Advances in Networks
2022; 10(2): 15-30
http://www.sciencepublishinggroup.com/j/net
doi: 10.11648/j.net.20221002.11
ISSN: 2326-9766 (Print); ISSN: 2326-9782 (Online)

Analysis of Docker Networking and Optimizing the Overhead of Docker Overlay Networks Using OS Kernel Support

Yusuf Haruna*, Abdulmalik Ahmad Lawan, Kamaluddeen Ibrahim Yarima, Muhammad Mahmoud Ahmad, Mustapha Abdulkadir Sani

Department of Computer Science, Kano University of Science and Technology, Wudil, Nigeria

Email address:

yusuf.haruna@kustwudil.edu.ng (Yusuf Haruna), aalawan@kustwudil.edu.ng (Abdulmalik Ahmad Lawan), kiyarima@kustwudil.edu.ng (Kamaluddeen Ibrahim Yarima), mmahmoud@kustwudil.edu.ng (Muhammad Mahmoud Ahmad), mustapha.abdulkadir@kustwudil.edu.ng (Mustapha Abdulkadir Sani)

*Corresponding author

To cite this article:

Yusuf Haruna, Abdulmalik Ahmad Lawan, Kamaluddeen Ibrahim Yarima, Muhammad Mahmoud Ahmad, Mustapha Abdulkadir Sani. Analysis of Docker Networking and Optimizing the Overhead of Docker Overlay Networks Using OS Kernel Support. *Advances in Networks*. Vol. 10, No. 2, 2022, pp. 15-30. doi: 10.11648/j.net.20221002.11

Received: April 8, 2022; **Accepted**: May 14, 2022; **Published**: October 17, 2022

Abstract: The superior performance of lightweight virtualization with containers over traditional virtualization enables the implementation of scalable systems and multi-tier/distributed networks. Containers supports the creation of dedicated network overlays, spanning over several virtual machines (VMs) or physical hosts to interconnect application fragments. Hence, there is a need to understand the comparative performance of various interconnection solutions in terms of needed resources (CPU, RAM, and networking). In this work, we use a variety of applications to benchmark the performance of different container interconnection solutions. Accordingly, we experimented with four applications namely Memcached, Nginx, PostgreSQL, and *iperf3*. 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. The experimental results revealed superior performance in host mode, followed by NAT and the overlay networks (VXLAN and weave) which have the least performance due to packet encapsulation. In each case, *sar* was used to monitor the CPU utilization. We were able to reduce the overhead of the two overlay networks using RPS (Receive Packet Steering) technique because they brought solutions to some of the problems faced when connecting containers using host and NAT modes in the cloud.

Keywords: Virtualization, Container, Virtual Machine, Network

1. Introduction

Virtualization technology is a computing breakthrough that enables the deployment of applications on virtual (rather than physical) hardware resources [1]. Several approaches were proposed in optimizing the performance of this technology when deployed as a full-fledged Operating System (traditional virtual machines) or container-based virtualization. Exemplary studies focuses on addressing how many virtual machines (VMs) can be consolidated on one physical machine, optimizing VMs startup time, and networking multiple VMs among others [2]. Particularly, container-based virtualization provides lightweight virtualization environment, 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. Unlike the traditional VMs, containers enables

partitioning of hardware resources to users in order to speedup deployment of applications [2]. The most popular containerization solutions currently in the market is Docker [3, 4].

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 [5]. 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.

The present study aimed at analyzing the performance of Docker container networking in single Virtual Machine (VM) and multiple VM scenarios. Accordingly, a number of benchmarks including *Sparkyfish* [6], *Sockperf* [7] and *iperf3* [8] will be used in carrying out the analysis. Furthermore, to understand the performance of different Docker Networking solutions, we will experiment on four distinct modes of connecting containers in the cloud (host, NAT [9], Docker default overlay and weave). We will build a realistic testbed by selecting three popular cloud applications and deploying them into Docker containers to benchmark their performance in the four modes and optimize the overhead of the overlay Networks using OS/hardware support.

2. Literature Review

Docker is an open-source software produced by a team of researchers at Docker Inc. It allows automation of applications deployment into containers and it was designed in such a way that application deployment engine is added on top of a virtualized environment that allows execution of containers [3, 4]. Figure 1 shows a comparison between traditional virtualization and lightweight virtualization using Docker. Traditional virtualization 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 allows running of applications in containers. The containers share the kernel and other files of the same OS.

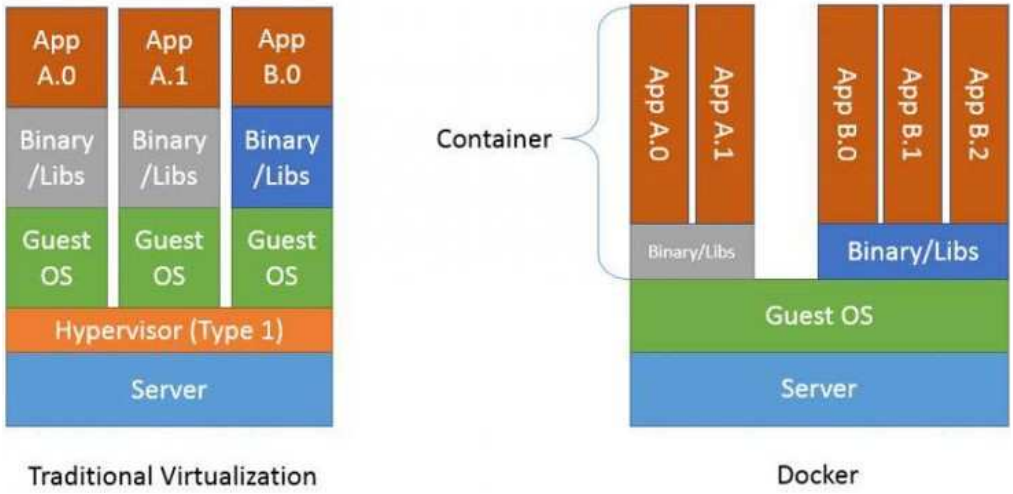

Figure 1. *Traditional Virtualization Vs Lightweight Virtualization (Source: https://www.docker.com/).*

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. Several 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 [2]. Indeed, in many deployment scenarios, containers are deployed within VM and not directly on the host OS because containers do share the host kernel and any breach in the containerization engine might compromise the whole machine.

Several studies were carried out to understand how container-based virtualization works. However, most of the studies did not approached the problem from networking point of view. For instance, Xavier et al. [10] conducted a comparative evaluation on a high performance computing (HPC) environments based on Memory Performance, Disk Performance, Network Performance (on a very narrow perspective), and Performance Overhead. Similarly, Lee et al. [11] studied the impact of container virtualization on network performance with restriction on IoT devices. Suo et al. [2] studied and analyzed the performance of Docker container networking in two scenarios; both containers are on a single VM and on multiple VMs. The first experimental scenario was carried out 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 experimental scenario was carried out using host mode, NAT and overlay networks (Docker default overlay, weave [12], flannel [13] and calico [14]. Simple Benchmarks (bulk transfer tools similar to iperf) that perform active test were also used. The comparative

findings of the study are crucial when deciding which mode to use in networking containers. The study findings revealed that the overlay networks used in the multiple VM case have some significant overhead on the performance of the network due to packet encapsulation. 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. The comparative findings of the study are crucial when deciding which mode to use in networking containers. However, the study failed to highlight ways to reduce the overhead of the overlay networks despite their importance, and was conducted without deploying some applications into the containers when carrying out the experiments. Accordingly, a recent study by Zhuo et al. [15] used OS kernel support to reduce the overhead of one of the overlay networks (weave) and experimented with some applications deployed into containers. The study utilized only *Sparkyfish* and *Sockperf* benchmarks in the analysis process.

In the present study, we extended the work of Zhuo et al. [15] by considering the case of containers running inside VMs, deploying some popular cloud applications to understand the performance of Docker networking modes, and involving an additional networking mode based on Network Address Translation (NAT). Moreover, apart from the weave overlay network, we built a testbed and further experimented with Docker default overlay network which uses virtual extensible LAN (VXLAN) [16] tunnel in connecting containers. Furthermore, we evaluate the experimental performance of the four distinct modes (Host, NAT, Docker default overlay (VXLAN) and weave) by deploying applications like Memcached [17, 18], Nginx [19, 20] and PostgreSQL [21, 22, 23] together with their benchmarks. The experimentation was carried out with iperf3 and we approached overhead optimization using RPS (Receive Packet Steering) by testing on PostgreSQL.

3. Method

3.1. Research Motivation

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 uses 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 resources) has 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.

3.2. Research Procedure

We tested the performance of Docker container networking in single Virtual Machine (VM) and multiple VM scenarios. We used three distinct benchmarks in carrying out the analysis including *Sparkyfish*, *Sockperf* as used in [15] and we extended with *iperf3*. Therefore, in the present study we adapted and extended the research procedure reported in [15] as follows:

1) We evaluated the performance of different Docker Networking solutions based on four modes (host, NAT, Docker default overlay and weave) of connecting containers in the cloud.
2) We built a realistic testbed by selecting three popular cloud applications (and their benchmarks) and deploying them into Docker containers to benchmark their performance in the four modes. We also performed similar analysis with *iperf3, which* performs an active measurement.
3) We obtained comparative results by testing our testbed and monitor system level performance with sar (System Activity Reporting) [24] which is a Unix System V-derived system monitor command.
4) We also utilized statistical tools like boxplot and standard deviation in understanding the level of variability of the results (thirty samples in each case) for statistical significance.
5) We approached overhead optimization of the overlay networks using OS/hardware support.

3.3. Experimental Settings

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. 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.

For each test, a container was created using Docker [3] 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 2. The two VMs were connected using one of the four modes (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.

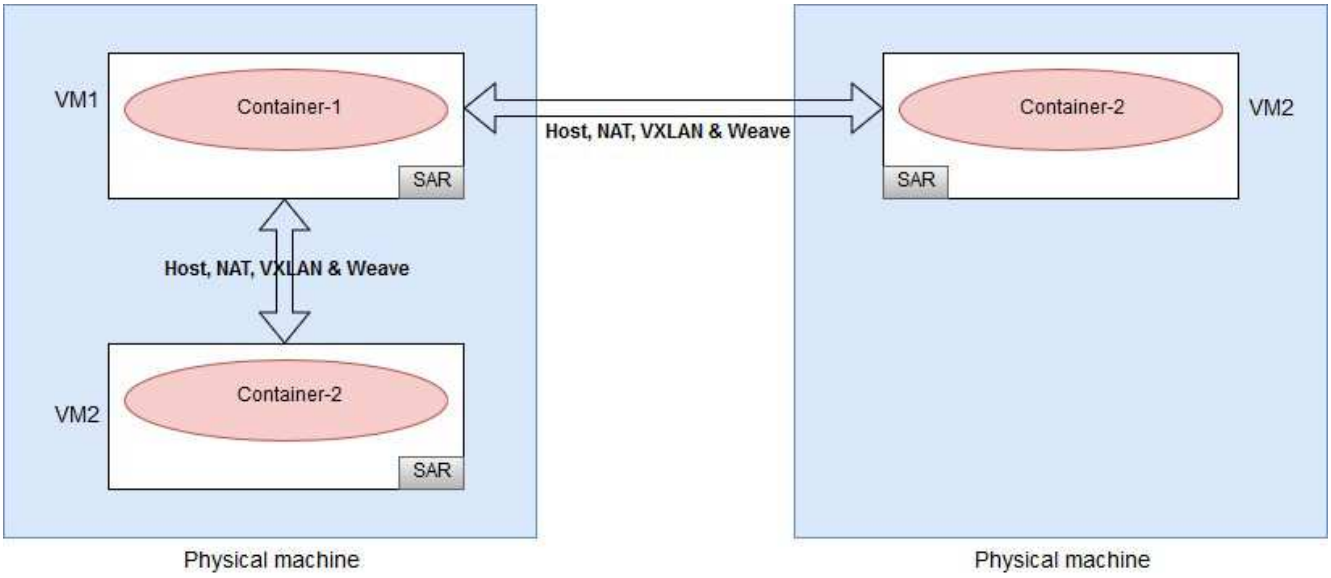

Figure 2. *Virtual Machines configuration in a physical machine.*

4. Results and Discussion

4.1. Iperf3

We collected a total of thirty (for each mode) samples of the result by running the shell scripts of our testbed. Consequently, a container was created with iperf3 installed and the server was started in one VM followed by 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 1 shows the average and standard deviation of the TCP throughput in Mbps while Table 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 3) and boxplots (Figure 4) of both TCP and UDP results. The results have less variability especially in VXLAN and weave modes. Figure 5 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.

Table 1. *Iperf3 TCP throughput.*

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 2. *Iperf3 UDP 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

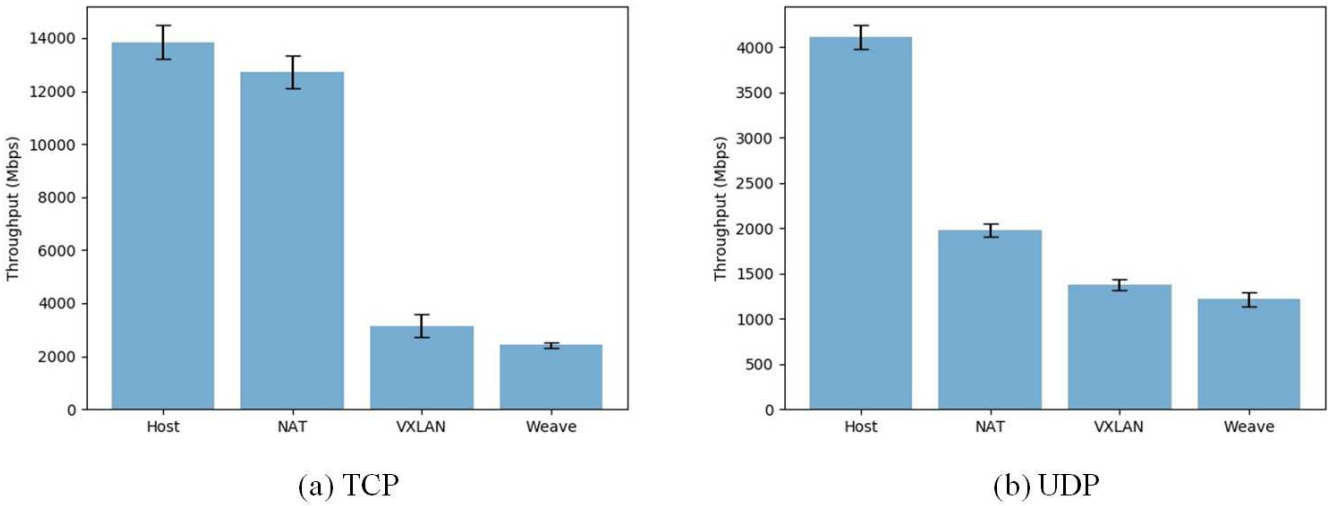

Figure 3. *Iperf3 throughput.*

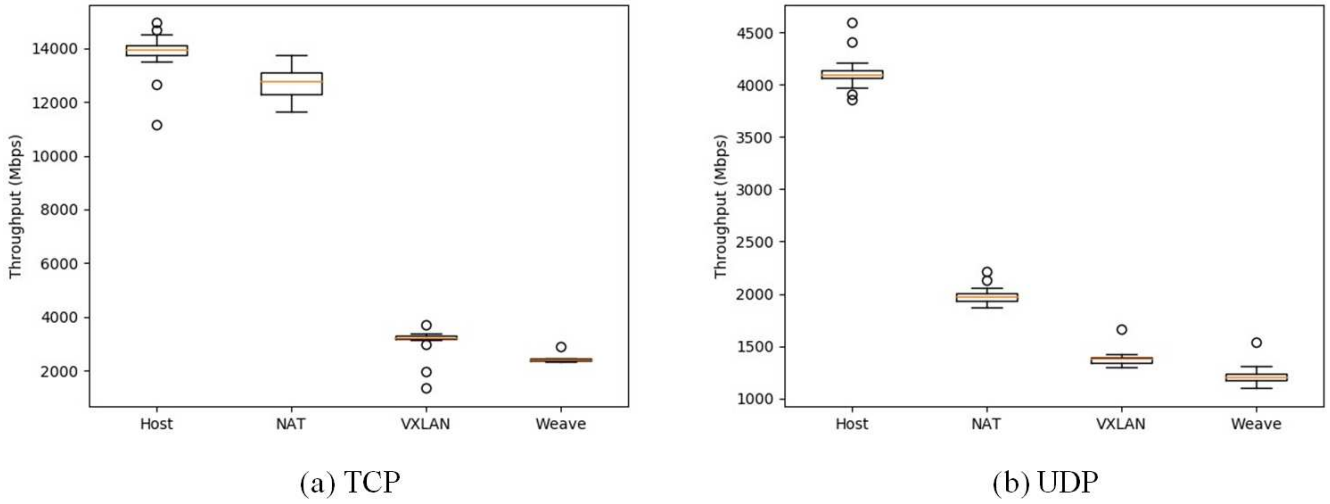

Figure 4. *Iperf3 throughput boxplot.*

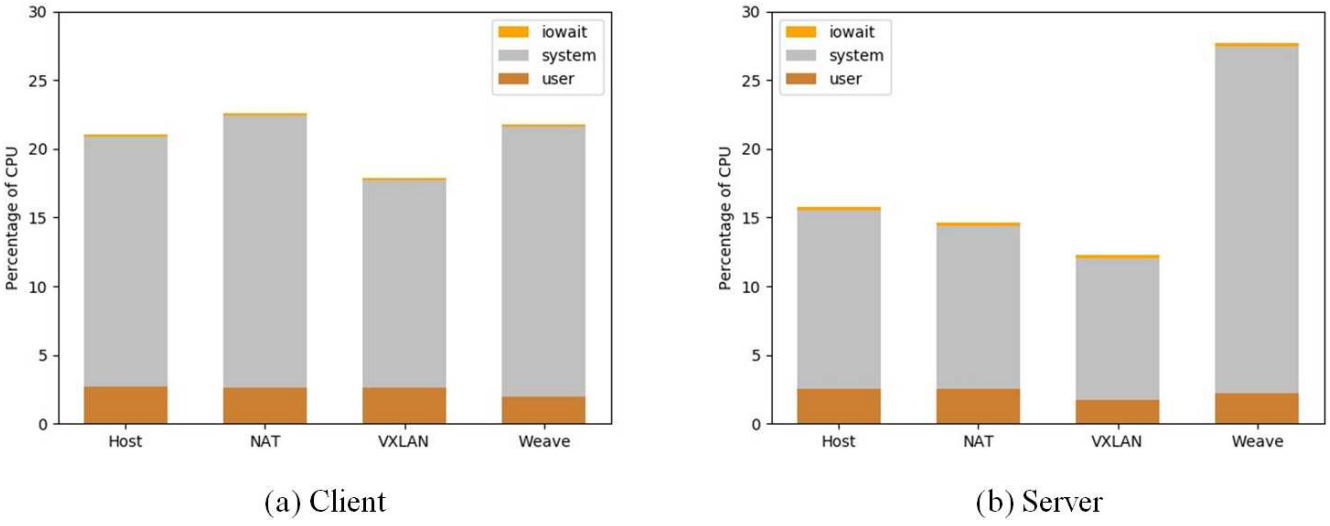

Figure 5. *CPU utilization of iperf3 client and server.*

4.2. 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 *memcached 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 3), average latency in millisecond and the corresponding deviation (Table 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 6 shows the bar plot of the throughput and latency while figure 7 shows the corresponding boxplots where less variability was observed with few outliers. Table 5 shows the mean and standard deviation of SET operation latency also in millisecond while Table 6 shows the latency of GET operation. The distribution of latency for Memcached SET and GET operations is shown in figure 8 where the two overlay network lines overlapped in both cases which means the performance of the overlay networks is almost the same. Figure 9 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.

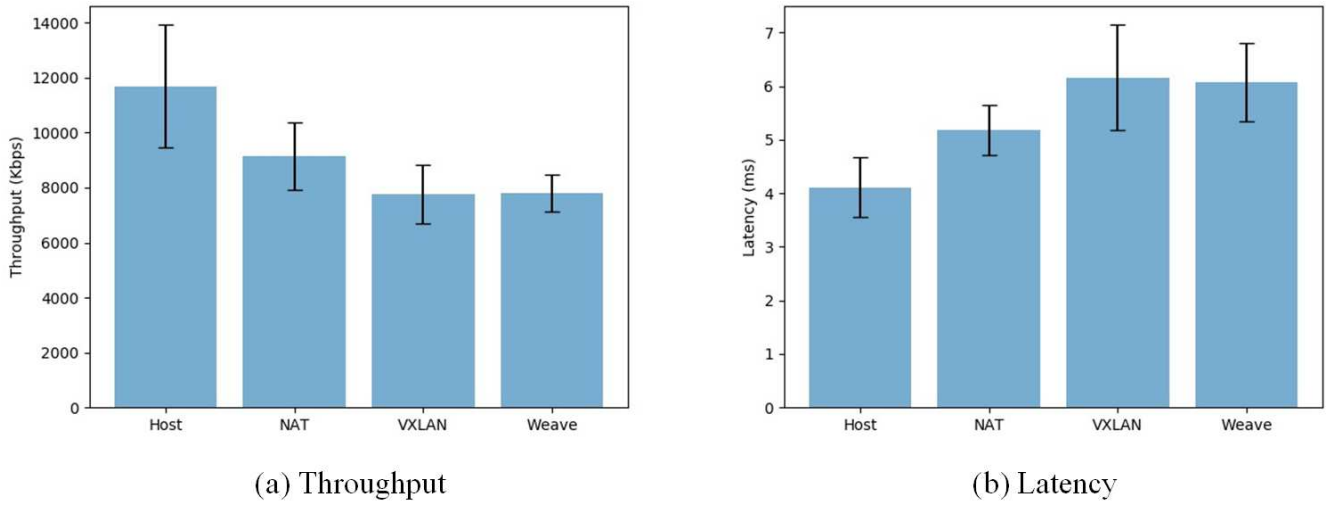

Figure 6. *Memcached throughput and latency with memcache text protocol.*

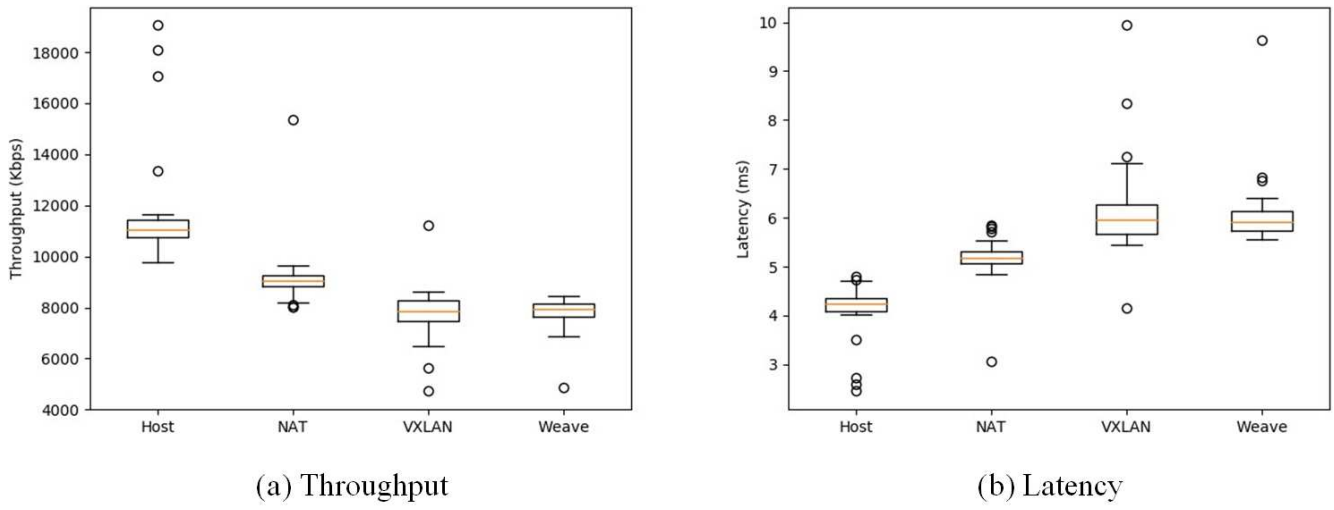

Figure 7. *Memcached throughput and latency boxplot with memcache text protocol.*

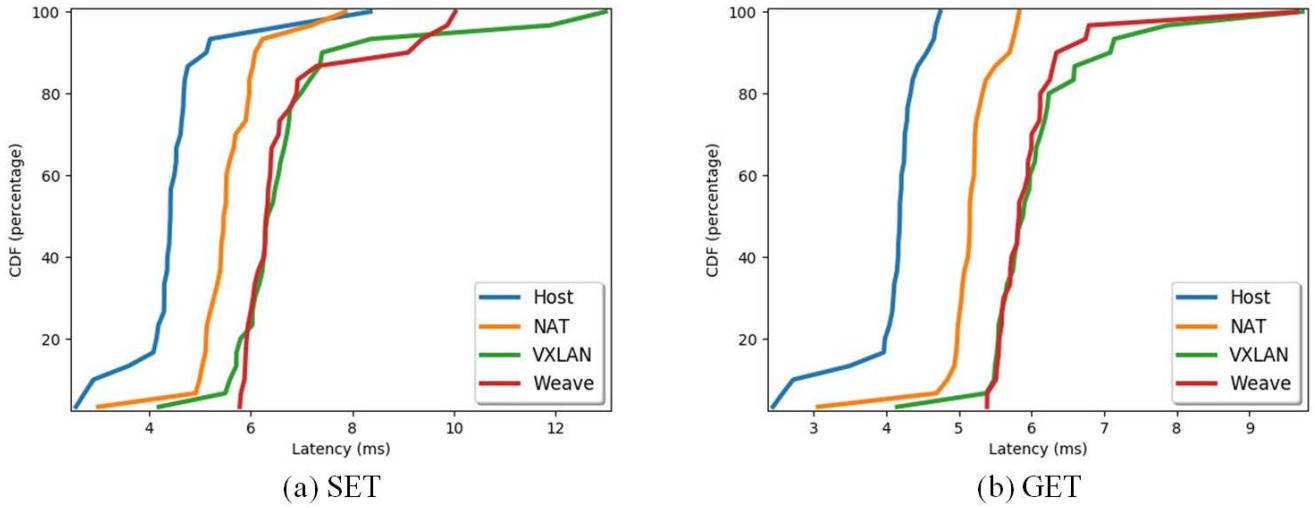

Figure 8. *Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. The two overlay network lines overlap.*

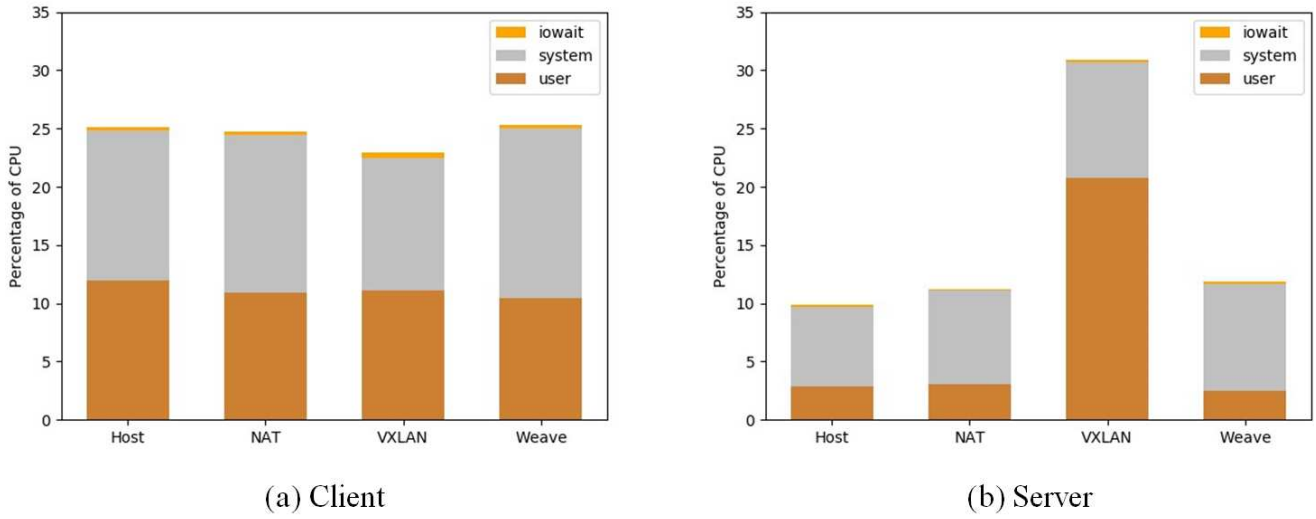

Figure 9. *CPU utilization of memcached client and server.*

Table 3. *Memcached throughput with memcache text protocol.*

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 4. *Latency for responding to Memcached command 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. *Latency of Memcached SET operation 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 6. *Latency of Memcached GET 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

Moreover, for the *memcache binary* protocol, Table 7 shows the average throughput in Kbps and their standard deviation while Table 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 10 shows the bar plots of the throughput and latency with error bars denoting standard deviation. On the other hand, Figure 11 shows the corresponding boxplots, the results did not get much variability and there are few outliers. Table 9 shows the mean and standard deviation of SET operation latency also in millisecond while Table 10 shows the latency of GET operation. The distribution of the latency for Memcached SET and GET operations is shown in Figure 12 where the two overlay network lines overlapped in both cases which means the performance of the overlay networks is almost the same. Figure 13 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.

Table 7. *Memcached throughput with memcache binary protocol.*

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 8. *Latency for responding to Memcached command 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 9. *Latency of Memcached SET operation 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 10. *Latency of Memcached GET 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

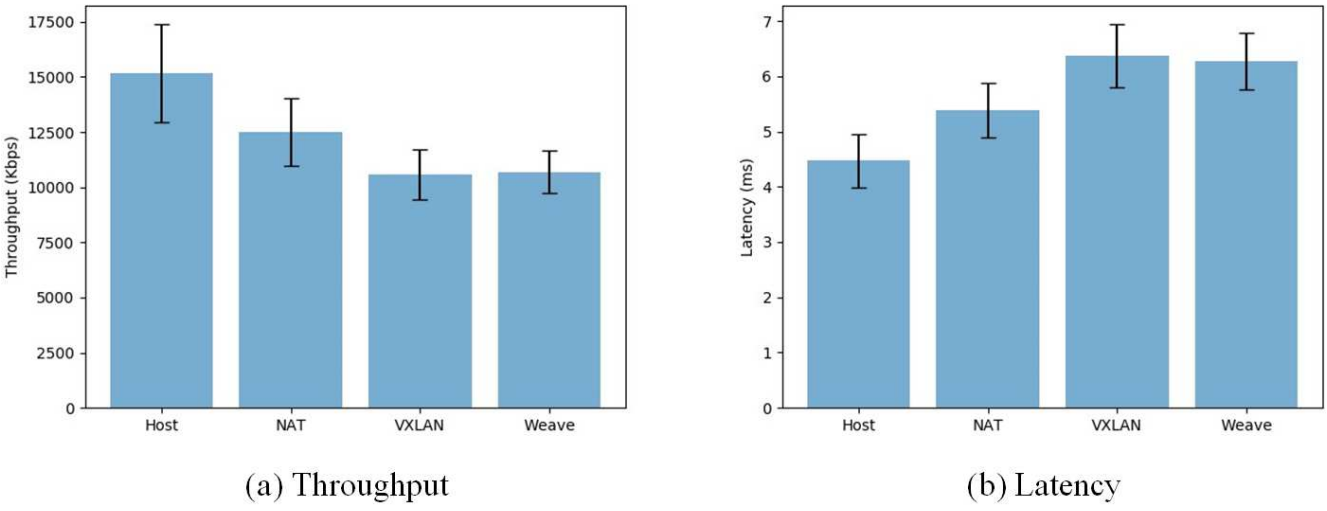

Figure 10. *Memcached throughput and latency with memcache binary protocol.*

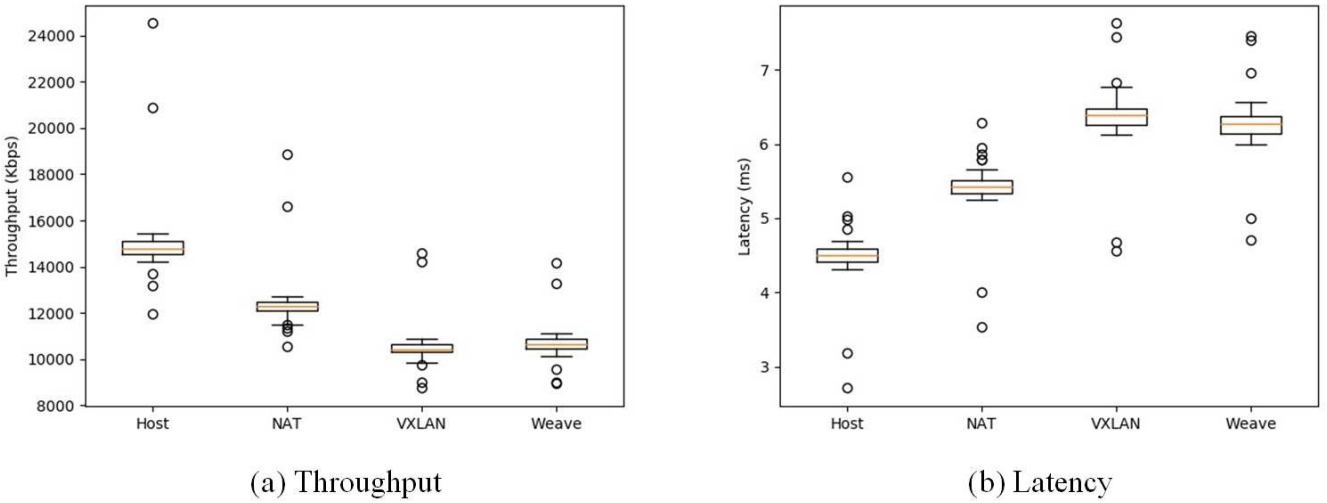

Figure 11. *Memcached throughput and latency boxplot with memcache binary protocol.*

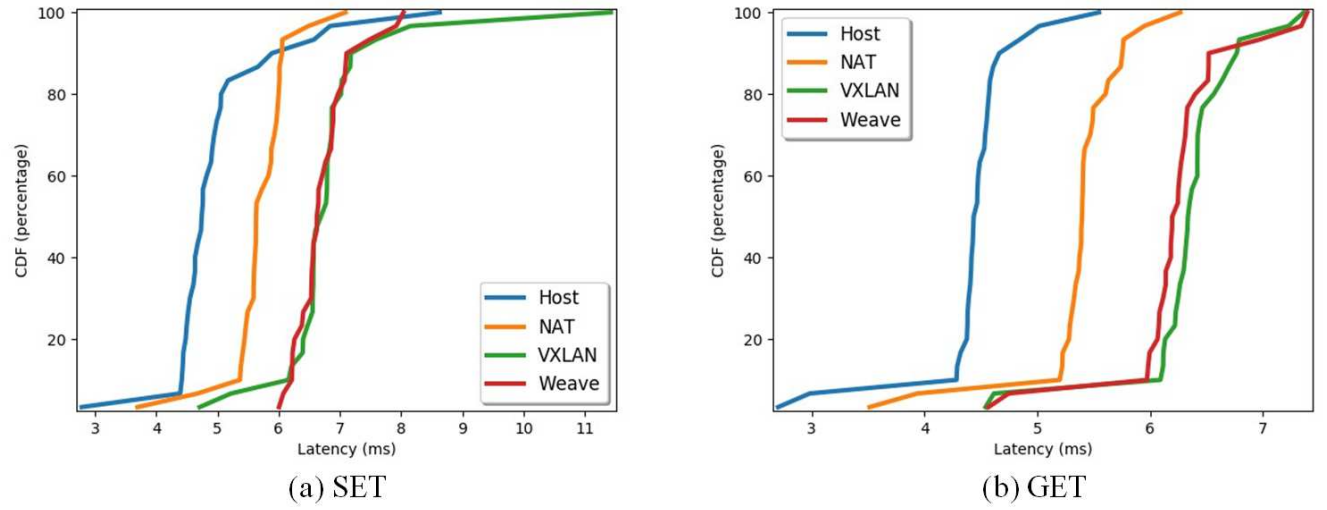

Figure 12. *Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. The two overlay network lines overlap.*

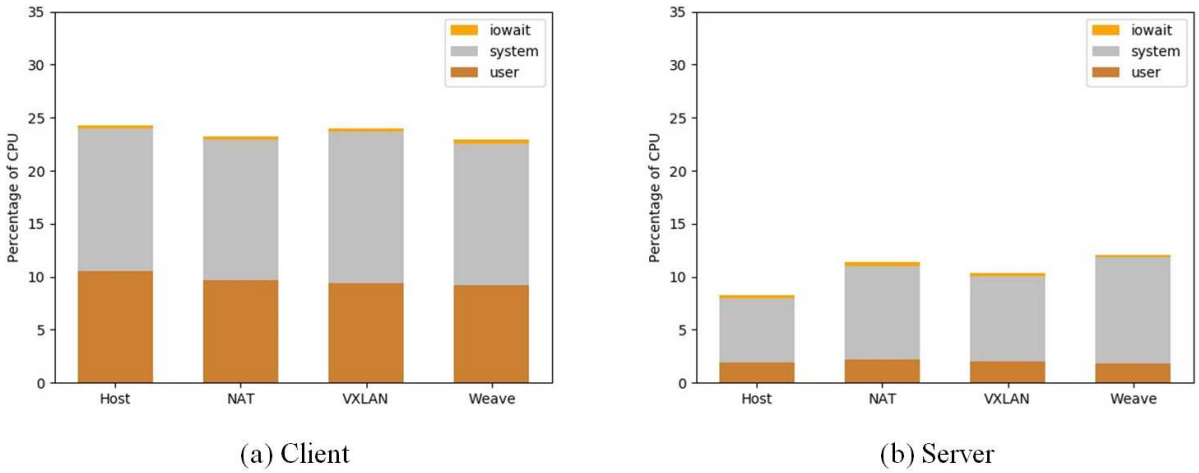

Figure 13. *CPU utilization of memcached client and server with memcache binary protocol.*

4.3. 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 11 shows the average of the results and their standard deviation. Figure 14 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 15 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.

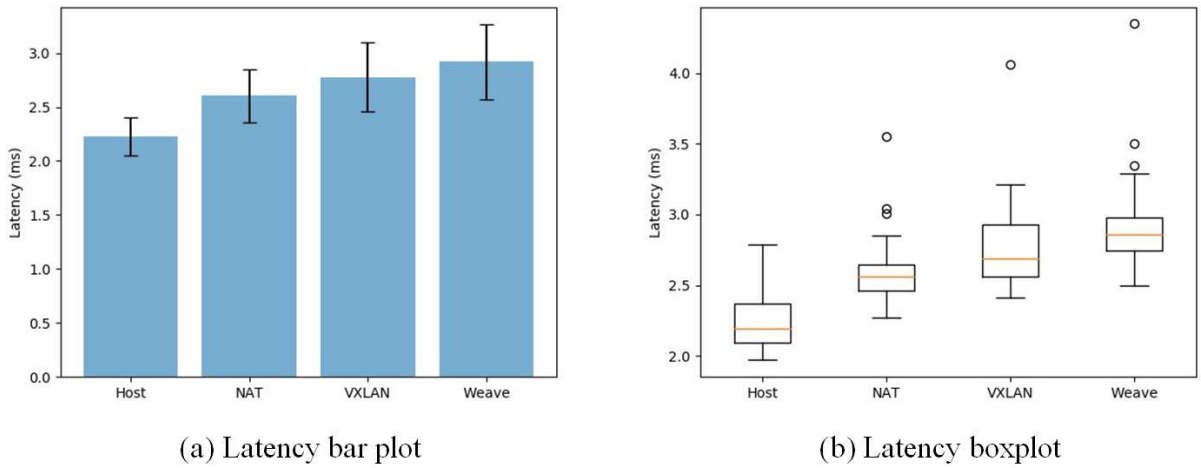

Figure 14. *3K reqs/sec Nginx 1KB latency.*

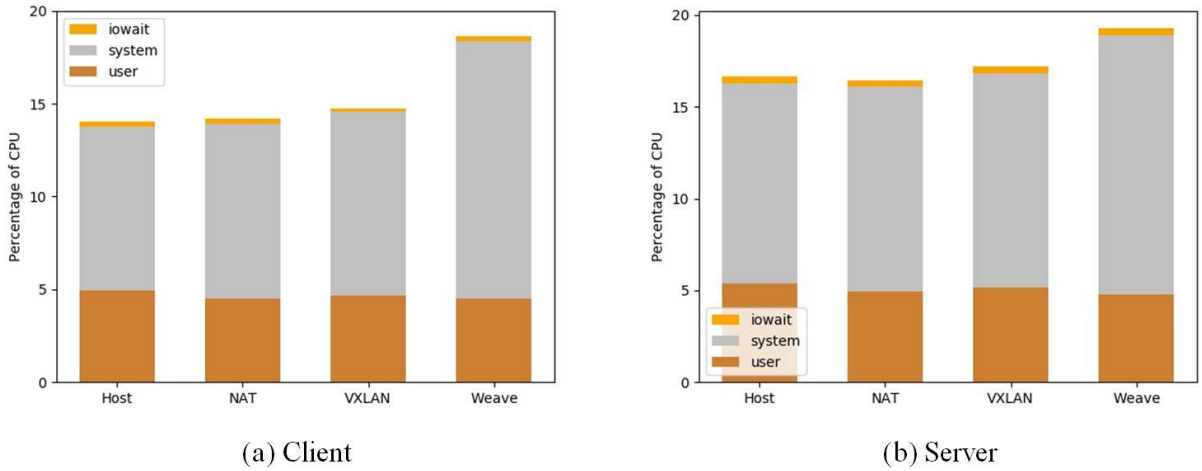

Figure 15. *CPU utilization of Nginx client and server in 1KB file.*

Table 11. *Nginx 1KB html file latency.*

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

1MB HTML file: thirty samples were collected by setting the throughput to 3K requests/second for each experiment on the four modes. Table 12 shows the average of the results and their standard deviation. Figure 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 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 17 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.

Table 12. *3K reqs/sec Nginx 1MB html file latency.*

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

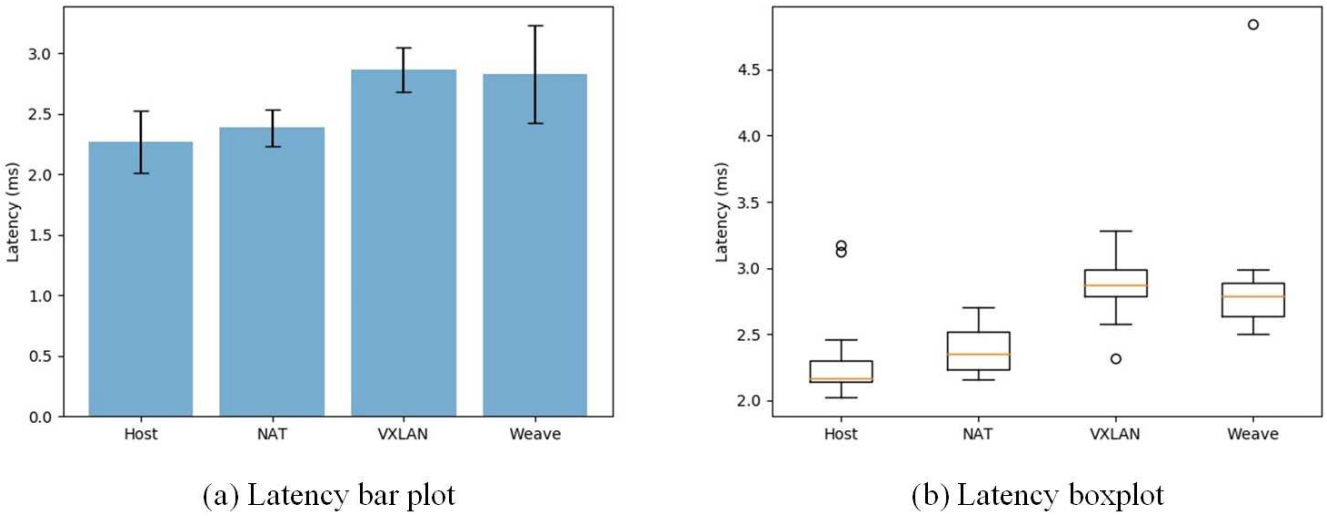

Figure 16. *3K reqs/sec Nginx 1MB file latency.*

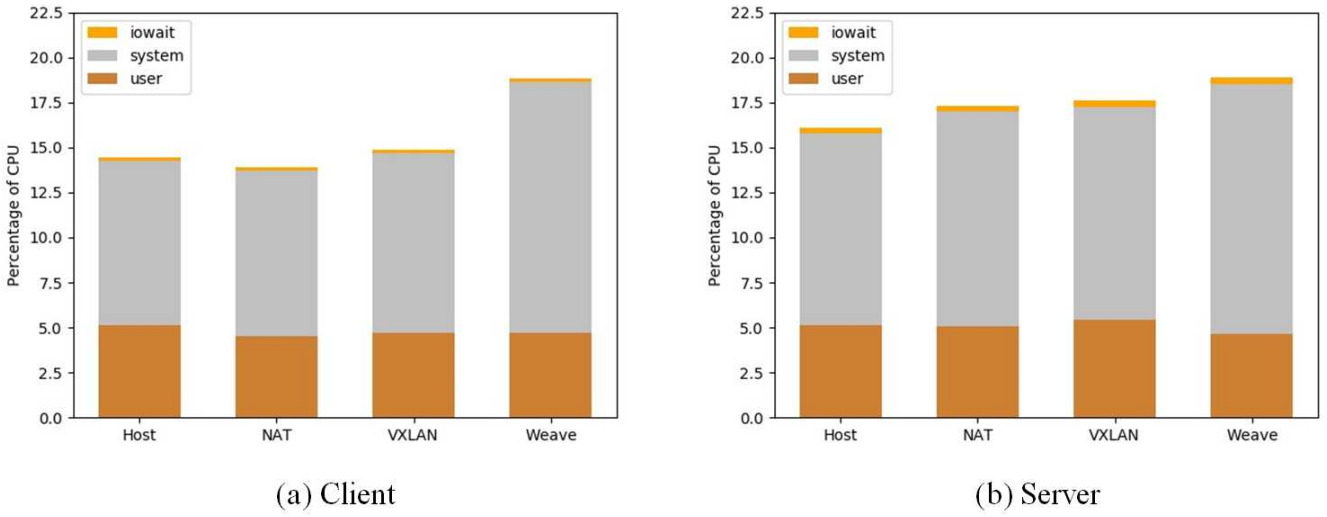

Figure 17. *CPU utilization of Nginx client and server in 1MB file.*

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.

4.4. 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 13 shows the average of the results and their standard deviation. Figure 18(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 19 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.

Table 13. *PostgreSQL latency on 300 trans/sec.*

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

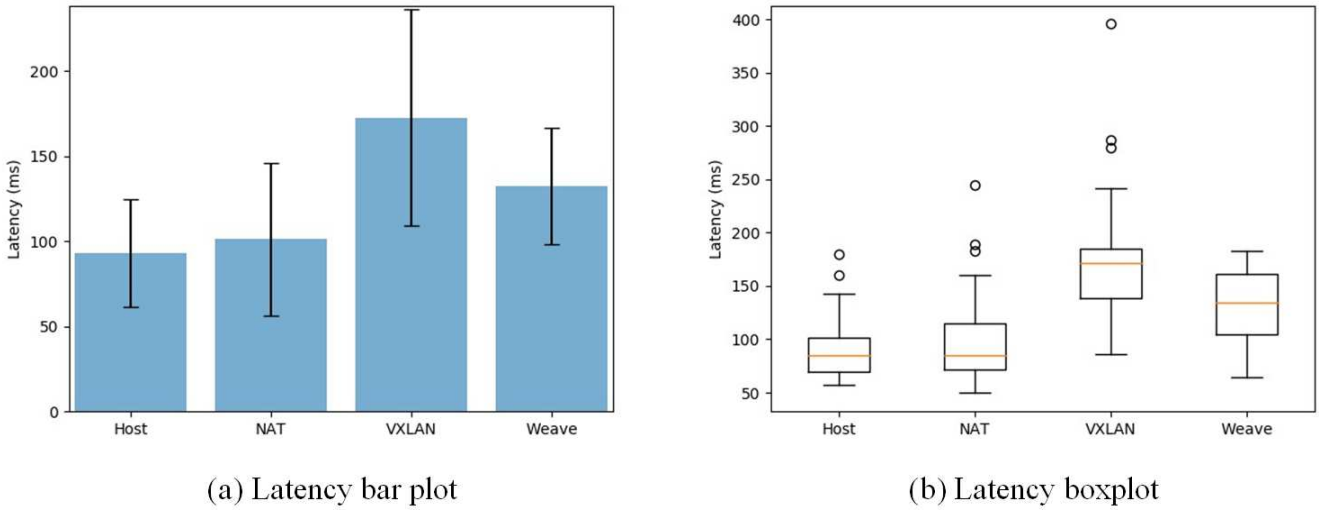

Figure 18. *PostgreSQL latency on 300 trans/sec.*

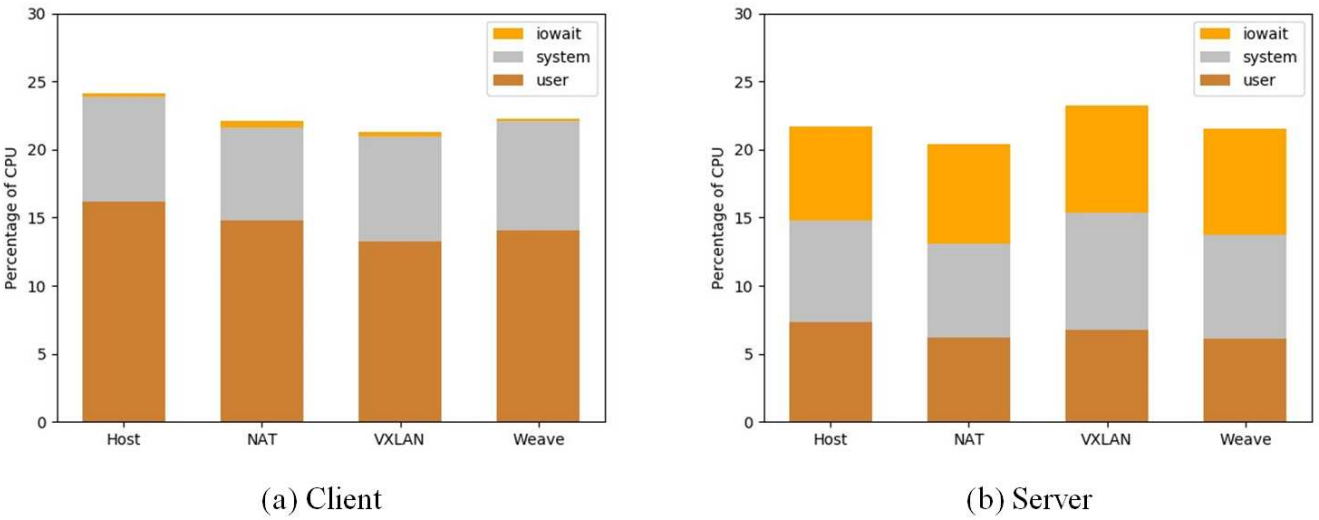

Figure 19. *CPU utilization of postgresql client and server.*

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 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 20 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 21 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.

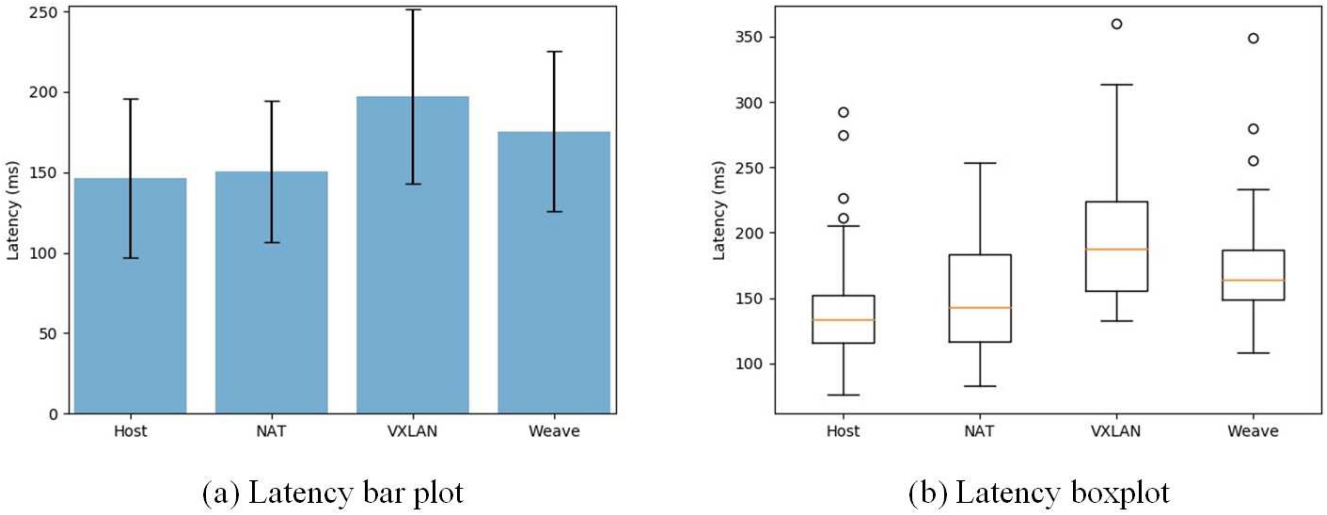

Figure 20. *PostgreSQL latency on 500 trans/sec.*

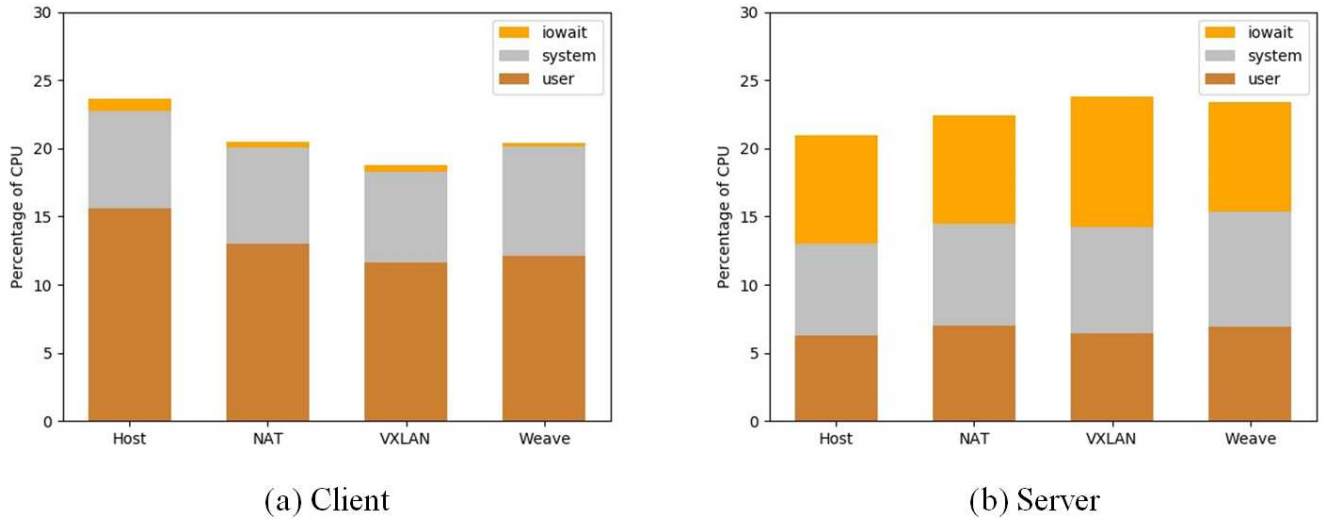

Figure 21. *CPU utilization of postgresql client and server.*

Table 14. *PostgreSQL latency on 500 trans/sec.*

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

4.5. Preliminary Results of the Reduced Overhead of Some Applications

The experimental results reported from the three applications and iperf3 revealed the best performance in the host mode 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 worst 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 15 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 22(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 23 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 improve the performance of VXLAN by 11.4% and weave by 22.8%. Table 16 shows the average and standard deviation of the thirty samples collected. Figure 24(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 25 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. One thing unique about our work is reducing the overhead of Docker default overlay network (VXLAN) in addition to *Weave* using the RPS technique. Other works like [15] reduced the overhead of only *Weave*. Some works like [2] performed an analysis of different docker networking modes which reveals the modes that have the best performance, this work does not optimize the overhead of overlay network(s) despite of the fact that overlay networks have solved some of the problems faced when using host and NAT modes. Our work does not optimize the overhaed of only VXLAN but that of both VXLAN and Weave.

Table 15. *PostgreSQL latency on 300 trans/sec with RPS.*

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*

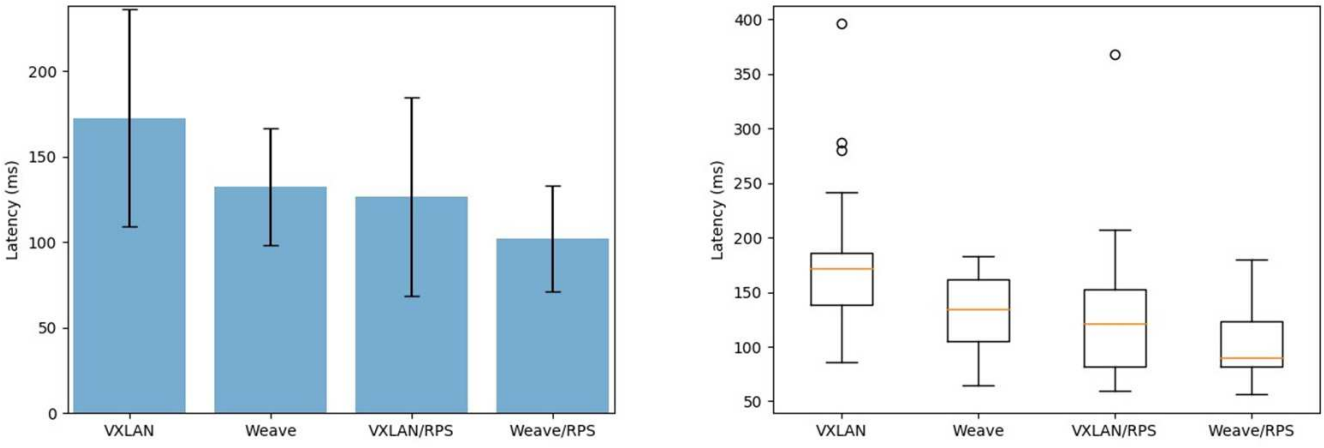

Figure 22. *PostgreSQL latency on 300 trans/sec with RPS.*

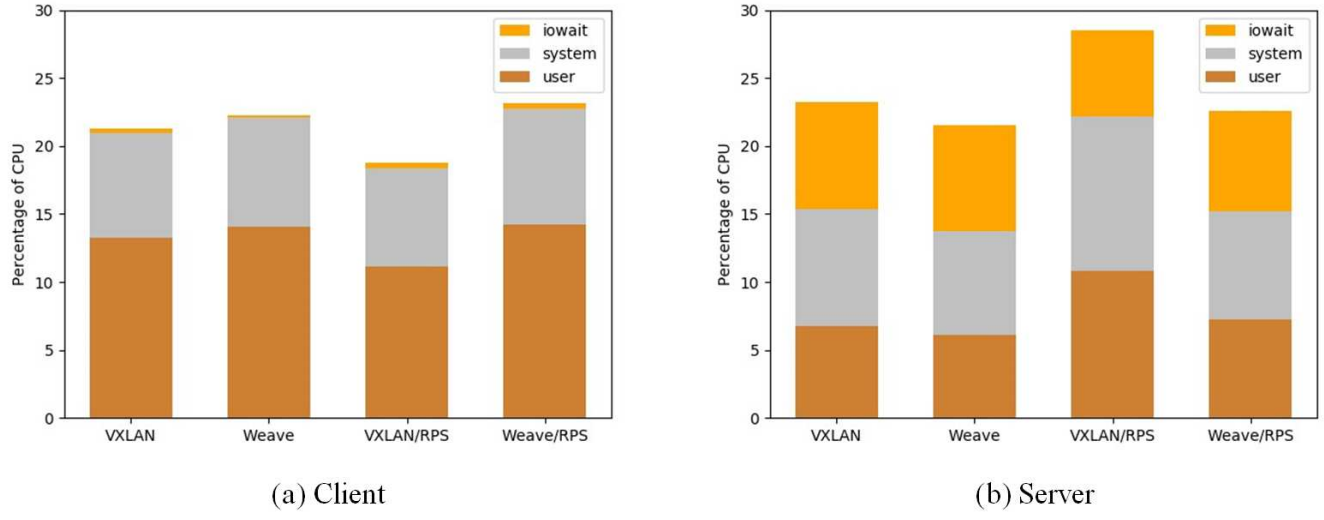

Figure 23. *CPU utilization of postgresql client and server.*

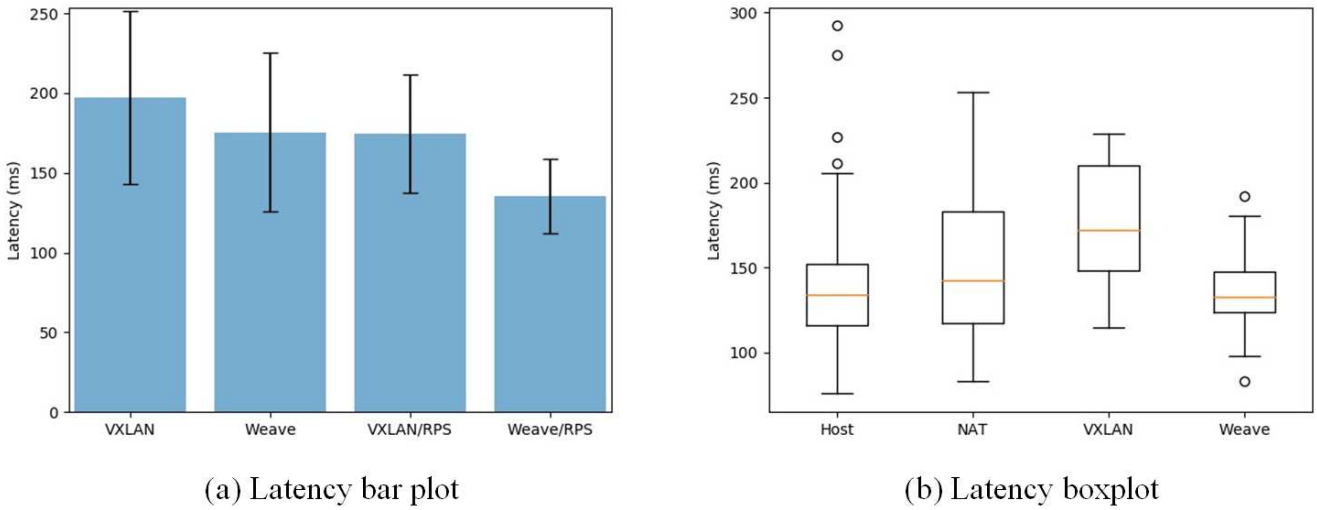

Figure 24. *PostgreSQL latency on 500 trans/sec with RPS.*

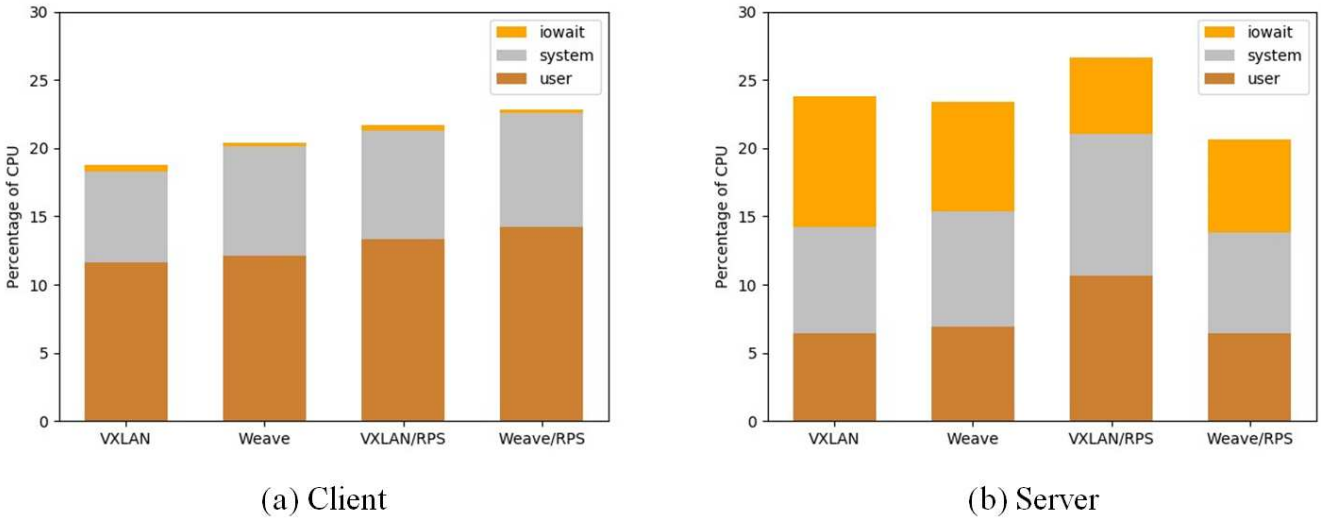

Figure 25. *CPU utilization of postgresql client and server.*

Table 16. *PostgreSQL latency on 500 trans/sec with RPS.*

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*

5. Conclusion

In this work, we conducted an 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 (Docker default overlay and Weave) used in this work have some performance overhead when compared with the first two modes (host and NAT). 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 reduce 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 Docker default overlay that uses VXLAN tunnel. This overlay network was not employed in previous studies [15]. Our work seems to be the first to approach overhead optimization of this network. The study findings can help users in selecting 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. However, the main challenge of this study was to tune the testbed and perform some tests with RPS/RFS (Receive Packet Steering/Receive Flow Scaling).

References

[1] Wikipedia contributors, (2019) "Virtualization." [Online]. Available: https://en.wikipedia. org/wiki/Virtualization

[2] 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.

[3] "Docker." (2018) [Online]. Available: https://www.docker.com/

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

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

[6] "Sparkyfish." (2018) [Online]. Available: https://github.com/chrissnell/sparkyfish

[7] "Sockperf." (2017) [Online]. Available: https://github.com/Mellanox/sockperf

[8] "iperf." (2018) [Online]. Available: https://iperf.fr/

[9] Wikipedia contributors, (2020) "Network address translation." [Online]. Available: https://en.wikipedia.org/wiki/Network_address translation.

[10] 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.

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

[12] "Weaveworks." (2019) [Online]. Available: https://www.weave.works/docs/net/latest/ overview/

[13] "Flannel." (2018) [Online]. Available: https://github.com/coreos/flannel/

[14] "Calico." (2018) [Online]. Available: https://github.com/projectcalico/calicoctl

[15] 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.

[16] Wikipedia contributors, (2021) "Virtual extensible lan." [Online]. Available: https://en.wikipedia.org/wiki/Virtual Extensible LAN.

[17] "Memcached." (2018) [Online]. Available: https://memcached.org/

[18] "memtier benchmark." (2019) [Online]. Available: https://github.com/RedisLabs/memtier_benchmark

[19] "Nginx." (2019) [Online]. Available: https://nginx.org/en/

[20] "wrk." (2019) [Online]. Available: https://github.com/giltene/wrk2

[21] "Postgresql." (2020) [Online]. Available: https://www.postgresql.org/

[22] "pgbench." (2020) [Online]. Available: https://www.postgresql.org/docs/9.5/pgbench.html

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

[24] "sar(1) - linux man page." (2019) [Online]. Available: https://linux.die.net/man/1/sar