud. It make use of VLAN-like (Virtual LAN) encapsulation style to encapsulate OSI layer 2 Ethernet frames inside layer 4 UDP datagrams [20]. Many Docker container overlay networks exist but the main idea behind each is that, containers store the mapping between their private IP addresses and their host IP in a key-value (KV) store that is accessible from all hosts. Containers use private IP addresses within a virtual subnet to communicate with one another. However, an overlay put an additional layer in the host network stack [3]. When a container sent a packet, the overlay layer looks for the host IP address of the destination in the KV store using the private IP address of the destination container in the original packet and then create a new packet with the host IP address of the destination. It then insert the original packet sent by the container as the new packet's payload, this process is known as packet encapsulation. When the encapsulated packet reaches the destination host, the host network stack decapsulates the wrapped packet to recover the original packet and sends it to the destination container using the container's private IP address. One drawback of overlay network is that, this process introduces some overhead in the performance. Figure 3.4 shows the encapsulation process between two hosts that are connected using a VXLAN tunnel. Virtual Tunnel End Point (VTEP) is an endpoint (a VMkernel interface) that is responsible for encapsulating an Ethernet frame in VXLAN frame or decapsulating a VXLAN frame and forward the original frame (inner Ethernet frame) to its destination. Moreover, Figure 3.5 shows how the inner Ethernet frame is encapsulated in a User Datagram Protocol (UDP) packet, the default UDP port number assigned by Internet Assigned Numbers Authority (IANA) is 4789.

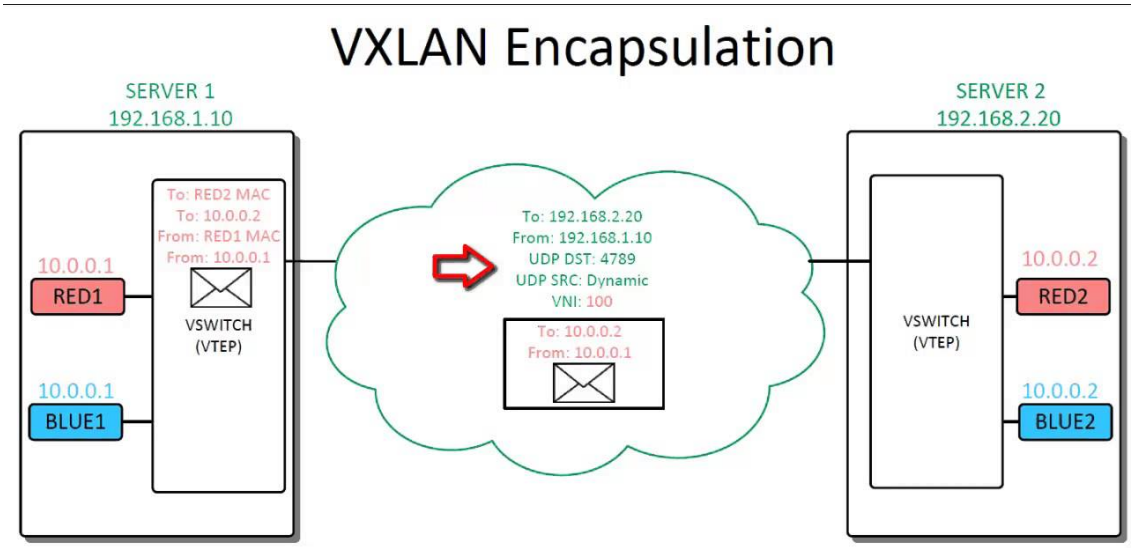


Figure 3.4: Docker overlay network using VXLAN

Source: https://www.youtube.com/watch?v=Jqm_4TMmQz8

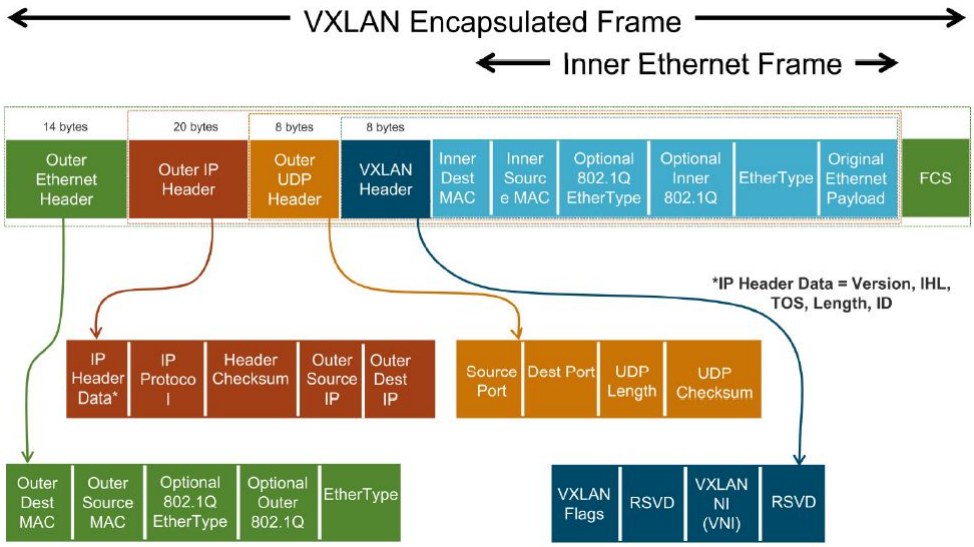


Figure 3.5: VXLAN frame

Source: https://matscloud.blogspot.com/2014/12/how-vxlans-work.html

Overlay Driver Internal Architecture

Docker Swarm [21] is an open-source container orchestration platform, its control plane can be used to automates all the provisioning of an overlay network. Neither VXLAN configuration nor Linux networking configuration is required. An optional Data-plane encryption feature is automatically configured by the overlay driver as well when the network is created. A user or network administrator can simply define the network and attach containers to the network. When creating the overlay network, the Docker Engine builds the network infrastructure needed for overlays on every host. A Linux bridge is built for every overlay together with its associated VXLAN interfaces. The Docker Engine logically instantiates overlay networks on hosts only if the attached container to the network is due on the host. This averts sprawl of the overlay networks in a situation where the connected containers are not in existence [4]. Figure 3.6 shows what is added on a given host and in every container when creating an overlay network.

- Docker_gwbridge: this is the egress bridge for the traffic that goes out of the cluster.
- ovnet: is the point for the ingress and egress of the overlay network that VXLAN encapsulates traffic that goes between containers within the same overlay network. It also expands the overlay across entire hosts that join this particular overlay. There is only one for an overlay subnet on a host and it has the same name as the overlay network.

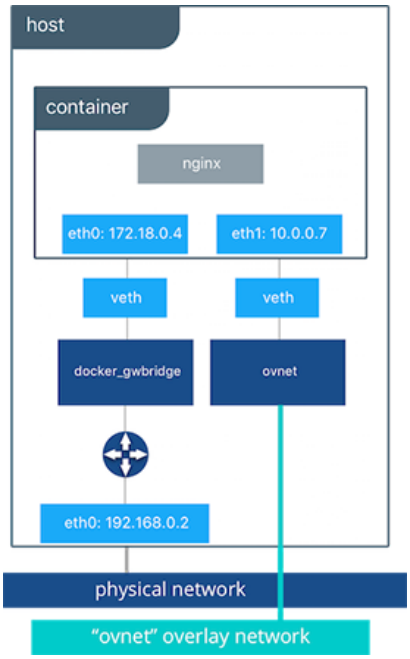


Figure 3.6: Internal Architecture of overlay driver

Source: Docker website

3.2.4 Weave

Weave is another implementation of overlay network, it is a virtual network solution produced by Weavework. It uses a weave router container on each Docker host and the network is made from these connected weave routers. Weave creates a network bridge called *vethwe-bridge* on every host in order to connect containers with the weave router and this connection is via a veth pair. When a container sends a packet, the weave in the host captures the packet and then remove the local traffic and forward it to the weave router on the other host. When the packet reaches the recieving host, the weave router forward it to the bridge and then the bridge move the packet to the destination container [3]. Figure 3.7 shows how weave can be used to connect two containers between two different hosts. Weave routers create TCP (Transmission Control Protocol) connections with one another. They perform a protocol handshake and then exchange topology information over these connections which are encrypted if configured. UDP (User Datagram Protocol) connections are also establish by peers through which encapsulated packets flow. Weave uses exactly same encapsulation mechanism as that of VXLAN [14]. Some of the advantages [40] of weave are as follows:

- Weave network is fast: automatically chooses the fastest path between two hosts without intervention of the user.
- Weave network is secure: user can encrypt traffic on the weave network.
- Multicast Support: it fully support multicast addressing and routing. Data can be forwarded to one multicast address and it will be broadcasted automatically to all recipients.
- NAT Traversal: it has built-in NAT traversal.
- Operates in partially connected Networks: it can forward traffic between nodes even if the mesh network is partially connected.
- No external cluster store required: a cluster store that is a central database like Consul or Zookeeper is required to be set up in all other Docker networking plugins before you can use them but weave does not require.
- Run with Anything: Kubernetes, Mesos, Amazon ECS, and so on.

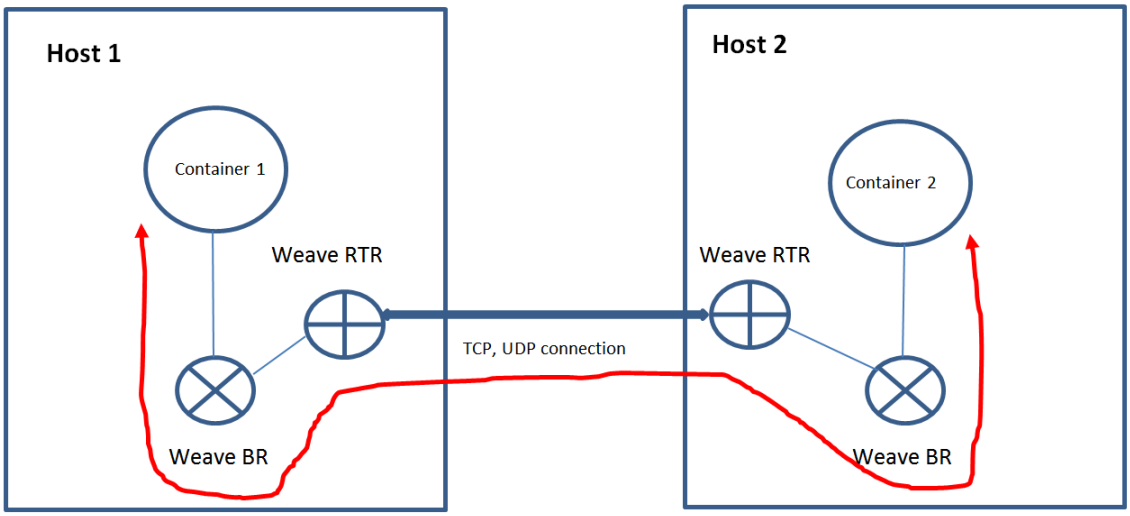


Figure 3.7: Weave network between two containers on two hosts

Source: https://sreeninet.wordpress.com/2015/01/18/docker-networking-weave/

Chapter 4

Applications

4.1 Iperf3

Iperf is a tool used to test maximum achievable throughput on IP networks [8]. We used this benchmark to test TCP and UDP throughputs by running the server in one container and the client in another container within two separate VMs. We used it as its a de facto standard in the performance evaluation community for testing throughput. A simple configuration of iperf3 client and server is shown in Figure 4.1.

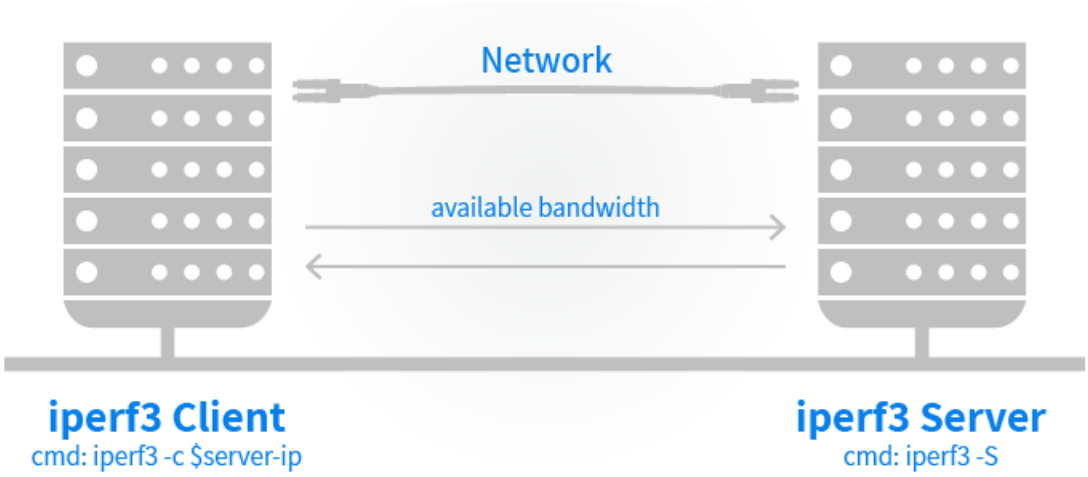


Figure 4.1: iperf3 throughput test

Source: https://datapacket.com/blog/10gbps-network-bandwidth-test-iperf-tutorial/

4.2 Memcached

Memcached is a distributed memory object caching system, it is used to speed up websites that work with dynamic database by caching data and objects in the RAM in order to

minimize the number of times the main data storage must be read [22]. Figure 4.2 illustrates a simple scerario of how memcached can be used to optimise the usage of the cache and Figure 4.3 illustrates how memcached can be used to speedup data access. Without Memcached, server 1 cannot access the objects in the RAM of server 2 and must perform a query on the back-end server. We measure its performance by running the memcached server inside a container within one VM and run a standard memcached benchmark tool *memtier_benchmark* [23] inside another container in a different VM. The benchmark tool spawns four threads, where each thread creates fifty TCP connections to the Memcached server and reports the average number of responses per second, the average latency for responding to a memcache command, and the distribution of response latency. We also measured the latency of SET and GET operations, SET is an operation to store data into the memory in form of key-value while GET is an operation which is used to retrieve the stored data by supplying a key. The SET:GET ratio for test was 1:10. We carried out the experiments with two memcached protocols *memcache_text* and *memcache_binary*.

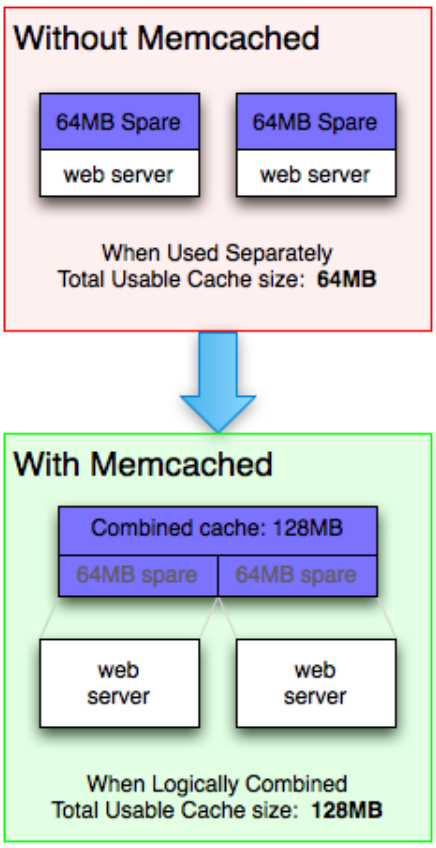


Figure 4.2: Memcached usage

Source: memcached website

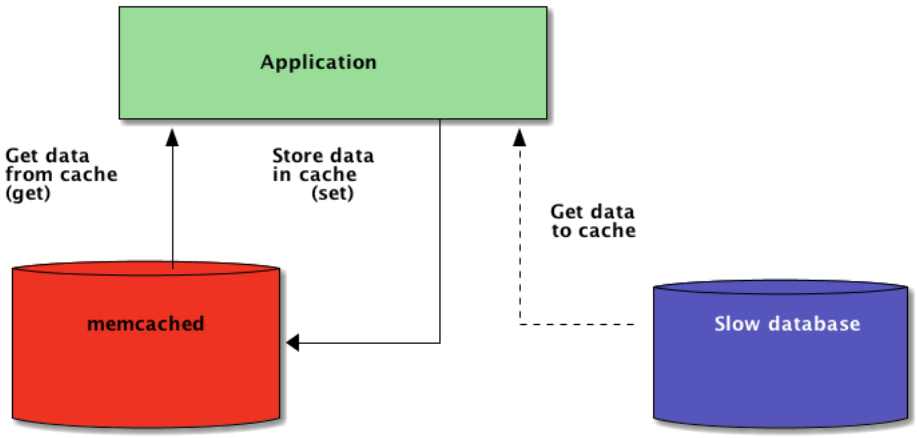


Figure 4.3: Memcached usage

Source: https://julien.danjou.info/python-memcached-efficient-caching-in-distributed-applications/

4.3 Nginx

Nginx is an HTTP server which can also serve as reverse proxy, mail server and generic TCP/UDP proxy server [24, 25, 26]. Figure 4.4 shows different functionalities of nginx. We run the server in one container inside one VM and a standard web server benchmarking tool *wrk2* [27] in another container within a separate VM. The tool spawns two threads and creates a total of 100 TCP connections in order to request for an HTML file from the nginx server. The throughput in requests/second can be set in the tool which outputs the latency. We set up two HTML files of size 1MB and 1KB on the nginx server and perform the test with a throughput of 3K reqs/sec similarly to what was done in [1].

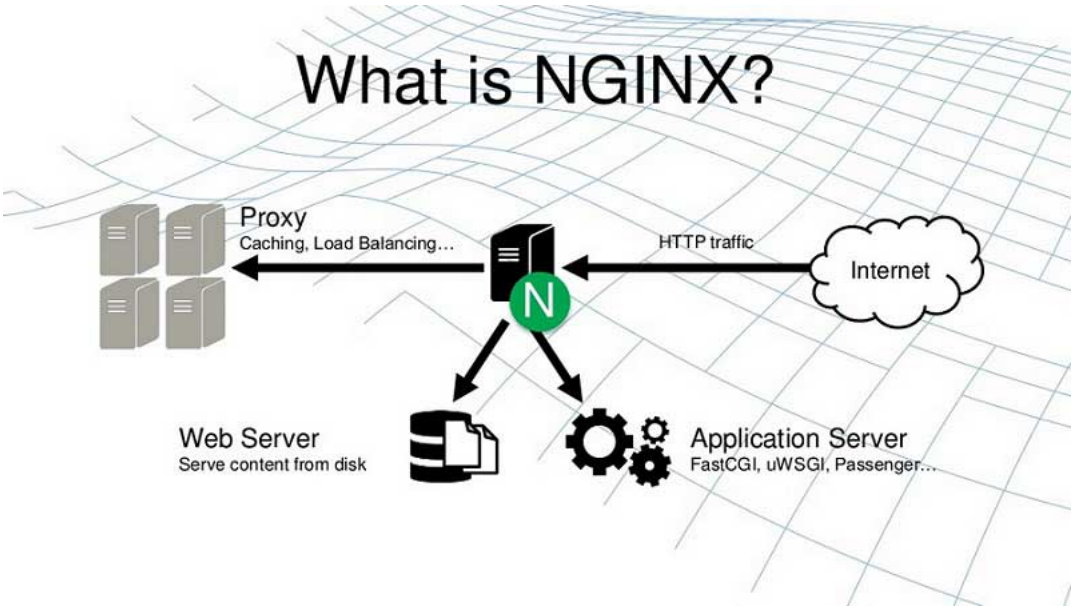


Figure 4.4: Nginx server

Source: https://scaleyourcode.com/blog/article/2

4.4 PostgreSQL

PostgreSQL is an open source Relational Database Magagement System (RDBMS) [28]. Figure 4.5 shows a simple postgreSQL server from user's point of view. In this case, the postgresql server was run by deploying a relational database in a container and then use its default benchmark tool, *pgbench* [29] to benchmark the performance of the server in another container in a separate VM. The tool creates a database of one million banking accounts and executes transactions with a total of 100 connections within four threads.

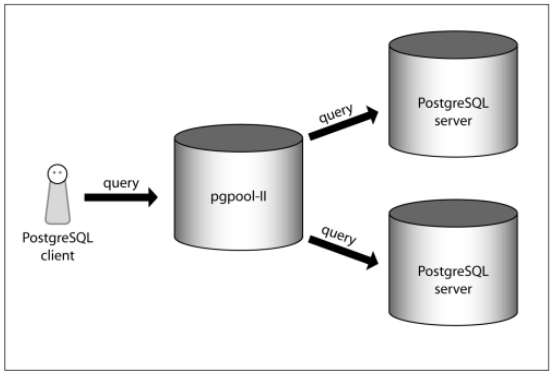


Figure 4.5: postgrsql

Source: PostgreSQL replication, second edition

Chapter 5

Experimental Results

5.1 Iperf3

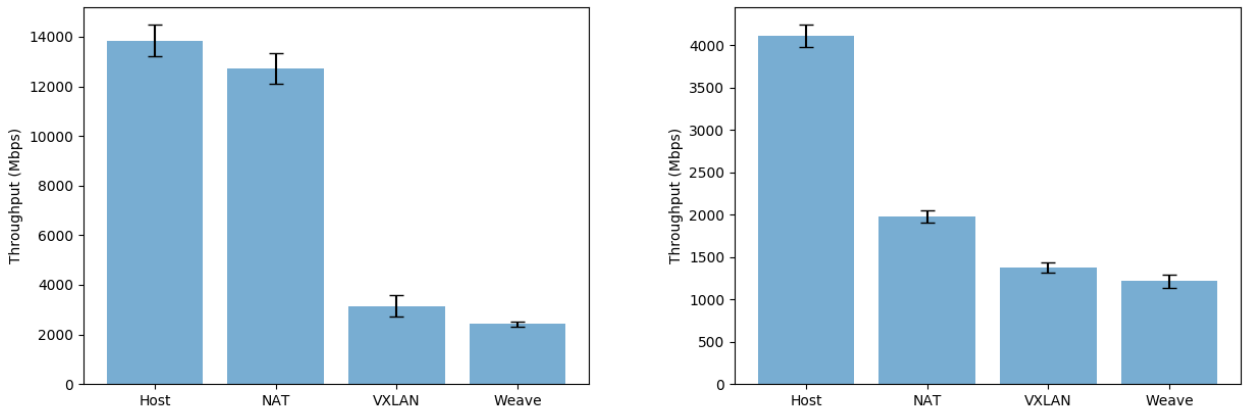
We collected a total of thirty (for each mode) samples of the result by running the shell scripts of our testbed. It created the container, install iperf3 inside and started the server in one VM and the client in another VM on the four modes of container connections between multiple hosts. We carried out the experiments with the two popular protocols TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). We used bar plots to represents the average and error bars (which denote the standard deviations) to analyze the results and boxplots to visualize the level of variability of the results. Table 5.1 shows the average and standard deviation of the TCP throughput in Mbps while table 5.2 shows that of UDP. Host mode achieved the highest throughput followed by NAT which dropped by 8% when compared with host mode, VXLAN and weave have the least throughput with a drop of 77% and 82% respectively, in the case of TCP. Moreover, in the UDP case NAT dropped by 52% while VXLAN and weave dropped by 66% and 70% respectively. The following figures show the bar plots (Figure 5.1) and boxplots (Figure 5.2) of both TCP and UDP results. The results have less variability especially in VXLAN and weave modes. Figure 5.3 shows the CPU utilization of the client and server where the client consumed more CPU than the server except in weave mode. In both cases, most of the CPU was spent in the kernel part more than the user part and this is in line with the fact that it is the kernel that actually does most of the job of packet sending, which is what iperf does. Both the client and server have very less I/O in all of the modes.

Modes	**TCP throughput in Mbps**	
	Average	Standard deviation
Host mode	13849.866	634.865
NAT	12739.733	620.433
VXLAN	3148.266	422.759
Weave	2416.0	93.614

Table 5.1: iperf3 TCP throughput

Modes	**UDP throughput in Mbps**	
	Average	Standard deviation
Host mode	4113.066	129.206
NAT	1979.466	70.493
VXLAN	1375.466	62.582
Weave	1217.066	75.830

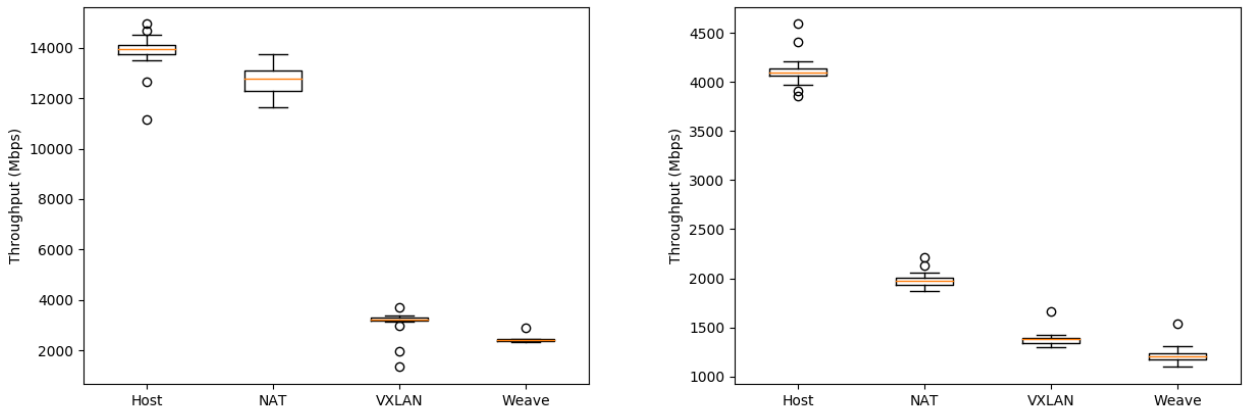
Table 5.2: iperf3 UDP throughput



(a) TCP

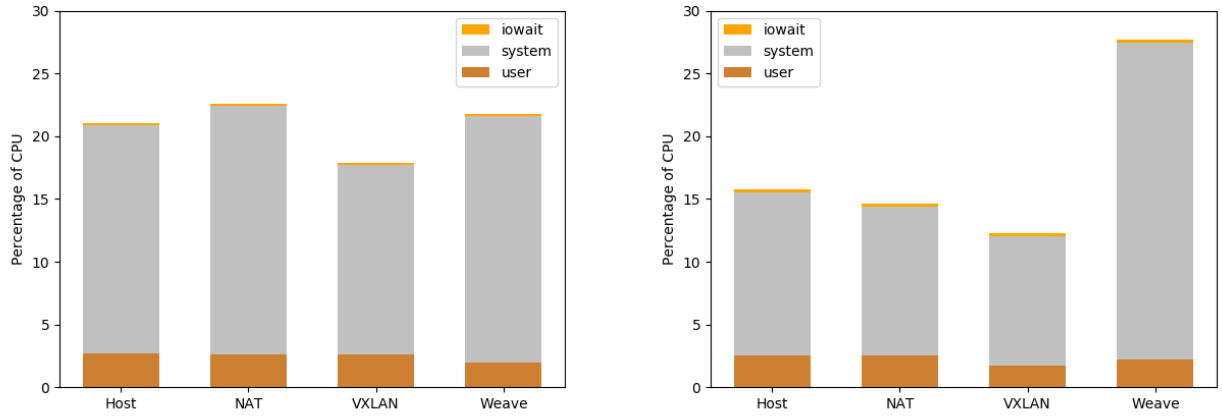
(b) UDP

Figure 5.1: iperf3 throughput



(a) TCP (b) UDP

Figure 5.2: iperf3 throughput boxplot



(a) Client (b) Server

Figure 5.3: CPU utilization of iperf3 client and server

5.2 Memcached

We collected a total of thirty (for each mode) samples of the result by running the shell scripts of our testbed. It created the container, installed memcached server inside and started the server in a container on one VM. It then installed *memtier_benchmark* in another container and start the test on a separate VM. The VMs were connected using one of the four modes of container connections in each run of the experiments. We carried out the experiments using the two protocols of memcached which are *memcache_text*

and *memcache_binary*. We used bar plot with error bars (to denotes the standard deviation) to analyze the result and boxplot to visualize the level of variability of the results.

For the *memcached_text* protocol, the following tables show the average throughput in Kbps and standard deviation (Table 5.3), average latency in millisecond and the corresponding deviation (Table 5.4). Host mode achieved the highest throughput followed by NAT which dropped by 23% compared to host but VXLAN and weave recorded almost same throughput they both dropped by about 34%. Furthermore, host has the least latency followed by NAT which increased by 26% then VXLAN and weave with an increase of 49% and 48% respectively. Figure 5.4 shows the bar plot of the throughput and latency while figure 5.5 shows the corresponding boxplots where less variability was observed with few outliers. Table 5.5 shows the mean and standard deviation of SET operation latency also in millisecond while table 5.6 shows the latency of GET operation. The distribution of latency for Memcached SET and GET operations is shown in figure 5.6 where the two overlay network lines overlapped in both cases which means the performance of the overlay networks is almost the same. Figure 5.7 shows the CPU utilization of the client and server where the client in which the benchmark (*memtier_benchmark*) was installed consumed more CPU than the server (running the memcached server) except in VXLAN mode. The client CPU consumption in the kernel is almost equal to that of the user in all of the four modes and has very low I/O. On the other hand, the server spent more time in the kernel part than the user part in all of the four modes except VXLAN.

Modes	**Throughput in Kbps**	
	Average	Standard deviation
Host mode	11688.304	2238.144
NAT	9149.296	1222.44
VXLAN	7776.544	1066.856
Weave	7790.952	670.744

Table 5.3: Memcached throughput with *memcache_text* protocol

Modes	**Latency in msec**	
	Average	Standard deviation
Host mode	4.1105	0.5641
NAT	5.1807	0.469
VXLAN	6.1569	0.9858
Weave	6.0802	0.7295

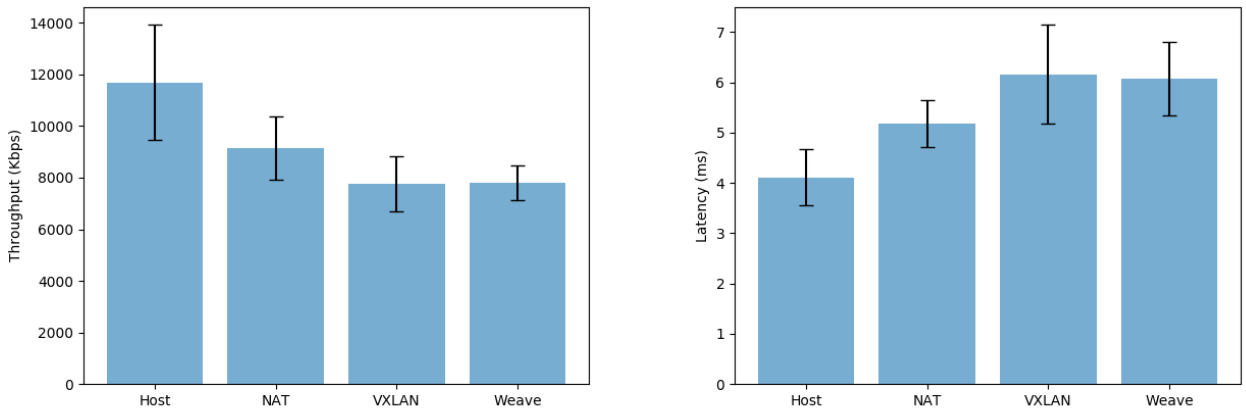
Table 5.4: Latency for responding to Memcached command with *memcache_text* protocol

Modes	**SET latency in msec**	
	Average	Standard deviation
Host mode	4.506	1.029
NAT	5.573	0.774
VXLAN	6.781	1.676
Weave	6.730	1.182

Table 5.5: Latency of Memcached SET operation with *memcache_text* protocol

Modes	**GET latency in msec**	
	Average	Standard deviation
Host mode	4.07	0.549
NAT	5.14	0.470
VXLAN	6.09	0.93
Weave	6.01	0.759

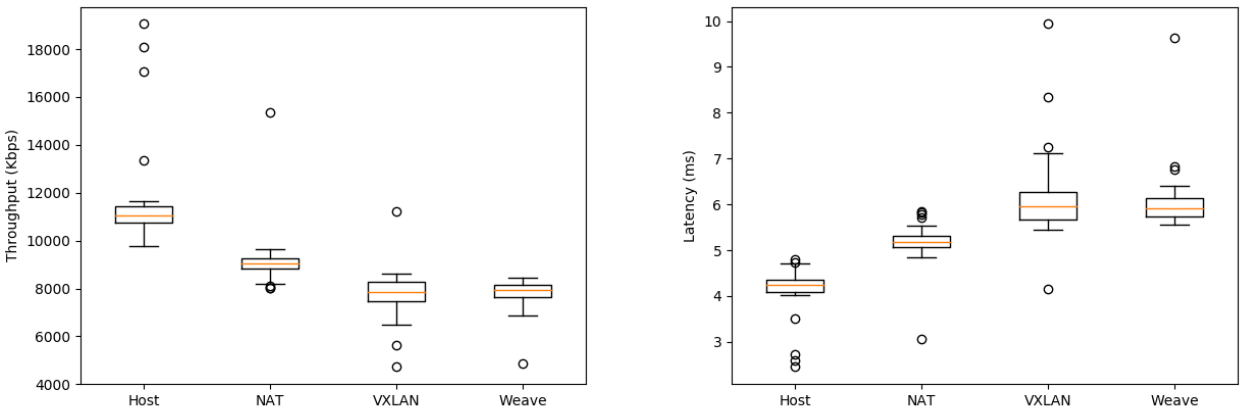
Table 5.6: Latency of Memcached GET operation with *memcache_text* protocol



(a) Throughput

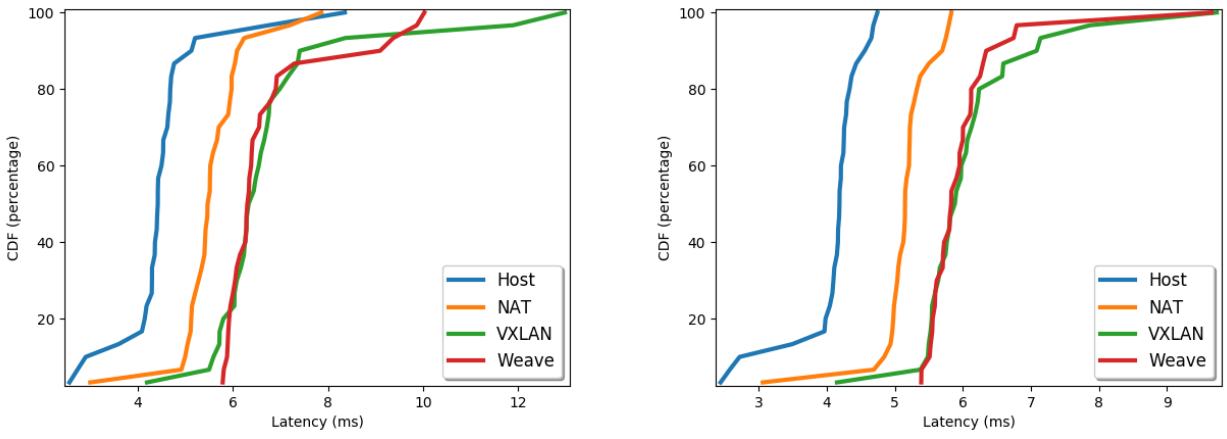
(b) Latency

Figure 5.4: Memcached throughput and latency with *memcache_text* protocol



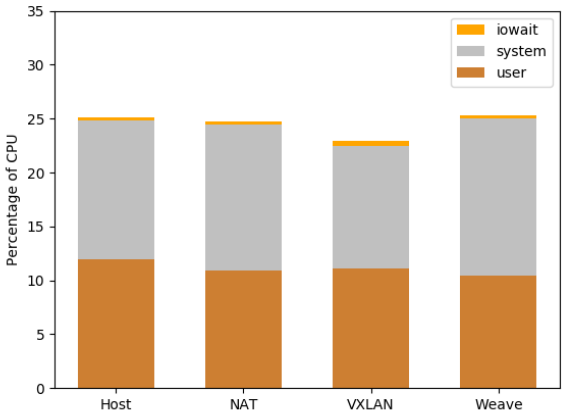
(a) Throughput (b) Latency

Figure 5.5: Memcached throughput and latency boxplot with *memcache_text* protocol

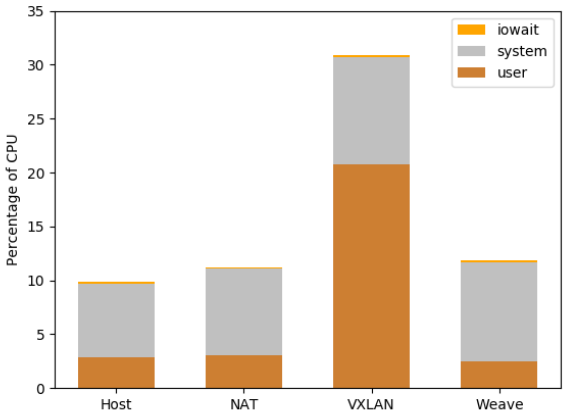


(a) SET (b) GET

Figure 5.6: *Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. The two overlay network lines overlap.*



(a) Client



(b) Server

Figure 5.7: CPU utilization of memcached client and server

Moreover, for the *memcache_binary* protocol, table 5.7 shows the average throughput in Kbps and their standard deviation while table 5.8 shows the latency for responding to the memcached command in millisecond and the standard deviation for all the modes. As usual host mode recorded the highest throughput followed by NAT which dropped by 17% and the two overlay networks have the least throughput where VXLAN dropped by 30% and weave dropped by 29%. Also in the latency of the memcached server response, the order remain the same. NAT had an increase of of 20% when compared with host mode because in case of latency the lower the better. VXLAN increased by 42% while weave increased by 40%. Figure 5.8 shows the bar plots of the throughput and latency with error bars denoting standard deviation. On the other hand, figure 5.9 shows the corresponding boxplots, the results did not get much variability and there are few outliers. Table 5.9 shows the mean and standard deviation of SET operation latency also in millisecond while Table 5.10 shows the latency of GET operation. The distribution of the latency for Memcached SET and GET operations is shown in figure 5.10 where the two overlay network lines overlapped in both cases which means the performance of the overlay networks is almost the same. Figure 5.11 shows the CPU utilization of the client and server. On the server part, there is no much consumption of the CPU resource while in the client that is the benchmark part, the consumption is a bit high. There is no much difference between the CPU consumption of the overlay networks and the rest of the modes.

Modes	**Throughput in Kbps**	
	Average	Standard deviation
Host mode	15168.813	2214.971
NAT	12488.656	1532.746
VXLAN	10564.314	1134.282
Weave	10699.154	962.507

Table 5.7: Memcached throughput with *memcache_binary* protocol

Modes	**Latency in msec**	
	Average	Standard deviation
Host mode	4.471	0.478
NAT	5.390	0.493
VXLAN	6.365	0.572
Weave	6.268	0.510

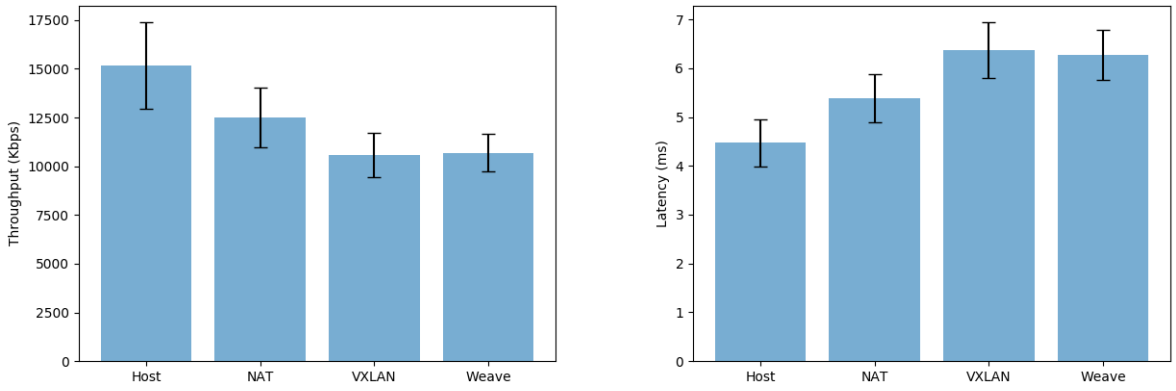
Table 5.8: Latency for responding to Memcached command with *memcache_binary* protocol

Modes	**SET latency in msec**	
	Average	Standard deviation
Host mode	4.983	0.972
NAT	5.691	0.546
VXLAN	6.812	1.044
Weave	6.723	0.469

Table 5.9: Latency of Memcached SET operation with *memcache_binary* protocol

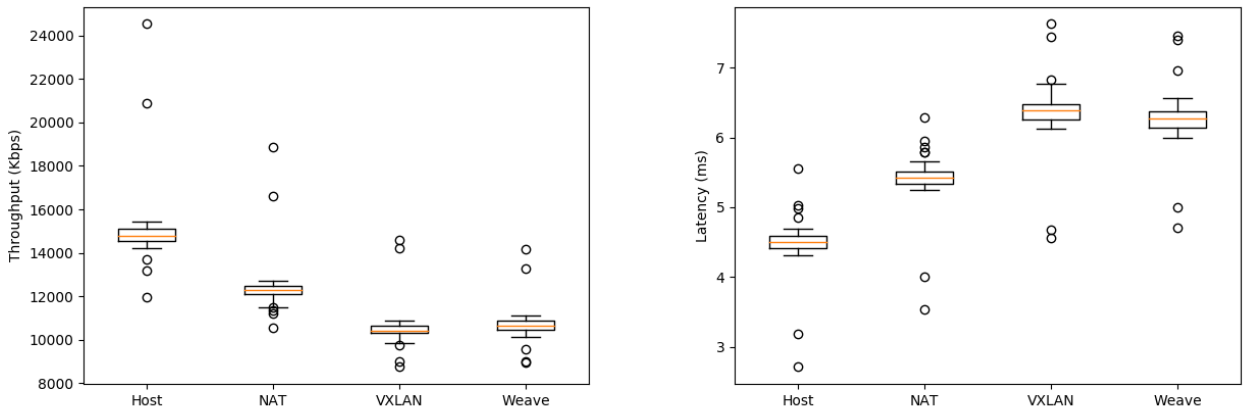
Modes	**GET latency in msec**	
	Average	Standard deviation
Host mode	4.418	0.486
NAT	5.359	0.495
VXLAN	6.319	0.547
Weave	6.221	0.538

Table 5.10: Latency of Memcached GET operation with *memcache_binary* protocol



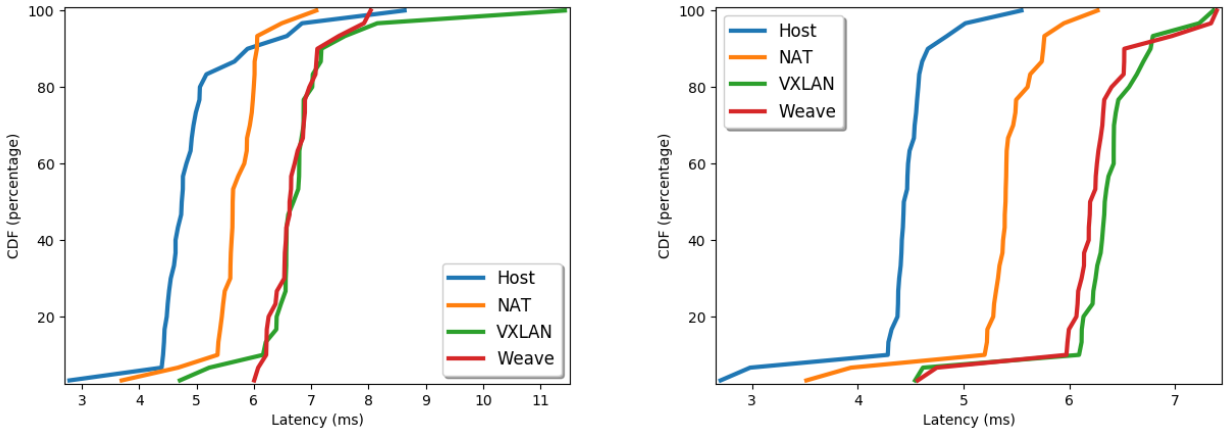
(a) Throughput (b) Latency

Figure 5.8: Memcached throughput and latency with *memcache_binary* protocol



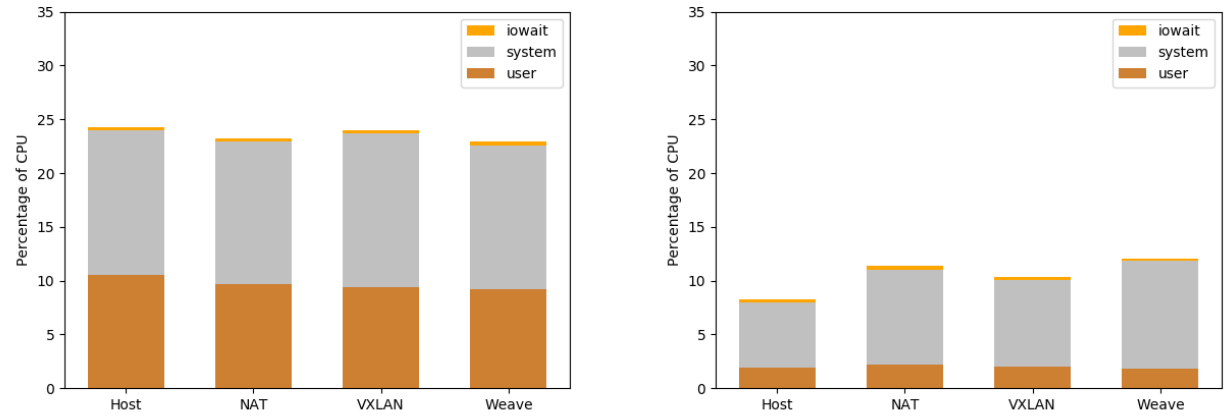
(a) Throughput (b) Latency

Figure 5.9: Memcached throughput and latency boxplot with *memcache_binary* protocol



(a) SET (b) GET

Figure 5.10: *Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. The two overlay network lines overlap.*



(a) Client (b) Server

Figure 5.11: CPU utilization of memcached client and server with *memcache_binary* protocol

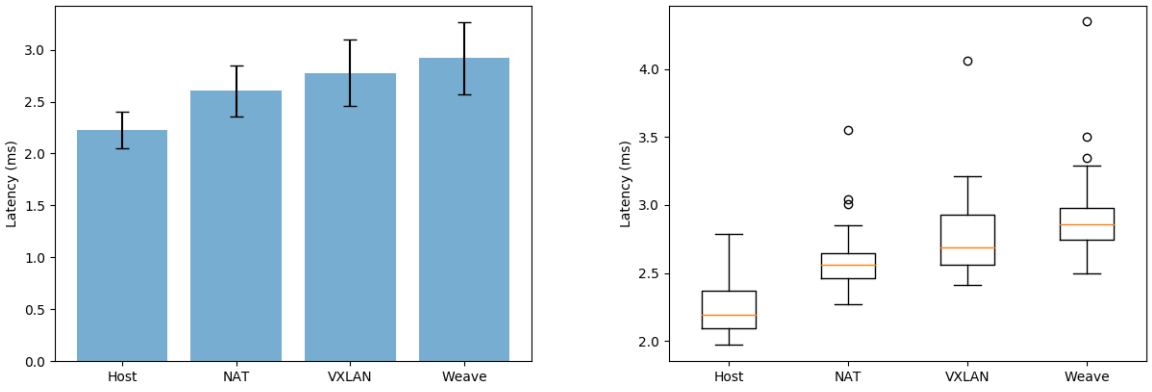
5.3 Nginx

1KB HTML file: A total of thirty (for each of the four modes) samples were collected by setting the throughput to 3K requests/second. Table 5.11 shows the average of the results and their standard deviation. Figure 5.12 shows the bar plot (with the error bar

denoting the standard deviation) and the boxplot showing the level of variability of the results. Host mode recorded the least latency followed by NAT with an increase of 16% then weave and VXLAN with an increase of 24% and 30% respectively. Figure 5.13 shows the CPU utilization of the client (*wrk*) and server (nginx) where they both have almost the same CPU consumption in which kernel part is more than the user part and I/O is very low in all of the modes.

Modes	**Latency in msec**	
	Average	Standard deviation
Host mode	2.228	0.176
NAT	2.603	0.247
VXLAN	2.776	0.320
Weave	2.918	0.344

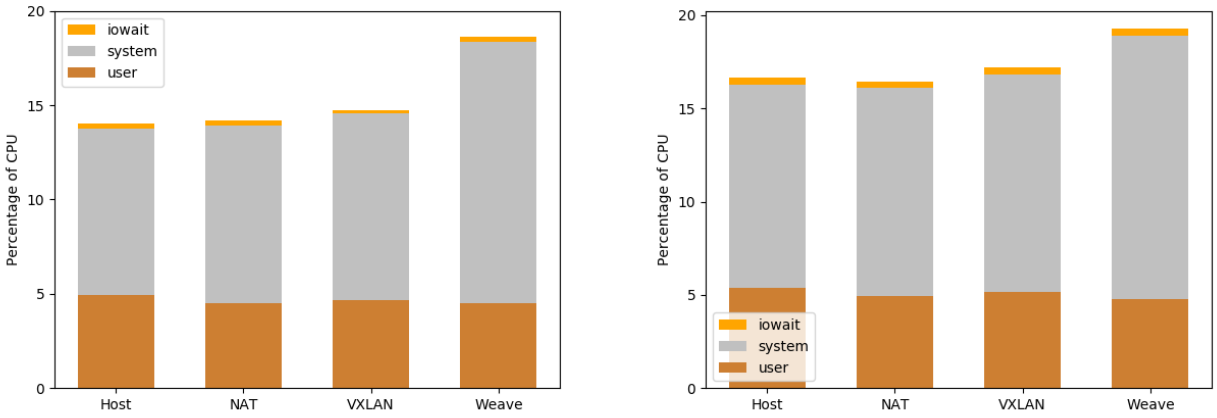
Table 5.11: Nginx 1KB html file latency



(a) Latency bar plot

(b) Latency boxplot

Figure 5.12: 3K reqs/sec Nginx 1KB latency



(a) Client

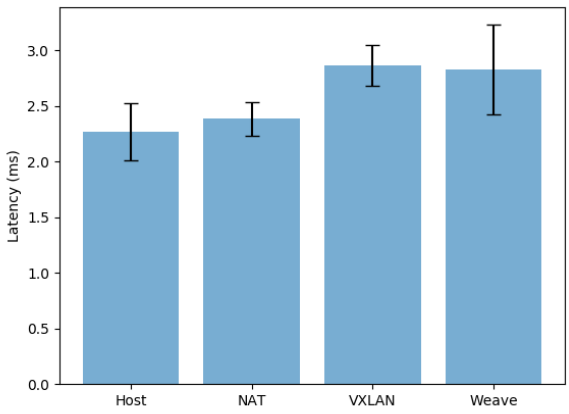
(b) Server

Figure 5.13: CPU utilization of Nginx client and server in 1KB file

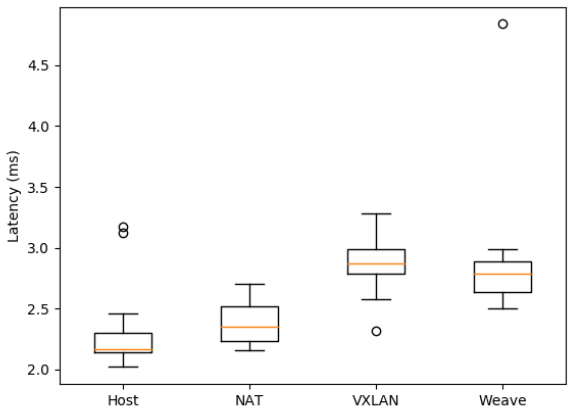
1MB HTML file: thirty samples were collected by setting the throughput to 3K requests/second for each experiment on the four modes. Table 5.12 shows the average of the results and their standard deviation. Figure 5.14 (a) shows the bar plot (with the error bar denoting the standard deviation) and the boxplot (b) showing the level of variability of the results. Host mode has the least latency followed by NAT with an increase of 5% then weave and VXLAN with an increase of 26% and 24% respectively. Figure 5.15 shows the CPU utilization of the client (*wrk2* benchmark) and server (nginx), similar to 1KB file, they almost have the CPU consumption in all the four modes.

Modes	**Latency in msec**	
	Average	Standard deviation
Host mode	2.266	0.258
NAT	2.385	0.150
VXLAN	2.864	0.184
Weave	2.827	0.400

Table 5.12: 3K reqs/sec Nginx 1MB html file latency

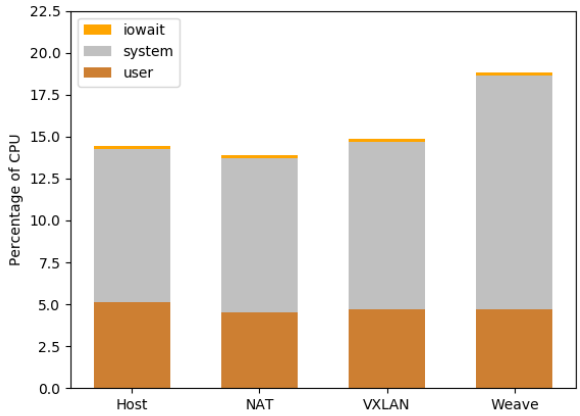


(a) Latency bar plot

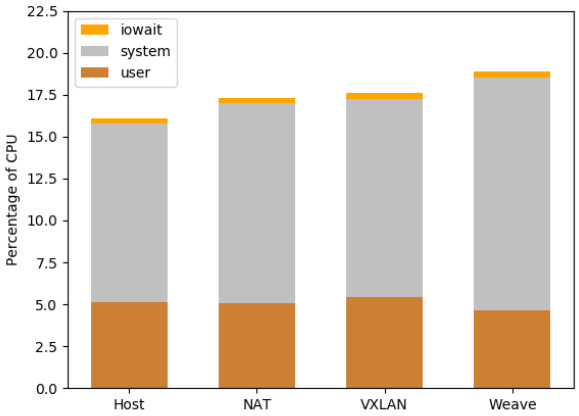


(b) Latency boxplot

Figure 5.14: 3K reqs/sec Nginx 1MB file latency



(a) Client



(b) Server

Figure 5.15: CPU utilization of Nginx client and server in 1MB file

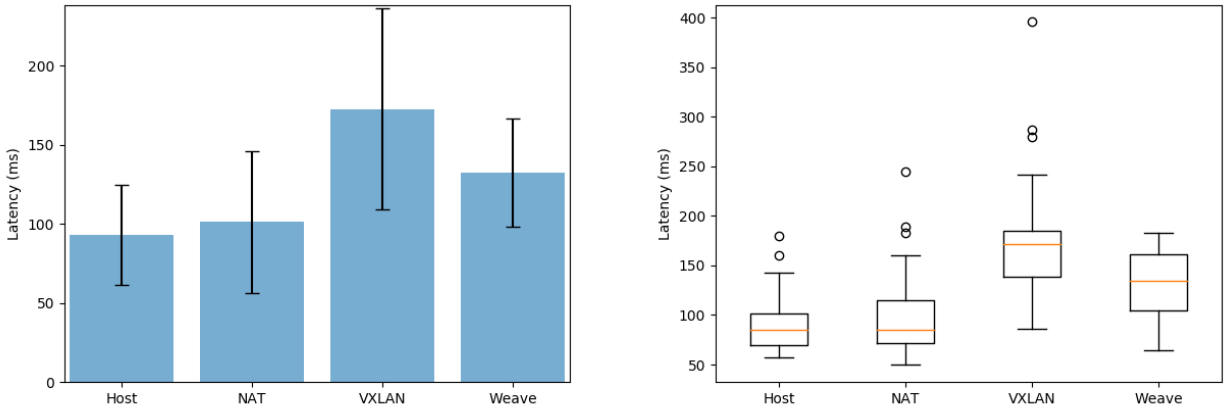
For the latency of the two html files (1KB and 1MB), we expect to see much difference in the results with higher values on the 1MB. We tried changing the configuration of the nginx server and some other options but we keep obtaining the same results. Nevertheless, we intended to do more work on this part in our future work.

5.4 PostgreSQL

We collected thirty samples of the results by executing the scripts of our testbed on each of the four modes. We carried out the experiments by setting the benchmark i.e *pgbench* to generate 300 transactions per second and then later stress more the system by generating 500 transactions per second. For the first case, table 5.13 shows the average of the results and their standard deviation. Figure 5.16 (a) shows the bar plot (with the error bar denoting the standard deviation) and the boxplot (b) showing the level of variability of the results. Host mode recorded the least latency followed by NAT (which increased by 8.5%) then weave with an increase of 42% and VXLAN achieved the highest latency with an increase of 85%. Figure 5.17 shows the CPU utilization of the client and server where the client which is the benchmark (*pgbench*) has more consumption in the user than the kernel part and a very low I/O. On the other hand, the server has kernel part almost same as that of the user and I/O which is big compared to other applications. This is because postgreSQL has more disk operations than the rest of the applications as the data is stored on disk.

Modes	**Latency in msec**	
	Average	Standard deviation
Host mode	93.197	31.443
NAT	101.206	45.058
VXLAN	172.694	63.472
Weave	132.604	34.330

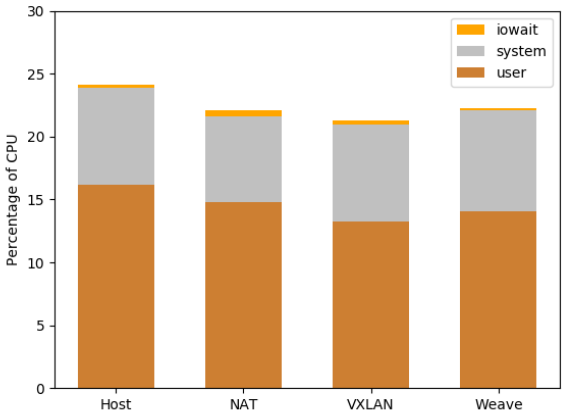
Table 5.13: PostgreSQL latency on 300 trans/sec



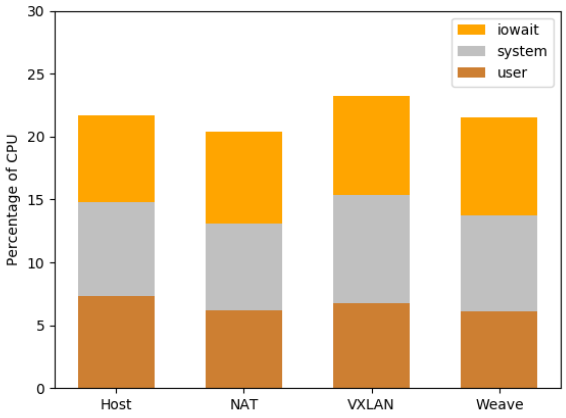
(a) Latency bar plot

(b) Latency boxplot

Figure 5.16: PostgreSQL latency on 300 trans/sec



(a) Client



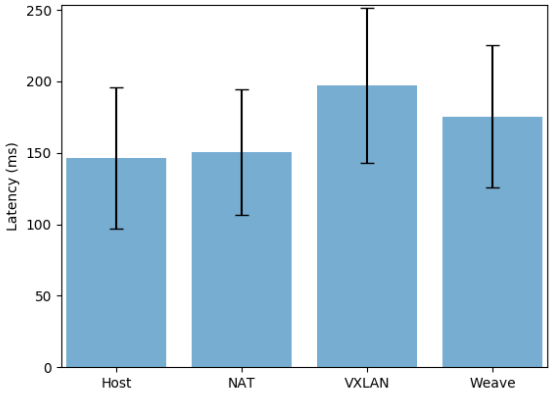
(b) Server

Figure 5.17: CPU utilization of postgresql client and server

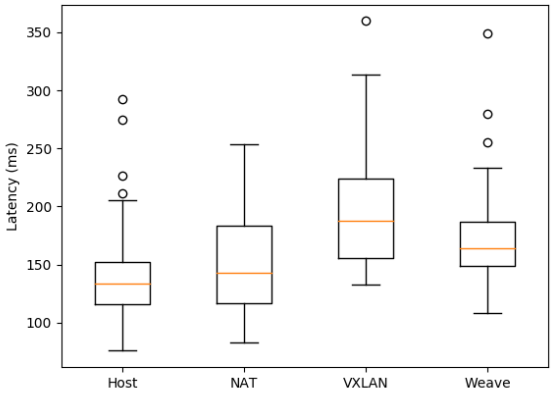
Moreover, for the 500 transactions per second, the average and standard deviation of the PostgreSQL latency for each of the four modes are shown in table 5.14. Similar to the previous cases, host has the best latency. It was followed by NAT with an increase of about 3%. VXLAN increased by 34% while weave increased by 20%. Figure 5.18 shows the bar plots and the corresponding boxplots where NAT and VXLAN have variability a bit more than host and weave whose variability are closed to the 300 transactions results. The CPU utilization of the client and server is shown in figure 5.19 where the benchmark (client) has less utilization of the CPU (especially in the two overlay networks) than in 300 transactions case. On the server side, the utilization is almost the same as that of the 300 transactions scenario. Stressing the system does not have much impact on the CPU utilization. It has more impact on the performance of the network.

Modes	**Latency in msec**	
	Average	Standard deviation
Host mode	146.454	49.352
NAT	150.503	43.897
VXLAN	197.007	54.227
Weave	175.468	49.913

Table 5.14: PostgreSQL latency on 500 trans/sec

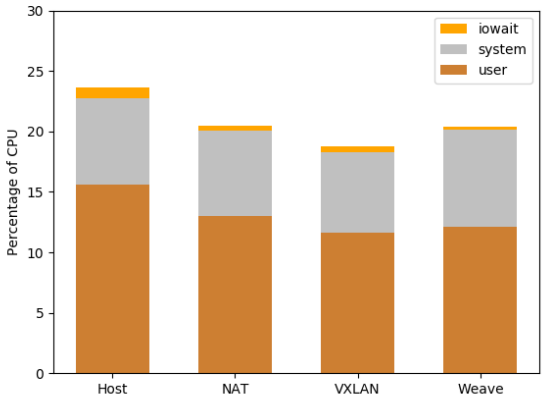


(a) Latency bar plot

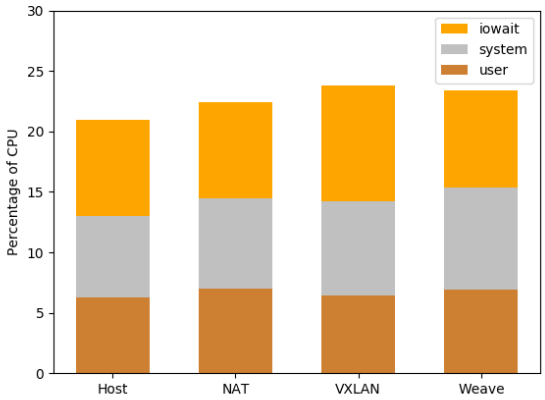


(b) Latency boxplot

Figure 5.18: PostgreSQL latency on 500 trans/sec



(a) Client



(b) Server

Figure 5.19: CPU utilization of postgresql client and server

Chapter 6

Reducing the overhead of the overlay Networks using OS/Hardware support

6.1 Receive Side Scaling/Receive Packet Steering

To increase parallelism and improve performance, there are a number of complementary techniques in the Linux networking stack which can be used in multi-processor systems. These techniques are: RSS (Receive Side Scaling), RPS (Receive Packet Steering), RFS (Receive Flow Steering), Accelerated Receive Flow Steering and XPS (Transmit Packet Steering). In this work, we used RPS to improve the performance of the two overlay networks (Docker default overlay and weave) due to their performance overhead as a result of the packet encapsulation. The description of each of these techniques is mostly inspired from [30] which is an entry point for the kernel support of these techniques.

RSS: Receive Side Scaling: Modern Network Interface Cards (NICs) do support multiple receive and forward descriptor queues. Upon reception, a NIC can forward different packets to different queues in order to distribute processing to various CPUs in the system. The NIC applies a filter to every packet which assigns it to one of a small number of logical flows when distributing the packets. Packets for every flow are routed to distinct receive queue, consequently, each flow can be processed by different CPU. This technique is known as Receive Side Scaling (RSS). The filter used in this technique is a hash function over the network and/or transport layer headers. The popular hardware implementation of RSS uses 128-entry indirection table in which every entry stores a queue number. Some modern NICs allow routing of packets to queues using programmable filters [30]. Figure 6.1 shows how RSS uses hash function in distributing the packets to the CPUs.

RPS: Receive Packet Steering: Is a logical software implementation of RSS. Since it is in software, it is called later along the data-path. Though RSS picks the queue and consequently the CPU that is going to run the hardware interrupt handler, RPS chooses

the CPU to do protocol processing above the interrupt handler. This is achieved by allocating the packet on the chosen CPU's backlog queue and getting up the CPU for processing. RPS is called throughout the bottom half of the receive interrupt handler at the same time a driver forwards a packet up the network stack with `netif_rx()` or `netif_receive_skb()`. These make a call to `get_rps_cpu()` function which makes a selection of the queue that will process a packet [30].

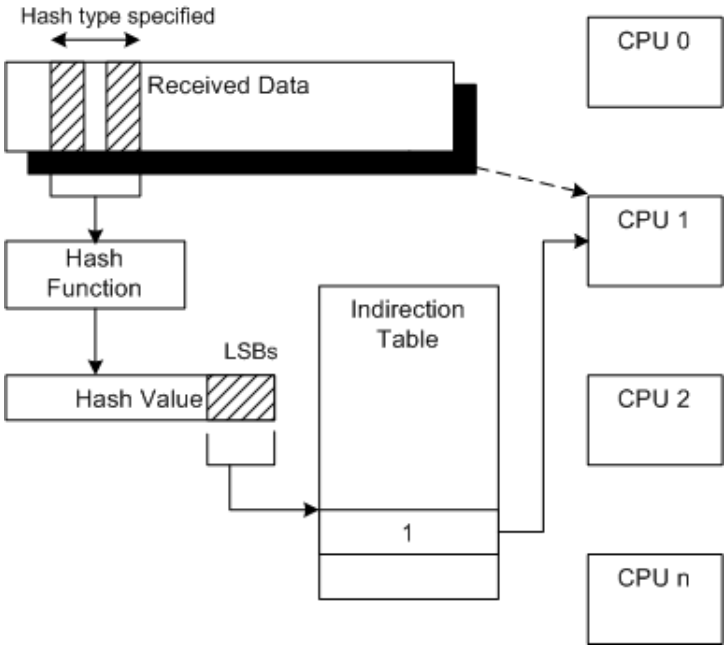


Figure 6.1: RSS mechanism for determining a CPU

Source: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling

6.2 Preliminary Results of the reduced overhead of some applications

As we have seen in chapter 5 of this report where the experimental results of the three applications and iperf3 were presented, host mode recorded the best performance among the four modes of Docker networking in all the applications. It was followed by NAT which had a few percentage of performance drop compared to host mode. The two overlay networks (Docker default overlay and weave) have the worse performance in all scenarios. Nevertheless, overlay networks have brought solutions to some of the problems faced when host and NAT modes are used for connecting containers in the cloud. Some of these problems are: port contention, scalability problem and so on. It is thus of great importance to improve their performance. We used RPS in achieving this objective by taking advantage of the nowadays multi-processor systems by balancing the loads to the

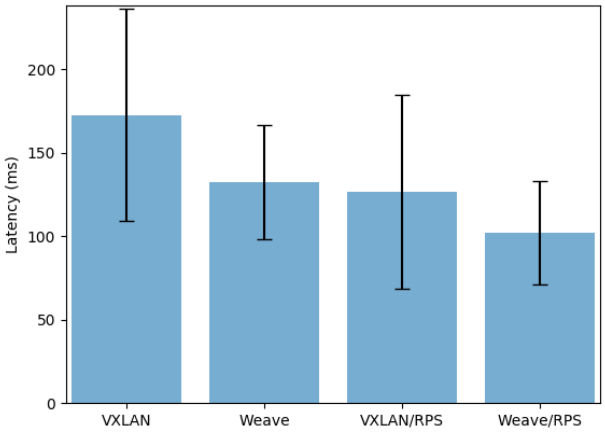
available CPUs. We focused on one of the three applications which is PostgreSQL in order to reduce the overhead on its performance. We reduced the latency of this application by activating the RPS technique on the NICs of our system when carrying out the experiments.

We carried out the experiments with 300 transactions per second when making a request to the PostgreSQL server. Table 6.1 shows the average and standard deviation of the thirty samples collected. Docker default overlay (VXLAN) had a performance gain of 26.6% and weave had 23% in the latency of the communication between the server and the benchmark. Figure 6.2 (a) can be used to visualize this result where the bars denote the average and error bars denote the standard deviation while (b) shows the corresponding boxplot where the results did not have much variability. This is a significant improvement in the performance in which weave had almost same performance with NAT. Moreover, Figure 6.3 shows the CPU utilization of the client (pgbench benchmark) and server (PostgreSQL server). The server consumed a bit more of CPU resource compared to non RPS setting especially in VXLAN. This can be due to involvement of more than one CPU in the process.

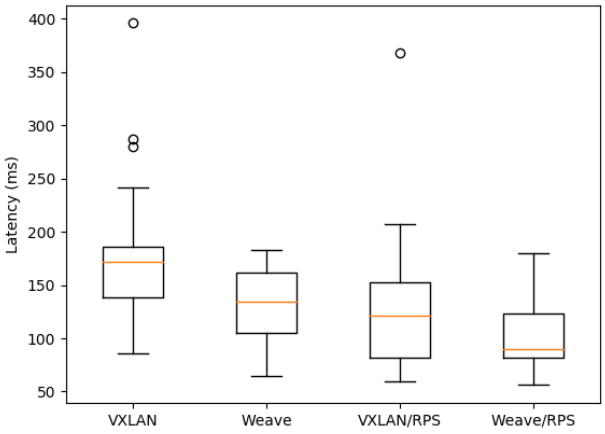
To stress more the system, we also carried out similar experiment with 500 transactions per second. In this case, we were able to improved the performance of VXLAN by 11.4% and weave by 22.8%. Table 6.2 shows the average and standard deviation of the thirty samples collected. Figure 6.4 (a) can be used to visualize the result with the bars representing the average and errors bar denoting the standard deviation while (b) display the corresponding boxplot. Figure 6.5 shows the CPU utilization of the client and server in which VXLAN consumed a bit more CPU than in non RPS setting while weave consumed less on the server side. On the client side, both of the two overlay networks consumed a bit more CPU than on non RPS and this could be due to the fact that more CPUs were involved in the process.

Modes	**Latency in msec**	
	Average	Standard deviation
VXLAN	172.694	63.472
Weave	132.604	34.330
VXLAN/RPS	*126.706*	*58.138*
Weave/RPS	*102.039*	*31.020*

Table 6.1: PostgreSQL latency on 300 trans/sec with RPS

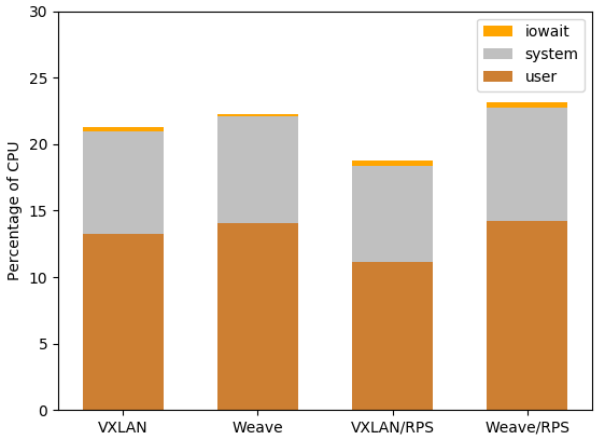


(a) Latency bar plot

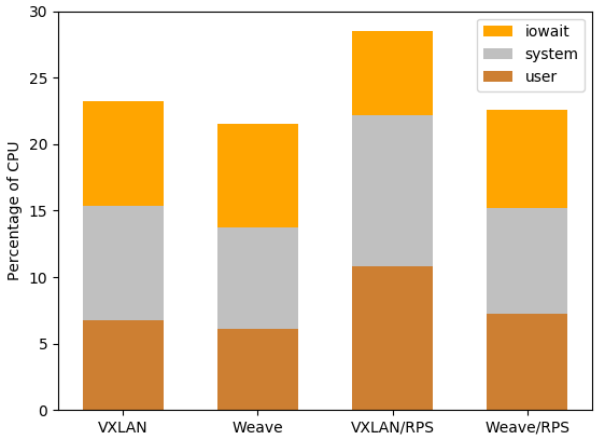


(b) Latency boxplot

Figure 6.2: PostgreSQL latency on 300 trans/sec with RPS



(a) Client

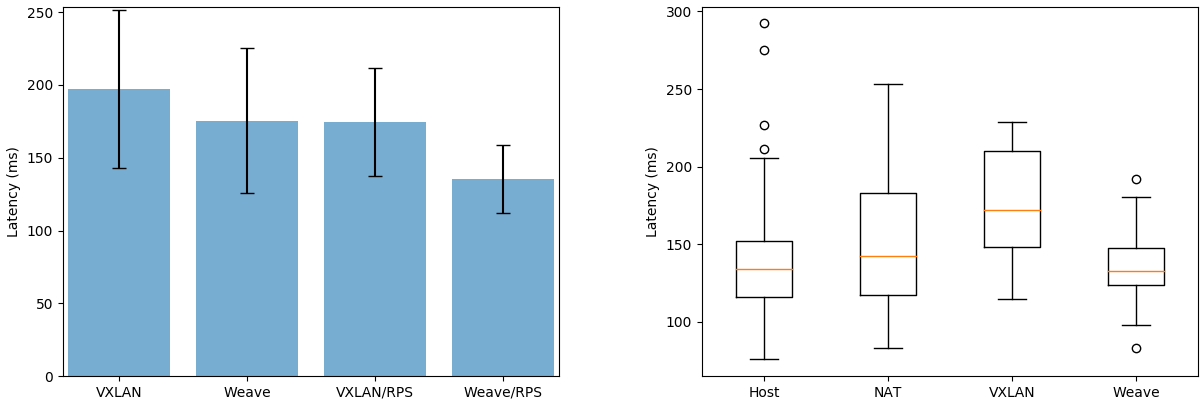


(b) Server

Figure 6.3: CPU utilization of postgresql client and server

Modes	**Latency in msec**	
	Average	Standard deviation
VXLAN	197.007	54.227
Weave	175.468	49.913
VXLAN/RPS	*174.521*	*37.031*
Weave/RPS	*135.42*	*23.353*

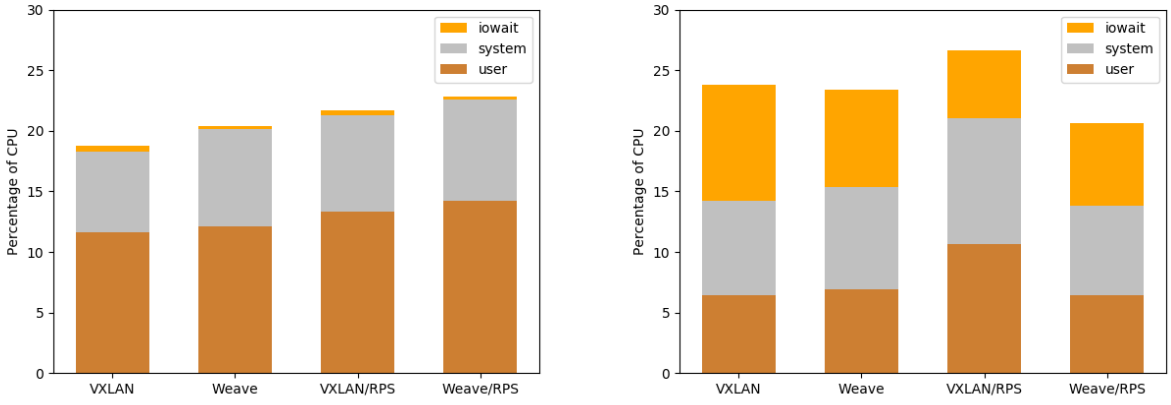
Table 6.2: PostgreSQL latency on 500 trans/sec with RPS



(a) Latency bar plot

(b) Latency boxplot

Figure 6.4: PostgreSQL latency on 500 trans/sec with RPS



(a) Client

(b) Server

Figure 6.5: CPU utilization of postgresql client and server

Chapter 7

Conclusion

In this work, we conducted a comprehensive empirical study of various container networks. We have confirmed what was observed in the previous work, that host mode has the best performance followed by NAT by deploying some cloud applications into Docker containers. We have also demonstrated that the two overlay networks used in this work have some performance overhead when compared with the first two modes. However, these two overlay networks have solved problems like port contention and scalability problem in connecting containers in the cloud. Because of this great advantage we thought of reducing this overhead as a very important research topic. We focused towards this target and we were able to reduced the overhead of these networks using some techniques in Linux networking stack by testing with some of the applications. One thing unique about our work is that, we have reduced the overhead of two overlay networks by adding one more that is Docker default overlay which uses VXLAN tunnel. This overlay network was not studied in the previous work [1]. Our work seems to be the first of its kind in reducing the overhead of this network. We have also studied NAT which was not analyzed by the authors of the previous work. These findings can help user select the right network for their workloads and serve as a guide in optimizing the existing container networks. Finally, we have the intention to further our work by doing the same thing to some other overlay networks and deploying more cloud applications. The shell scripts of our testbed is open sourced in a github repository in the link: https://github.com/Yusuf-Haruna/Docker-Cloud-Networking-M.Sc.-project.

Bibliography

[1] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim:{OS} kernel support for a low-overhead container overlay network," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 331–344.

[2] Wikipedia contributors, "Virtualization." [Online]. Available: https://en.wikipedia.org/wiki/Virtualization

[3] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 189–197.

[4] "Docker." [Online]. Available: https://www.docker.com/

[5] J. Turnbull, "The docker book," 2016.

[6] "Sparkyfish." [Online]. Available: https://github.com/chrissnell/sparkyfish

[7] "Sockperf." [Online]. Available: https://github.com/Mellanox/sockperf

[8] "iperf." [Online]. Available: https://iperf.fr/

[9] P. Killelea, *Web Performance Tuning: speeding up the web*. " O'Reilly Media, Inc.", 2002.

[10] "sar(1) - linux man page." [Online]. Available: https://linux.die.net/man/1/sar

[11] "Opsdash." [Online]. Available: https://www.opsdash.com/

[12] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 233–240.

[13] K. Lee, Y. Kim, and C. Yoo, "The impact of container virtualization on network performance of iot devices," *Mobile Information Systems*, vol. 2018, 2018.

[14] Weavework, "Weave." [Online]. Available: https://github.com/weaveworks/weave

[15] "Flannel." [Online]. Available: https://github.com/coreos/flannel/

[16] "Calico." [Online]. Available: https://github.com/projectcalico/calicoctl

[17] Wikipedia contributors, "Network address translation." [Online]. Available: https://en.wikipedia.org/wiki/Network_address_translation

[18] ——, "iptables." [Online]. Available: https://en.wikipedia.org/wiki/Iptables

[19] G. Urvoy-Keller, "Docker networking with linux," January 2019.

[20] Wikipedia contributors, "Virtual extensible lan." [Online]. Available: https://en.wikipedia.org/wiki/Virtual_Extensible_LAN

[21] "Docker swarm." [Online]. Available: https://docs.docker.com/engine/swarm/

[22] "Memcached." [Online]. Available: https://memcached.org/

[23] "memtier benchmark." [Online]. Available: https://github.com/RedisLabs/memtier_benchmark

[24] "Nginx." [Online]. Available: https://nginx.org/en/

[25] "Nginx." [Online]. Available: https://www.nginx.com/

[26] Wikipedia contributors, "Nginx." [Online]. Available: https://en.wikipedia.org/wiki/Nginx

[27] "wrk." [Online]. Available: https://github.com/giltene/wrk2

[28] "Postgresql." [Online]. Available: https://www.postgresql.org/

[29] "pgbench." [Online]. Available: https://www.postgresql.org/docs/9.5/pgbench.html

[30] "Scaling in the linux networking stack." [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

[31] N. Poulton, "Demystifying docker overlay networking." [Online]. Available: http://blog.nigelpoulton.com/demystifying-docker-overlay-networking/

[32] S. Makam, "Docker networking weave." [Online]. Available: https://sreeninet.wordpress.com/2015/01/18/docker-networking-weave/

[33] M. Gupta, "Container networking easy button for app teams; heartburn for networking and security teams." [Online]. Available: https://www.onug.net/blog/container-networking-easy-button-for-app-teams-heartburn-for-networking-and-security-teams/

[34] Datapacket, "10gbps network bandwidth test iperf tutorial." [Online]. Available: https://datapacket.com/blog/10gbps-network-bandwidth-test-iperf-tutorial/

[35] J. Danjou, "Python + memcached: Efficient caching in distributed applications." [Online]. Available: https://julien.danjou.info/python-memcached-efficient-caching-in-distributed-applications/

[36] C. Limpalair, "Nginx + php-fpm installation and configuration." [Online]. Available: https://scaleyourcode.com/blog/article/2

[37] H.-J. Schonig and Z. Boszormenyi, *PostgreSQL Replication*. Packt Publishing, 2015.

[38] P. Singh, "Configure tiny core linux as nat (p-nat) router using iptables." [Online]. Available: https://iotbytes.wordpress.com/configure-microcore-tiny-linux-as-nat-p-nat-router-using-iptables/

[39] "Microsoft." [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling

[40] "Weaveworks." [Online]. Available: https://www.weave.works/docs/net/latest/overview/