TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 1041 – 1051
© TÜBİTAK
doi:10.3906/elk-1806-154

A study on application container resource efficiency

Özmen Emre DEMİRKOL[1,*], **Cemil ÖZ**[2], **Aşkın DEMİRKOL**[3]
[1]Informatics and Information Security Research Center (BİLGEM - TÜBİTAK), Gebze, Kocaeli, Turkey
[2]Department of Computer Engineering, Faculty of Computer and Information Sciences, Sakarya University, Sakarya, Turkey
[3]Department of Electrical and Electronics Engineering, Faculty of Engineering, Sakarya University, Sakarya, Turkey

Received: 21.06.2018 • **Accepted/Published Online:** 10.12.2018 • **Final Version:** 22.03.2019

Abstract: Nowadays, the IT service environment develops in a dynamic, rapid, and unpredictable way. Microservices and application containers in this process have a significant impact on new generation IT service models. The fact that they have important capabilities such as modelability, presentability as service, and restructurability, are reasons for preferring them in many areas. Moreover, microservices can meet various needs of IT personnel. As it is known, all server system components, such as CPU, network, hard-drive I/O, affect energy consumption. At this point, microservices also play an important mediator role in resource management. Energy consumption of microservice-based applications is lower than that of the traditional approaches. However, there are still cases of recoverable and unnecessary consumption at some points. Microservices can be monitored and controlled using many methods. Thus, this provides us with opportunities to recover the wasted energy resources considerably. In this article, the effects of container-based microservice architectures on the energy consumption of the system and how to reduce these effects are presented. For this purpose, a methodology, which has 3 approaches (disconnect, pause, stop), and a tracing mechanism are proposed. The results show that this methodology has a considerable effect on energy efficiency.

Key words: micro services, application container, docker, resource efficiency

1. Introduction

New approaches direct data centers from a monolithic structure to more scalable, adaptive, and flexible ones that can respond to changes and new requests rapidly.

Monolithic structures, which have many limitations, are intended for operating on a single flow studies. The limited movement of all IT staff from software developers to system administrators in this field has caused a search for new alternatives in today's studies.

Nowadays, it is much easier and more rational to meet these demands with architectures such as containers and microservices. Due to the rooted and hard-to-change structure of classical architectures, microservices and containers provide convenience in expanding, monitoring and manageability for developers and practitioners.

Nowadays, the most common examples of microservice architectures are seen in service providers such as Netflix video streaming and Amazon [1].

In recent years, the tendency to use microservices has been increasing within the scope of both commerce and education. Although this application requires more effort compared to others, these structures are preferred due to their agility, resilience, and maintainability.

*Correspondence: ozmen.demirkol@tubitak.gov.tr

This work is licensed under a Creative Commons Attribution 4.0 International License.

Microservices can communicate via REST protocols. Application programming interfaces (APIs) are commonly used in the IaaS (infrastructure as a service) layer in cloud computing. Their more lightweight structures not only decreases their cost in resource utilization but also accelerates migration operations. Microservices are actually not a new issue. It is possible to encounter a study conducted in this field as an exokernel operating system in 1995 [2]. The idea here is to provide each application with its own infrastructure on hardware in the operating system. Isolating PID (Process ID)-based operations with the “Jail” mechanism in BSD operating systems by using this idea has been used for a long time.

This structure has found itself an important area for research, development, and utilization in the cloud computing due to its rapid development in recent years. Increasing the need for more efficient use of resources to be provided to customers especially in the commercial field accelerates the development in this area.

There is no doubt that some new problems and difficulties appear together with each innovation and convenience. For example, nowadays microservice and container structures widen the area of attack that can be performed on the system [3]. Moreover, they increase difficulties in issues such as traceability and logging. While studies on these issues continue, the resource utilization of these technologies and other innovative researches also continue.

Although container-based microservices have better resource consumption data when compared to virtualization, there are still areas to be recovered. In this study, it was aimed to identify the parts that are wasted in the resource consumption of container-based microservices with the approach at this point and to reveal the ways by which resources can be regained to the system among these. The results indicate that it is possible to make significant recovery depending on the scale.

The rest of the paper is organized as follows:

The related work is presented in Section 2. Monitor and trigger are described in Section 3. The environmental setup of the study and our approach are described in Section 4. Finally, the results and our conclusions are presented in Sections 5 and 6.

2. Related works

Microservices can be named as new-type operating systems. Since common OS’s support various hardware and service components, they are designed as large-scale. Thus, this makes them bulkier [4]. Furthermore, due to the elastic structure, it provides a speed advantage in processing times starting from a container-based microservice boot time consisting of only related components depending on the activity planned to be carried out. In a study conducted in this area, the benchmarking study results related to microservice boot time, image formation and resource overflow on OpenStack, which is an important cloud computing stack structure today [5].

The fact that microservices are reusable, auditable, and maintainable from the point-of-view of developers provides an opportunity for action in a large area. Moreover, it is observed that adaptation to software test processes and changes in the project life-cycle improves in projects conducted by using microservice architecture; moreover, this provides assistance to a more efficient use of the behavior-driven development (BDD) concept ensuring that the process is understood by the client better [6]. Another advantage emerges in code portability. The structure of container-based microservices operating as image-based also provides an opportunity to use a template code [7].

Recently, a more flexible microservice architecture has started to be used especially by major developers instead of old monolithic structures, and these transformations are tested, and their results are compared in many areas [8].

Together with the fact that the "Internet of things" term has started to be mentioned with "cloud computing", microservices are tested in studies to meet the needs from the software aspect as an intermediate layer [9]. It is observed in the literature that commodity architectures continue on the hardware side; moreover, specialized hardware operating as image-based specifically to microservice is also used [10].

In the studies conducted in the fields of resource efficiency and costs of microservices, while bottlenecks in the monolithic approach, in which the loads brought to the operating system are examined, generally appear in the storage and network aspect, they may result from APIs providing communication with the operating system in microservices. Among the studies conducted on the resource efficiency and costs of microservices; studies examining the loads brought to the operating system [11], approaches revealing cost-effectiveness up to 70% in systems in which migration from monolithic structure to microservice architecture takes place [12], studies pointing out the management of balancing the load in the network caused by microservices and indicating that it may bring excessive load to the system [13], publications stating that microservices will naturally increase the CPU cycle due to API calls and thus increase resource consumption, and resource consumption may be even higher when compared to the monolithic structure in cases in which orchestration and stabilization mechanisms are not good [14], and academic studies examining whether the Docker container system has positive effects on the microservice architecture are encountered [15].

When considered in terms of bottlenecks on the system, while this problem generally occurs from the storage and network aspects in the monolithic approach, it may originate from APIs ensuring communication with the operating system in microservices. Administrative operations (mount, create file system, etc.) can bring load to the system. For example, as default, the Docker container uses the network topology of the system as a bridge mode, and this increases resource consumption values. When the container system is set to use the physical ethernet, the consumption decreases directly.

The approaches that we have developed to reduce the resource consumption in container-based microservice architectures and their results are presented in this study. One of these approaches, situational behavior in the network topology and resource limitation in passive states, which was examined in academic studies in different ways, and how it was used at different points were examined [16]. In another study, it was observed that virtual machines are examined according to their resource usages as an approach to reduce the resource consumption of the system, the virtual systems in physical machines are gathered at a point by being migrated at a certain time or state, and the resources remaining free are suspended or closed [17]. These studies have found a new study area on container-based microservice architectures in our study.

3. Monitor and trigger

Monitor and trigger is a commonly used binary type technique. Almost each system service creates one or more logs depending on its own state. In some cases, these logs are used to trigger another service. Depending on the requirement, a service may remain on standby until special conditions occur. Similarly, time can take the place of monitor and something else can be triggered at a certain bit τ moment.

In this study, first, the condition of application containers and active connections will be monitored by using monitor and trigger technique, then the approaches related to the resource management will be triggered when special conditions occur. The questions are that "Can we detect the situation when containers are on standby or there are no active connections and can we ensure resource saving with the approach from that?" We will test the following three possible ways to find answers to these questions. Disconnecting container from the underlay network, pausing the container-related PID by using the cgroup feature and stopping container

to recover all resources used by it. These three situations are explained in Section 4.4. Furthermore, another important problem is enlightened in Section 4.3. How will a container in the disconnected/paused/stopped state respond to the demand of a client?

4. Environmental setup and approach

4.1. Physical environment

An HP Proliant DL380 G7 server was used for the test environment. The server has 2 x Intel Xeon X5650 @ 2.67 GHz (12 Cores) processors, 18 x 4096 MB, in total 72 GB DDR3-1333 MHz memory, 146 GB "RAID10" HD, and Broadcom NetXtreme II BCM5709 1 GB Network card. An Emerson power distribution unit (PDU) with its software and server power monitoring was used together to measure the power consumption.

4.2. Software environment

Operating System - CentOS 7.2, Linux kernel - 3.10.0-514.2.2.el7.x86_64 , shell - bash version 4.2.46, network listener – tcpdump 4.5.1, firewall – iptables 1.4.21, firewall manager(firewalld) – 0.4.3.2 and container platform - docker version 1.10.3 were used. Moreover, a hundred containers are used in the system: fifty (50) httpd (web server) and 50 mysqld (database server).

4.3. Process flows and methodology

Two shell scripts were encoded in this laboratory study. While one of them monitors the container's condition and changes the status when possible, the other one responds to the network traffic connected to the relative container and awakens the container when required. During the operation, Process-I firstly takes active container names from the system. Then, it solves the PID data of the active container names. All operations in a Linux system have a specific folder in the form of /proc/$PID under /proc. Information about the relevant operation is registered here. Process-I accesses the /proc folder of the related container and network state files under it. For Docker, all containers have /proc/$PID/net/{tcp,tcp6,udp,udp6} files. These files produce real-time information. Process-I checks out the script in TCP files and searches for active connections. If it finds an open connection, it leaves the container as it is. Otherwise, it carries the state of the container to one of the three preferred ways in the script (disconnect, pause, stop). After the change of the state, Process-II starts on its own. Process-II reveals which container is out of service with the information received from Process-I. It starts a dynamic tcpdump subprocess.

When a demand is received to get the container of the port to the service status, the port information that is the output of Process-I in a loop responds to the port related to tcp flag ack/syn additional compacting and triggers the container.

When the information of the related port is caught by tcpdump, the inverse of the command previously used to activate the container is run. Exp. "*$docker network connect bridge www01*" , "*$docker unpause mysql01*", "*$docker start www02*".

This flow was pursued within the framework;

These processes and link between them visualized as a flowchart in Figure 1.

Before performing all these operations, it is required to change something in the system, for example, a firewall. Most Linux distribution uses iptables as a firewall of the system. A firewall can be used to protect the system against undesired accesses-requests, block, drop or redirect network packets. A firewall can also be

Algorithm 1 Process-I: system analyzing

Ensure: *Running Docker System* ≥ 0

while true **do**
Get active containers' ids
Analyze containers' established connections
if there is any **then**
Change containers' status $\rightarrow$ *DISCONNECT/PAUSE/STOP*
GoTo Process-II: passive containers' list
return *Get active containers' ids*
else
return *Get active containers' ids*
end if
end while

Algorithm 2 Process II: port listening

Ensure: *Running Docker System* ≥ 0

Require: *Passive containers list* ≥ 0

while true **do**
Create list for passive containers' ports
Listen ports for incomming requests
if container is online **then**
Delete port (Manually started container or faulty list)
return *Create list for passive containers' ports*
else
Activate the container $\rightarrow$ *CONNECT/UNPAUSE/START*
Delete port
return *Create list for passive containers' ports*
end if
end while

used to change the resource and target address information of network packets. Docker uses this logic to direct the network packets coming to the system to the IP address of the related container. This is called destination network address translation (DNAT), and it is used in the system by the docker-proxy. This proxy is separately regenerated/rerun/respawn for each container. There is a Docker rule chain in iptables and if the target port information in an incoming packet is numb, then it is directed to the related container IP address by masking the target IP address. As it is previously mentioned, in this study, iptables is used to run the flow properly. Like all other firewalls, firewalld—iptables manager—blocks all unauthorized or closed port requests in a predefined way. This means that the incoming connection request is rejected and responded as icmp-host-prohibited and icmp6-adm-prohibited to the client. In this case, the connection request is cut off by the client. We change this predefined case as packet dropping instead of blocking. By this way, a response about the rejection of a request is not sent by the client. When we change the firewall, we ensure that the client renews the request until receiving a response or reaching the time-out point. In this study, the repetition number of the client's requests for connection is determined as one. It is observed that the container is awakened and started to respond to requests during this process.

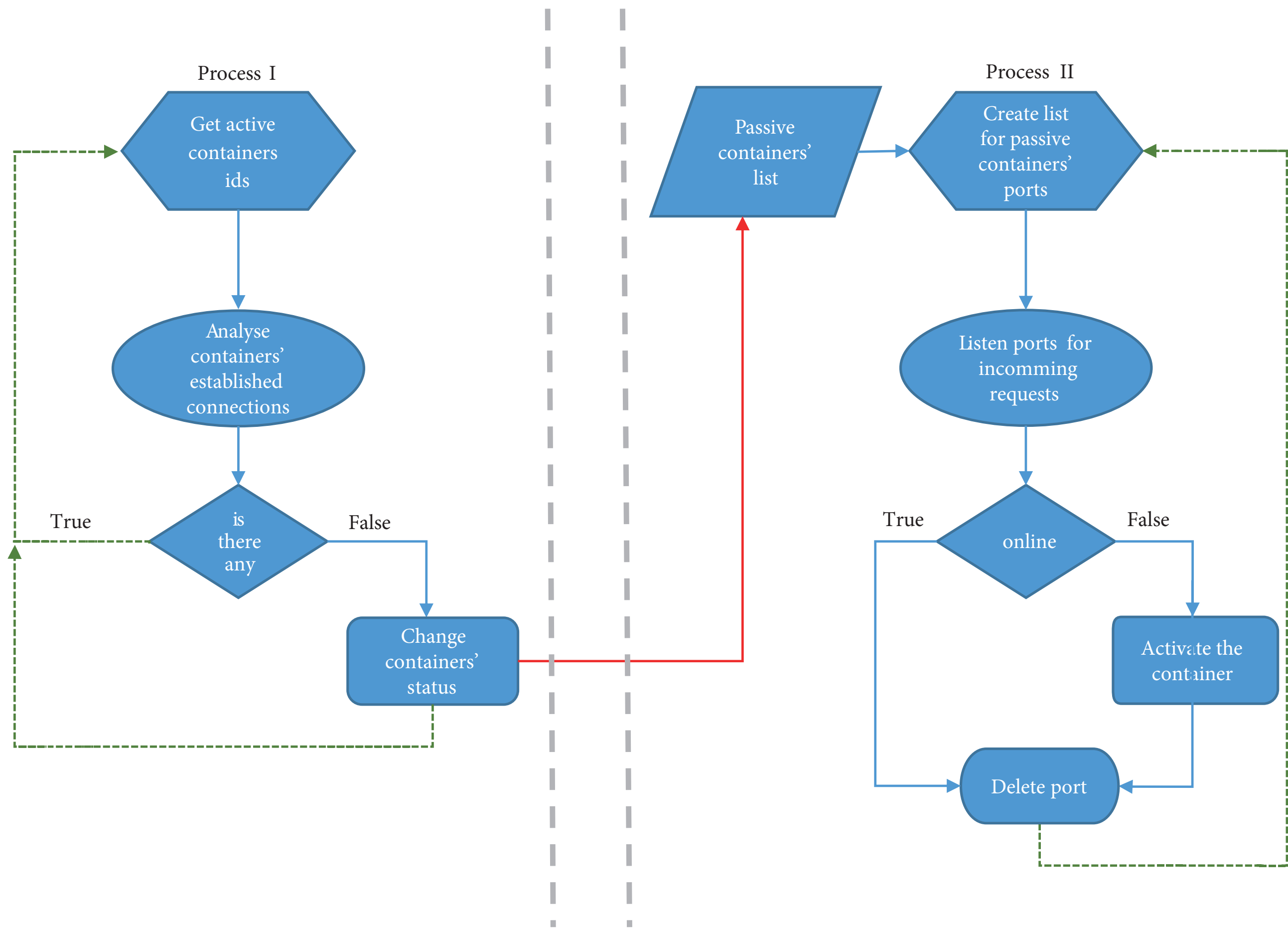

Figure 1. Flowchart of the processes and link between them.

4.4. Used approaches

As mentioned in Section 3, 3 methods in the Docker container system were used as an approach to reduce the resource consumption of the system. Every method was repeated 3 times and average of them was accepted as result.

Disconnect: Docker uses the predefined bridge technique to control and direct inputs and outputs in the network traffic of a container. Here, a private IP address is provided to the related container, a rule related to this is then added to the local firewall of the system. It is carried out with the docker-proxy found in this system. This structure is presented in Figure 2.

Each container with an IP address in the system causes 2 additional resource consumptions. One of them operates at the core level, and it is the net filter that catches incoming/outgoing packets on the network and can intervene, the other one is the docker-proxy that directs the incoming packets to the related container on the firewall.

One hundred containers are used in the tests as explained in Section 4.2. When the network connections are disconnected by using the related command of Docker, the PID of the related DNAT in Exp. "*$ docker network disconnect bridge www01*" firewall is terminated. The elimination of these rules and PIDs on which they are dependent does not affect the main PID. There are http and mysqld container services in our test environment.

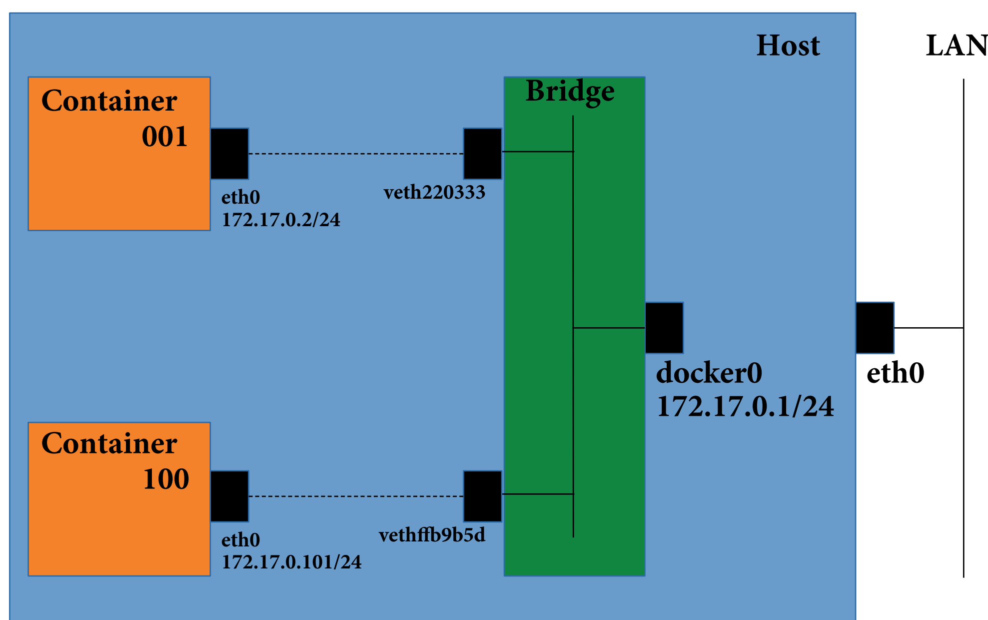

Figure 2. Docker-proxy network.

Pause: This is a suspending technique filling the process by means of cgroup. In this state, the operation itself is not aware of its condition. When the container is suspended by using the relevant Docker command, Exp. "*$ docker pause mysql01*", the CPU usage of the container is reduced to 0%. This only affects the CPU usage condition. The area separated for the container from memory remains as it is and continues to consume energy. At this point, a remarkable amount of CPU resource is regained to the system.

Stop: The Docker stop command stops all the resources of the relevant container. The Exp. "*$ docker stop www02*" command is an example used to recover resources and reduce consumption. Although no process remains on the system in relation to the container in this situation, the monitor and trigger technique we have suggested can determine the incoming requests to the relevant container and retrieve it back to the service status.

All these three techniques have different response times and resource consumption recovery values. The results will be discussed in detail in the next section.

5. Results

In this section, the effects of the three approaches on the CPU, memory, network I/O, and power consumption of containers are revealed.

Firstly, the normal energy consumption values of the containers are indicated. It was observed that two commonly used applications (httpd, mysqld) have approximately the same energy consumption. The iptables rule is used by an empty container for a docker-proxy organizing 0.04%–0.08% CPU, 0.02% memory, 0.01% network connections, and at least one DNAT. The spectrum was extended to 100 containers to examine this condition. At the beginning, when the system was running with zero containers, it consumed 122 W energy per hour. After run with 100 containers and waited for some time until the system became stationary, 137 W consumption per hour was determined. This means that each container in the stationary state consumed approximately 0.15 W energy per hour. The results are presented in Tables 1 and 2.

Table 1. Example of resource consumption for five different containers

CONTAINER	CPU %	MEM %	NET I/O
b7bcb1844c8e	0.08%	0.02%	5128 MB/3192 kB
b9e3ac08939e	0.04%	0.02%	7482 MB/1488 kB
b9ea351e749d	0.08%	0.02%	8679 MB/687 kB
c336225d7b71	0.07%	0.02%	14876 MB/4066 kB
c3bce5564f62	0.06%	0.02%	4798 MB/854 kB

Table 2. Zero/hundred containers system resource consumption.

	Power Consumption	CPU Idle	Memory Usage
Zero	122 W	99%	64883 MB
Hundred	137 W	94%	68850 MB

Power consumption (the more running container the more power consumption), CPU Idle (the less running container the more idle cpu), Memory Usage (the more running container the more memory consumption).

At this point, it is required to indicate how much CPU, memory, and energy resources Process-II, the monitor and trigger mechanism consume. The resource consumption of Process-II is almost equal to one container in a stationary state. It consumes 0.05% CPU, 0.02% memory, and 0.14 W power per hour. Even if the number of containers, which are being monitored by Process-II, increase or decrease, it is not considered additional consumption.

The beginning of the gaining energy points by using Process-II, advantages and disadvantages of the three approaches in resource consumption are discussed in detail in Sections 5.1, 5.2, 5.3.

5.1. Disconnect from network

It is the first and quickest method. Disconnecting a container from the network makes it possible to recover the resources that it is using on iptables and docker-proxy. Iptables runs at the core level. Thus, it has rapid and low resource consumption. All containers in the test environment provide 1.52 W energy recovery when the network connections of all containers are disconnected. When this result is reduced to a container, ∼0.015 W recovery is observed. 10 containers must be simultaneously disconnected from the network connections on the system for Process-II to be equal to the resource it consumes and more efficient. After this point, the approach starts providing a gain to the system. As additional information, with this approach, the restoration time of a container by Process-II is only 200μs (1/5 s).

5.2. Pause: freeze CPU Usage

As indicated in Figure 2, to pause a container by using the cgroup feature is a more effective method. Since the container itself does not know that CPU resources are available, when the resource is released, its return becomes faster. All containers in the system in the stationary state consume approximately 0.06% of the CPU power. As explained in Section 5, CPU constitutes the largest part of the general consumption. In this test, it is also measured that energy decreases approximately at the rate of 0.1 W (0.09). According to these results, pausing 2 containers recovers more resources to the system than Process-II consumes.

Considering the condition in terms of time, the recovery time of a container from a pause state is $400\,\mu$s (2/5sec). Since this is within the limits of packet loss time, it is fast enough to provide service before the client realizes.

5.3. Stop: cut all resources

The most effective and final approach is stopping the container. In general, a container is stopped for maintenance. However, this will not be a problem if there is another service to meet the access requests incoming to the system in maintenance and to remove the main service.

When a container is stopped, no related information remains on the system. The relevant PID disappears, and information about CPU, memory, network I/O, etc. is removed from the system. Therefore, when a container is stopped, significant resource recovery is provided. In the presented test, this value is 0.15 W per hour that is equal to the value of Process-II. Furthermore, the system provides a considerable recovery for each container.

At the same time, it is determined that the relevant container has become able to provide service again in less than a second.

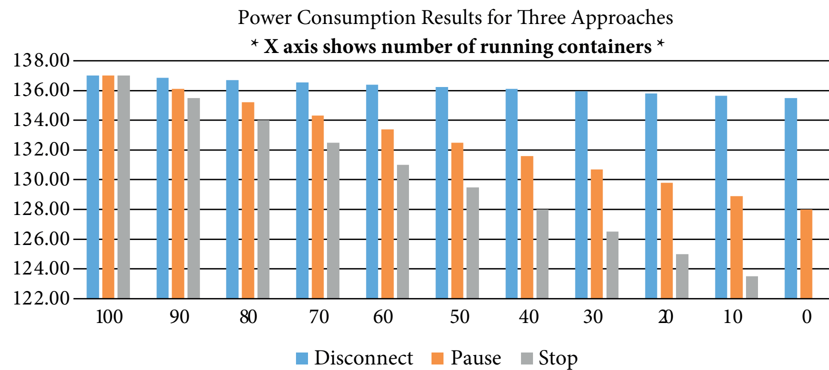

Figure 3. Power consumption results for three approaches.

6. Conclusion

In this study, how the efficiency of container-based microservices in resource utilization can be increased is demonstrated. CPU, memory, and network I/O have a direct effect on energy consumption and a container consumes 0.15 W power, ∼0.06% CPU and 0.02% memory, 0.01%CPU for the docker-proxy carrying out the process of network traffic follow-up and address modification even if it does not provide service. At this point, 3 approaches with regard to how resource recovery can be provided from containers when they are in an idle state are suggested. These are dropping all related operations by disconnecting from the network, freezing the CPU usage by using cgroup, and cutting off all resources by stopping them. It is accepted that one or more containers are in an idle state on the system for all these techniques to work. As indicated in Sections 5.1, 5.2, and 5.3 and the related figures, the considerable amount of resources is recovered. As the number of containers in the idle state increases, resource recovery increases more with the approaches we suggest.

Moreover, it is believed that this study provides new opportunities for researchers and developers. For example, an analysis system can be developed to make these approaches more effective. This mechanism can determine the idle intervals by examining the life cycle of containers and which containers will be appropriate for this technique depending on this and direct this to the system. This distinction can make the idea tested here more effective. In another study, there may be changes to be made on the technique in listening to the network

in Process-II we have tested, keeping the relevant port open and returning the container that the demand is connected to when it is received. One of these is packet listening and changing, and OS shell management in the user space with the net filter hooking technique at the core level, the other one is performing the listening and recovering process at this point by manipulating the docker-proxy.

Reducing resource utilization is a general, common, and widespread purpose. Each idea, laboratory and field applications, and developments bring us closer to the ideal point. It is indicated that the idea we have suggested can be useful in field applications. We have supported this with the laboratory tests we have conducted.

References

[1] Le VD, Neff MM, Stewart RV, Kelley R, Fritzinger E, Dascalu SM, Harris FC. Microservice-based architecture for the NRDC. IEEE 13th International Conference on Industrial Informatics, Cambridge, UK, 2015; 1659-1664.

[2] Engler DR, Kaashoek MF, O'Toole J. Exokernel: an operating system architecture for application-level resource management. ACM symposium on Operating systems principles, New York, USA 1995; 251-266.

[3] Esposito C, Castiglione A, Choo KR. Challenges in delivering software in the cloud as microservices. IEEE Cloud Computing, New York, USA, 2016; 3: 10-14.

[4] Bratterud A, Walla A, Haugerud H, Engelstad PE, Begnum K. IncludeOS: A minimal, resource efficient unikernel for cloud services. IEEE 7th International Conference on Cloud Computing Technology and Science, Vancouver, Canada, 2015; 250-257.

[5] Xavier B, Ferreto T, Jersak L. Time provisioning evaluation of KVM, Docker and Unikernels in a Cloud Platform. 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Cartagena, Colombia, 2016; 277-280.

[6] Rahman M, Gao J. A reusable automated acceptance testing architecture for microservices in behavior-driven development. IEEE Symposium on Service-Oriented System Engineering, San Francisco Bay, CA, USA, 2015; 277-280.

[7] Kang H, Le M, Tao S. Container and microservice driven design for cloud infrastructure DevOps. IEEE International Conference on Cloud Engineering, Berlin, Germany, 2016; 202-211.

[8] Villamizar M, Garces O, Castro H, Verano M, Salamanca L, Casallas R,Gil S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 10th IEEE International Conference on Computing Colombian Conference, Cartagena, Colombia, 2015; 583-590.

[9] Butzin B, Golatowski F, Timmermann D. Microservices approach for the internet of things. IEEE 21st International Conference on Emerging Technologies and Factory Automation, Berlin, Germany, 2016.

[10] Barais O, Bourcier J, Bromberg Y, Dion C. Towards microservices architecture to transcode videos in the large at low costs. International Conference on Telecommunications and Multimedia, Heraklion, Crete, Greece, 2016.

[11] Inagaki T, Ueda Y, Ohara M. Container management as emerging workload for operating systems. IEEE International Symposium on Workload Characterization, Providence, RI, USA, 2016; 65-74.

[12] Villamizar M, Garcés O, Ochoa L, Castro H, Salamanca L, Verano M, Casallas R, Gil S, Valencia C, Zambrano A, Lang M. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Cartagena, Colombia, 2016; 179-182.

[13] Malavalli D, Sathappan S. Scalable microservice based architecture for enabling DMTF profiles. 11th International Conference on Network and Service Management, Washington, DC, USA, 2015; 428-432.

[14] Ueda T, Nakaike T, Ohara M. Workload characterization for microservices. IEEE International Symposium on Workload Characterization, Providence, RI, USA, 2016; 85-94.

[15] Jaramillo D, Nguyen D, Smart R. Leveraging microservices architecture by using Docker technology. SoutheastCon 2016.

[16] Hai NH. A dynamic link speed mechanism for energy saving interconnection networks. PhD, The University of Barcelona, Barcelona, Spain, 2014.

[17] Beloglazov A. Energy-efficient management of virtual machines in data centers for cloud computing. PhD, The University of Melbourne, Australia, 2013.