Autonomic Vertical Elasticity of Docker Containers with ElasticDocker

Yahya Al-Dhuraibi[*][§], Fawaz Paraiso[*], Nabil Djarallah[§], Philippe Merle[*]
[*]Inria / University of Lille, [§]Scalair
Email: yaldhuraibi@scalair.fr, fawaz.paraiso@inria.fr, ndjarallah@scalair.fr, philippe.merle@inria.fr

***Abstract*—Elasticity is the key feature of cloud computing to scale computing resources according to application workloads timely. In the literature as well as in industrial products, much attention was given to the elasticity of virtual machines, but much less to the elasticity of containers. However, containers are the new trend for packaging and deploying microservices-based applications. Moreover, most of approaches focus on horizontal elasticity, fewer works address vertical elasticity.**

In this paper, we propose ElasticDocker, the first system powering vertical elasticity of Docker containers autonomously. Based on the well-known IBM's autonomic computing MAPE-K principles, ElasticDocker scales up and down both CPU and memory assigned to each container according to the application workload. As vertical elasticity is limited to the host machine capacity, ElasticDocker does container live migration when there is no enough resources on the hosting machine. Our experiments show that ElasticDocker helps to reduce expenses for container customers, make better resource utilization for container providers, and improve Quality of Experience for application end-users. In addition, based on the observed migration performance metrics, the experiments reveal a high efficient live migration technique. As compared to horizontal elasticity, ElasticDocker outperforms Kubernetes elasticity by 37.63%.

***Keywords*—Cloud computing; Vertical elasticity; Container; Docker; Live migration**

I. Introduction

Elasticity is one of the key characteristics of cloud computing, which leads to its widespread adoption. Elasticity is defined as the ability to adaptively and timely scale computing resources in order to meet varying workload demands [1], [2]. There are two types of elasticity: horizontal and vertical [1], [3]. Horizontal elasticity consists in adding or removing instances of computing resources associated to an application. Horizontal elasticity is also known as replication of resources. Vertical elasticity consists in increasing or decreasing characteristics of computing resources, such as CPU time, cores, memory, and network bandwidth. Vertical elasticity is also known as resizing of resources. Both elasticities are driven by the variation of workload demands, such as the request response time or the number of end-users. In the scientific literature but also in industry practices, most of proposed approaches focus on horizontal elasticity but few addresses vertical elasticity.

Virtualization techniques are the keystone of elasticity in cloud computing and consist to virtualize the actual physical resources – e.g., CPU, storage, network – as virtual resources such as virtual machines (VMs), virtual storages, virtual networks. Numerous works proposed various cloud elasticity handling mechanisms for VMs [3], [4], [5], [6]. However, with the advent of Docker [7], containers are becoming the new trend for packaging and deploying microservices-based applications [8]. Since Docker provides more flexibility, scalability, and resource efficiency than VMs [9], [10], [11], it becomes popular to bundle applications and their libraries in lightweight Linux containers and offers them to the public via the cloud. Then, Docker containers have gained a widespread deployment in cloud infrastructures such as in Amazon EC2 Container Service, Google Container Engine, Docker Datacenter, Rackspace. But compared to VMs, there are only few works that deal with elasticity of Docker containers: [12], [13], [14] focus on automatic horizontal elasticity, [15] address manual vertical elasticity, [16] supports migration. To the best of our knowledge, there is no related work that handles vertical elasticity of containers autonomously.

The main contribution of this paper is to present ElasticDocker: the first system powering vertical elasticity of Docker containers autonomously. Based on the well-known IBM's autonomic computing MAPE-K principles [17], ElasticDocker scales up and down both CPU and memory assigned to each container when the application workload grows up and down, respectively. This approach modifies resource limits directly in the Linux control groups (cgroups) associated to Docker containers. Vertical elasticity is limited to host machine capacity as it cannot provision more resources when all the host machine resources are already allocated to containers. Therefore, in this work, we use live migration to handle this limit. Live migration is the process of moving a container in its executing state from source to target host. Container migration takes place when resizing is no longer possible on the host machine. ElasticDocker uses Checkpoint/Restore In Userspace (CRIU) [18] to implement the concept of container live migration. The approach then evaluated by running experiments using Graylog[1] and RUBiS [19] applications. These experiments show that ElasticDocker helps to improve performance and Quality of Experience (QoE) for application end-users, reduce expenses for container customers, and make better resource utilization for container providers. Our experiments also show that ElasticDocker outperforms Kubernetes autoScaling [20] by 37.63%. We have

[1]https://www.graylog.org

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.

also evaluated the efficiency of containers live migration for different Docker images, and the observed migration, checkup and restore times are small and negligible.

The remainder of this paper is organized as follows. Section II describes the motivation for vertical elasticity of Docker containers. Section III provides the technical background on Docker. Section IV presents our ElasticDocker approach to scale up/down and migrate Docker containers. Section V describes the experimental setup to evaluate ElasticDocker and discusses obtained results. After that, we discuss this approach and its limits in Section VI. We present related works in Section VII. Section VIII concludes the paper and hightlights research perspectives.

II. Motivation

A. Resource over-provisioning and de-provisioning

The load of cloud applications varies along with time. Diverse applications have different requirements. Therefore as shown in Fig. 1, maintaining sufficient resources to meet workload burst and peak requirements can be costly. Conversely, maintaining minimum or medium computing resources can not meet workload's peak requirements, and cause bad performance and Service Level Objective (SLO) violations. Autonomic cloud elasticity permits to adaptively and timely scale up/down resources according to the actual workload.

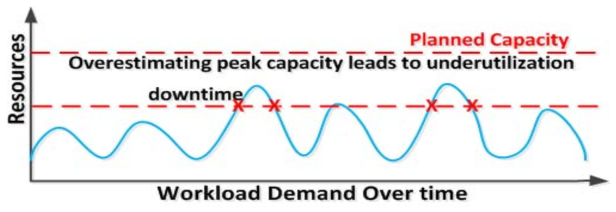

Fig. 1. Resource over-/under-provisioning

B. Vertical elasticity

Elasticity is defined as the ability to adapt resources on the fly to handle load variation. There are two types of elasticity: horizontal elasticity and vertical elasticity. Horizontal elasticity requires more support from the application, so that it can be decomposed into instances. Recently, most attention has been given to horizontal elasticity management. Vertical elasticity is limited due to the fact it can not scale outside the resources provided by a single physical machine and then introduces a single point of failures. However, vertical elasticity is better when there are enough available resources. Vertical elasticity has the following characteristics:

- Vertical elasticity is fine-grained scaling while it permits to add/remove real units of resources.
- Vertical elasticity is applicable to any application, it also eliminates the overhead caused by booting instances in horizontal elasticity while horizontal elasticity is only applicable to applications that can be replicated or decomposed.
- Vertical elasticity does not need running additional machines such as load balancers or replicated instances.
- Vertical elasticity guarantees that the sessions of the application are not interrupted when scaling.
- Some applications such as MATLAB, AutoCAD do not support horizontal elasticity. They are not replicable by design. These applications are composed of components and their interconnections. These components can not be elastic, which means that it is impossible to create several instances of the same component.
- Since horizontal elasticity consists in replicating the application on different machines, some applications such as vSphere and DataCore require additional licenses for each replica. These licenses could be very expensive, while only one license is required in vertical elasticity.

Vertical elasticity maintains better performance, less SLO violations, and higher throughput. Vertical elasticity increases the performance because the elasticity controller just increases the capacity of the same instance. In horizontal scaling, the elasticity controller can add/remove instances, which impacts the application performance. This fact is verified by using the queuing theory [6], and the proof was demonstrated in [6]. Horizontal elasticity could result to have many small instances and modeled as M/M/1, while vertical elasticity controls one instance by varying its capacity and modeled as M/M/c in the queuing theory, where c is the number of CPUs. After solving the corresponding equations for each model, [6] founded that the response and waiting time in vertical elasticity is much less than that of horizontal elasticity for the same workload input. Due to the limit of space, we do not duplicate the equations and examples to show this fact. However in Section V-B5, we show experimentally that ElasticDocker vertical elasticity outperforms Kubernetes horizontal elasticity by 37,63%.

C. Containers vs VMs

Docker containers are a new lightweight virtualization technology. We outline this technology in details in Section III. Docker requires an autonomic elastic system in order to avoid the problems of over-provisioning and under-provisioning. We present here motivations towards Docker vertical elasticity.

- Containers consume low resource because they share resources with the hosting operating system, which makes them more efficient. Therefore, we can deploy more containers than VMs on a physical machine [9].
- Containers result in equal or better performance than VMs [11].
- Containers have small image size, therefore, time of generating, distributing and downloading images is short, in addition they require less storage space [9].

ElasticDocker proposes an approach that manages autonomous vertical provisioning and deprovisioning of Docker containers on the host machine and migrates them if there is no enough resources.

III. Background

This section gives a brief introduction about Docker technologies in order to facilitate the understanding of our work.

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.

It also elaborates Docker image filesystem and CRIU, the concept behind ElasticDocker container live migration.

A. Docker technology

Docker is a lightweight virtualization technology that allows to package an application with all of its dependencies into one standardized unit for software deployment. Docker uses a client-server architecture. It consists of three main components, *Docker client*, *Docker host* and *Docker registry*. Docker host represents the hosting machine on which Docker daemon and containers run. Docker daemon is an essential part of Docker architecture, it is responsible for building, running, and distributing the containers. The interactions with Docker daemon are done through Docker client. Docker client is the user interface to Docker. Docker registry is the service responsible for storing and distributing images.

B. Resource management of Docker

Docker containers use *namespaces* to isolate resources, and *cgroups* to manage and monitor resources. Runc is a lightweight tool that runs the containers (container runtime). Runc uses libcontainer and LXC drivers which are Linux container libraries that enable and abstract interactions with Linux kernel facilities such as cgroups and namespaces to create, control and manage containers.

1) Control groups (cgroups): Docker container relies on cgroups to group processes running in the container. Cgroups allow to manage the resources of a container such as CPU, memory, and network. Cgroups not only track and manage groups of processes but also expose metrics about CPU, memory and I/O block usage. Cgroups or subsystems are exposed through pseudo-filesystems. The filesystem can be found under */sys/fs/cgroups* in recent Linux distributions. Under this directory, we can access multiple subsystems in which we can control and monitor memory, CPU cores, CPU time, I/O, etc. In these files, Docker resources can be configured to have hard or soft limits. When soft limit is configured, the container can use all resources on the host machine. However, there are other parameters that can be controlled here such as *CPU shares* that determine a relative proportional weight that the container can access the CPU. Hard limits are set to give the container a specified amount of resources, ElasticDocker changes these limits dynamically according to the container workload.

By default, Docker sets no limits, then a Docker container can use all the available resources on the host machine. It can use all the CPU cores as well as memory. The CPU access is scheduled either by using Completely Fair Scheduler (CFS) or by using Real-Time Scheduler (RTS) [21]. In CFS, CPU time is divided proportionately between Docker containers. On the other hand, RTS provides a way to specify hard limits on Docker containers or what is referred to as ceiling enforcement. Our elastic approach is integrated with RTS in order to make it elastic. Limits on Docker containers are set, our ElasticDocker scales up or down resources according to demand. Once there is no limit set, it is hard to predict how much CPU time a Docker container will be allowed to utilize. In addition, as indicated, by default Docker can use all resources on the host machine, there is no control how much resources will be used by that container (customer) as many containers (customers) can coexist on the same hosting machine. A customer may not afford to pay for such uncontrolled amount of resource. Moreover, it will be complicated for the provider to manage the customer billing system, e.g., providers usually bill the customer by instance (VM) or according to the number of CPUs, not by a partial usage of many CPUs.

C. Docker image filesystem

Docker builds and stores images, these images are then used to create containers. Docker image consists of a list of read-only layers that represent filesystem differences. The layers are stacked on top of each other to form the base of a container's root filesystem. When container is created, a new writable layer called container layer is added on top of the underlying layers or stack. Docker supports many storage drivers such as aufs, btrfs, overlay, etc. In our case, AUFS is used. AUFS is a unification filesystem, which means it takes many multiple directories (image layers), stacks them on top of each other, provides a single unified view through a single mount point. Docker hosts the filesystem as a directory under */var/lib/docker/*.

D. Checkpoint/Restore In Userspace (CRIU)

CRIU is a Linux functionality that allows to checkpoint/restore processes. It has the ability to save the state of a running application so that it can latter resume its execution from the time of the checkpoint. Our migration approach for Docker containers with CRIU can be divided in two steps, checkpoint and restore, in addition to copy process if the dumped files do not reside on a shared file system between the source and target host [22].

IV. Elastic Docker Approach

A. System design

We designed ElasticDocker to automatically scale up/down Docker containers to adjust resources to the varying workload. ElasticDocker provisions and deprovisions Docker resources vertically on the host machine. As shown in Fig. 2, ElasticDocker consists of monitoring system and elasticity controllers. Elasticity controllers can adjust memory, vCPU cores, and CPU fraction according to the workload demand. These components are presented in the below sections. ElasticDocker adheres to use the well-known autonomic MAPE-K loop [17]. MAPE-K is an autonomic computing architecture introduced by IBM, it consists of four phases: Monitor, Analyze, Plan, Execute, and Knowledge. In our system, firstly, different Docker metrics are continuously monitored. In the second analysis phase, thresholds are calculated based on the monitored metrics, then the decision and plan to scale up/down is taken accordingly. Finally, we implement the decisions to adjust Docker resources according to the need.

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.

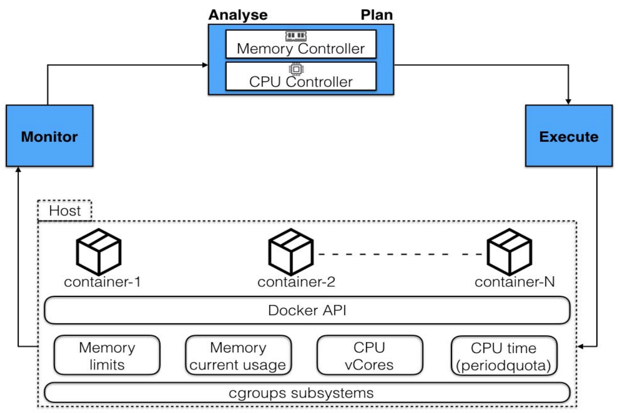

Fig. 2. ElasticDocker architecture

B. Monitoring system

Our monitoring system collects most resource utilization and limits of Docker containers by interrogating directly with Docker cgroup subsystems while it uses Docker RESTful API to check CPU usage. The system continuously monitors the memory subsystem in cgroup to check the memory current size assigned to each Docker container as well the current memory utilization. Similarly, we check the CPU parameters such as the number of vCores and time. In Section IV-C, we highlight how to control the CPU time, thus allowing us to control CPU percentage assigned to each Docker container. In the experimentation, we have noticed that the CPU and memory utilization values are sometimes fluctuating rapidly, which could be due to the nature of workload. Therefore, to avoid this oscillation, we measure CPU and memory utilization each 4 seconds on an interval of 16 seconds (as shown in Table I), then we take the average value as the current utilization of CPU and memory.

C. Elasticity controller

The elastic controller adjusts memory, CPU time, vCPU cores according to workloads. ElasticDocker modifies directly the cgroups filesystem of Docker containers to implement scaling up/down decisions. The memory is monitored and then based on its usage and thresholds, ElasticDocker increases or decreases its size. The upper threshold is set to 90%, and the lower threshold is set to 70%. The values shown in Table I are chosen following [6], [23] which are based on real-world best practices, in addition we tried different values, and selecting the best values that lead to less response time. Once the memory utilization is greater than the upper threshold, ElasticDocker adds 256MB to its size. In the deprovisioning state, the memory size is decreased by 128MB. We decrease memory size by small amount in the scaling down process because the applications are sensitive to the memory resource, and this could lead to interrupt the functionality of the application. In addition, after each scaling decision, ElasticDocker waits a specific period of time (breath duration). Breath duration is a period of time left to give the system a chance to reach a stable state after each scaling decision. As shown in Table I, we set two breath durations, breath-up and breath-down. Breath-up is time to wait after each scaling up decision. We chose these small values because the application adapts quickly to the container change, we have noticed that the application functions normally after these time periods. Breath-up is smaller than breath-down to allow the system to scale-up rapidly to cope with burst workload. Breath-down is larger than breath-up duration in order to avoid uncertain scaling down which could cause degradation in the performance of the system.

TABLE I
ElasticDocker parameters

Parameter or metric	value
monitored metrics	CPU utilization, CPU time, vCPUs, Memory utilization, Memory limit
measurement period/interval	4 seconds/16 seconds
breath-up/breath-down	10 seconds/ 20 seconds
upper threshold	90%
lower threshold	70%
CPU increase/decrease ratios	±10% of CPU time or ±1 vCPUs
mem. increase/decrease ratios	+256MB/-128MB

ElasticDocker also controls CPU time (percentage) and number of vCPUs assigned to each Docker container. As we have seen in Section III, we can control CPU time by changing CFS parameters, namely *cpu.cfs_period_us* and *cpu.cfs_quota_us*, we refer to them simply as period and quota. For example if period is set to 100000 and quota set to 10000, Docker can use 10% of CPU percentage (i.e, 0.01 second of each 0.1 second), if a Docker container has two vCPUs and $period = 100000$ and $quota = 200000$, this means the Docker container can completely use the two vCPUs. ElasticDocker increases quota or CPU percentage in function of CPU usage and dynamic thresholds. For example, if a container has 10% of CPU time, the threshold will be 9.5, 20% of CPU time, threshold will be 19% and so on. Once a container has used all the CPU time, new core will be added. Upper threshold to add a vCPU core is 90%. Lower threshold is set to 70%, if CPU usage is less than lower threshold, vcores will be removed. However let's suppose that a Docker container has three vCPUs cores and $quota/period = 250000/100000$ and CPU usage is less than 70%, the scaling down decision is taken according to the following condition: $cpu\_usage < 70\%$ and $no\_vCPUs > 1$ and $quota < period * (no\_vCPUs - 1)$, where no_vCPUs is the number of vCPUs allocated to the container. Similar to memory, breath durations are set for CPU resizing. It is worth noting that ElasticDocker takes in consideration the available resources on the host machine, and the allocated resources of other Docker containers on the host upon each scaling up/down decision.

D. Container live migration

Many containers generally reside on the same host machine. Therefore, when one Docker container continues to ask for more resources, if there is no more resources on the host,

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.

live migration will take place for that Docker container. The container will be migrated to another host machine. The process of live migration consists of four main steps as shown in Fig. 3. Firstly, the filesystem differences of the container image layers in */aufs/diff/* will be transferred. There are many directories in */aufs/diff/* representing image layers, so we tar and send these layers to the destination host. Secondly, The container process will be pre-dumped. The container is still running after the pre-dump. The objective of the pre-dump is to minimize the migration downtime. The pre-dumped images are compressed using LZ4 compression in a TAR file and sent to the destination. We perform several pre-dump iterations, each pre-dump generates a set of pre-dump images, which contain memory changes after previous pre-dump. This reduces the amount of data dumped on the dump stage. Thirdly, we proceed to dump the container state, the dump process will be rapid because it only takes the memory that has changed after the last pre-dump. On the destination host, we will restore the container to the same memory state on the source host thank to CRIU.

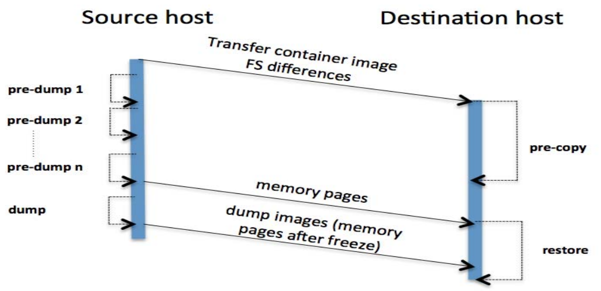

Fig. 3. Migration procedure based on CRIU

V. Experiments and Evaluation

A. Experimental setup

We evaluated our approach with respect to the performance and end-user QoE, customer cost, resource utilization, migration efficiency and then we compare ElasticDocker versus Kubernetes autoscaling. We performed all our experiments on Scalair[2] infrastructure inside VMs. Scalair is a private cloud provider company. The VM on which containers run has 7vCPUs with 5GB RAM and Centos 7.2 OS. We use Graylog, a powerful log management and analysis platform. We chose this application because it consumes a lot of resources. Since Graylog centralizes and aggregates all log files from different sources, it can suddenly get overloaded, and that requires a lot of attention from the providers to adjust resources according to the need particularly at peak's times. Graylog is widely implemented in the industry and it is based on four main components Graylog Server, Elasticsearch, MongoDB and Web Interface. Graylog Server is a worker that receives and processes messages, and communicates with all other components. Its performance is CPU dependent. Elasticsearch is a search server to store all of the logs/messages. It depends on the RAM, disk and also CPU. MongoDB stores read-only configuration and metadata and does not experience much load. Web Interface is the user interface. We run three containers, the first one is for the Graylog server version 2.0.0 and Web interface 2.0.0 while the second and third are for Elasticsearch version 2.3.3 and MongoDB version 3.2.6 respectively. We use Docker version 1.9.1 and Docker Compose version 1.7.1. Docker Compose is used to define and run the different components of Graylog in the containers. We also installed Ubuntu 14.04 and Httperf[3] version 0.9.0-2build1 on the second VM. It has 2vCPUs with 4GB RAM. We also set scripts to send log messages and overload Graylog server on this VM. The two VMs are on different VLANs. httperf generates requests to query the Graylog server.

B. Evaluation and results

1) Performance and end-users QoE: First, we investigate the impact of our proposed approach on the performance and end-user QoE and compare the results between Docker and ElasticDocker. Therefore, we run our experiments to evaluate the performance of the Graylog application in two cases: i) with Docker only, and ii) with ElasticDocker system. We generate different workloads using httperf to query and search information from Graylog via its REST API. Fig. 4 shows the results of comparison experiments of average response time (RT) between Docker and ElasticDocker. As shown in Fig. 4, we have different request rates 10 req./sec., 50 req/sec., etc. The more the number of requests are, the more the RT increases. When the number of requests are between 10 and 50 requests per second, there is no significant difference in average RT between the two cases. However, when the number of requests increases and requires more resources, ElasticDocker reacts to provision resources accordingly, therefore the RT decreases and the performance increases. The blue and red bars in Fig. 4 indicate Docker and ElasticDocker performances, respectively. ElasticDocker increases performance by 74,56%.

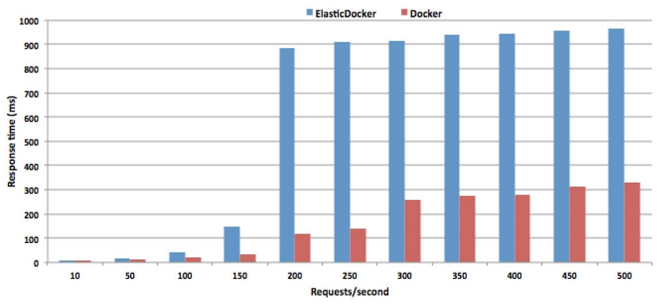

Fig. 4. Graylog response time with Docker vs ElasticDocker

2) Customers' expenses reduction: In this part of experiment, we study the impact of ElasticDocker on the cost. To put stress on Graylog and ElasticDocker containers,

[2]http://www.scalair.fr

[3]https://en.wikipedia.org/wiki/Httperf

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.

we generated workloads with a random rates that arrive to more than 1000 req./sec. by scripts on the second VM. The workloads are syslog and Graylog Extended Log Format (GELF) logs. The logs generated sent to be processed by Graylog server and stored in the Elasticsearch container. At the beginning, each Graylog, Elasticsearch and MongoDB Docker has 1vCPU and 1130MB, 384MB and 128MB respectively. Fig. 5 shows the vCPU and memory size for each Docker over time. Graylog and Elasticsearch containers consume CPU and memory while MongoDB has only 1vCPU and 128MB of RAM because it just stores metadata. To facilitate the understanding of this experiment, let us consider the following simplified pricing model used by cloud providers:

$$cost = \sum_{n=1}^{n} cpu(t_n, t_{n-1}) * (t_n, t_{n-1}) * p+ \\ mem(t_n, t_{n-1}) * (t_n, t_{n-1}) * p' \tag{1}$$

where $cpu(t_n, t_{n-1})$ is the number of vCPUs in a time period between (t_n, t_{n-1}), p is the price for each vCPU in time period (t_n, t_{n-1}), $mem(t_n, t_{n-1})$ is the memory size in a time period (t_n, t_{n-1}), p' price for the memory. To ease the understanding of the customer costs, let us consider the vCPU consumption of Graylog containers, referring to Fig. 5 and Table II, the cost according to Equation(1) is $cost = 1*(t1, t0)*p+2(t2, t1)*p+3(t3, t2)*p+4(t4, t3)*p+3(t5, t4)*p+2(t6, t5)*p+1(t7, t6)*p+2(t7, t8)*p+1(tx, t8)*p = 28, 63p$

Without ElasticDocker, the customer will reserve fixed resources all the time, in our case, it could be 4vCPUs for graylog server and then the cost will be $4*(t8, t0)*p = 66, 04p$, (from Fig. 5, the time periods (t_n, t_{n-1}) are translated against a fixed interval). From these results, ElasticDocker reduces cost by 56.65%. It is shown that ElasticDocker significantly decreases the charge for the customers.

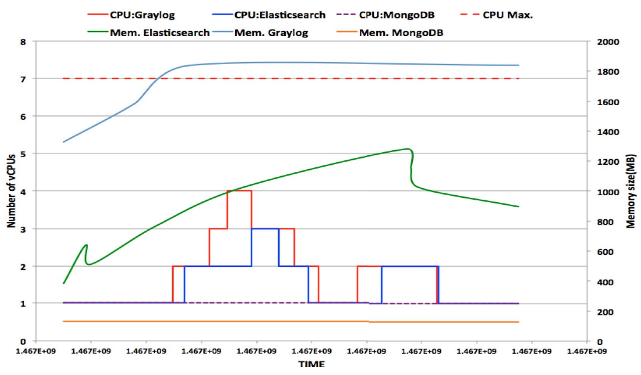

Fig. 5. CPU and memory consumption of Graylog, Elasticsearch and MongoDB containers

TABLE II
vCPUs vs. Time for Docker1

Time	t0	t1	t2	t3	t4	t5	t6	t7	t8
number of vCPUs	1	2	3	4	3	2	1	2	1

3) *Optimal utilization for resources:* We evaluated the resource utilization in a single host. Fig. 6 shows that ElasticDocker can maintain a better utilization of resources. Without ElasticDocker, the resources reserved while they are idle. For example in our experiment, to avoid services interruption in Graylog container, 4vCPU must be reserved, however the need for these vcores is for small period only, after that they are idle and would not be possible to run three containers on the same host. ElasticDocker reserves and frees resources according to the charge.

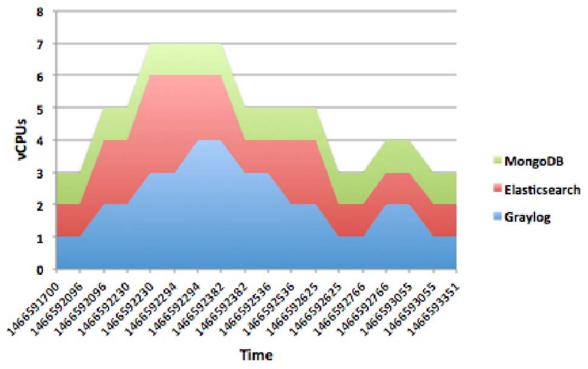

Fig. 6. Resource Utilization

4) *Docker live migration efficiency:* In this section we migrate different Docker applications from one host to another. We used Docker version 1.9.0-dev and CRIU version 2.2 on both hosts. We evaluate ElasticDocker live migration technique with respect to many parameters such as checkpoint, restore time, etc. We migrate different applications as shown in Table III. The simple workload generator tool (stress) is used. We have checked the application state, for example, we set a counter in the source container and we check the value of the counter once the container is migrated. It shows that the first value on the counter in the migrated container is the value following the last value in the source container. In addition, the container nginx with PHP-FPM pushes incremental counter to a web page, after migration, the operation continues except a suspension for few seconds. Table III shows different migration indicators. Pre-dump time is the time duration during the

TABLE III
Migration Performance Indicators

Application	Image size (MB)	Pre-dump time (s)	Dump time (s)	Restore time (s)	Migration total time (s)	Migration down-time (s)
Nginx	181.5	0.02022	0.2077	3.505	4.28	0.547
Apache	193.3	0.0807	0.196	3.19	5.18	1.712

pre-dump process of the container. Dump time is the time duration during the final dump of the container. Migration and restore times are the periods during the whole process of migration and time to restore the application respectively. Downtime is the time of interruption when the container process is frozen during the final dump process.

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.

Table III shows the different migration indicators and their values measured during migration of the application containers. The values are small especially the downtime which is the most important in the live migration. Downtime causes a negative impact particularly on stateful applications that are too sensitive for TCP sessions. It is worth noting that there are other factors that could impact the migration such as network bandwidth.

5) Vertical elasticity vs horizontal elasticity: We compared ElasticDocker with Kubernetes, i.e., vertical elasticity versus horizontal elasticity. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. To achieve the experiment, we use RUBiS and Kubernetes version v1.2.0. RUBiS is a well-known Web application benchmark that simulates an online auction site. Our deployment of RUBiS on Kubernetes uses three tiers: a loadbalancer (Kubernetes service performs this role), a scalable pool of JBoss application servers, and a MySQL database. Kubernetes platform is deployed on 4 nodes running CentOS Linux 7.2. RUBiS is deployed in two containers, in addition to a loadbalancer. Then, we set the Kubernetes Horizontal Pod Autoscaling (HPA) to scale RUBiS containers based on rules-based threshold. We use the same thresholds used in ElasticDocker. We generate three workloads (low, medium and high) using Apache HTTP server benchmarking tool (ab) and the total execution time is measured for each workload. We then generate the same workloads to ElasticDocker (running RUBiS on the machine described in Section V-A) and the total execution time is measured as shown in Fig. 7.

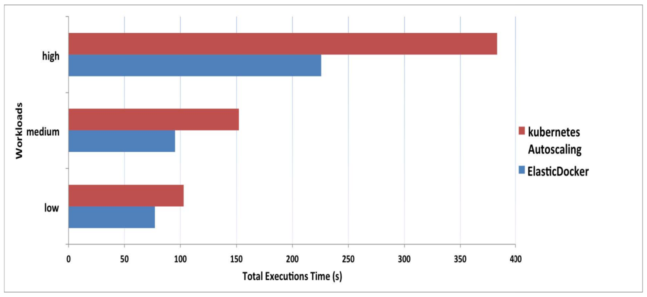

Fig. 7. Comparison between Kubernetes' elasticity capabilities and ElasticDocker

Based on the analysis of these results we concluded the following findings: (i) the average total execution time for the three workload in ElasticDocker is 132.6 seconds, while the average total execution time in Kubernetes is 212.62 seconds. (ii) ElasticDocker outperforms Kubernetes horizontal elasticity by 37.63%. (iii) This confirms the analysis in Section II-B that vertical elasticity is more efficient than horizontal elasticity. Due to the limit of space, we do not include the resource usage, however, Kubernetes uses more resources, even when the workloads are terminated, the scaled containers takes more than 5 minutes to start to scale in. This could be due to the slow monitoring component in Kubernetes (Heapster).

VI. Discussions

Our system uses reactive approach based on the threshold-based rules to perform elastic actions. While threshold based rules are the most popular auto-scaling technique, even in commercial systems, setting-up suitable thresholds is very tricky, and may lead to instability of the system. Therefore, in the performance section of our experimentation, we have tried different thresholds, i.e., 90, 85, 80, 70, 60 and different breath or cooling durations. After that, we have chosen the best values as shown in Table I, which yields to best performance (lower response time). The ideal will be to use machine learning. The improvement of QoE by ElasticDocker is not without cost. In fact, ElasticDocker allocates more resources to overcome the workload burst. The system proposed allows to control CPU and memory. Docker allows to control the numbers of operations per second (ops) or amount of data, bits per second (bps) on specific devices connected to Docker container. However, there is no direct method particularly in AUFS filesystem to adjust quota or amount of disk available to a specific container. So, it is difficult to resize the disk storage at runtime. Although it is true that Docker provides the option *–storage-opt* to resize storage, this option is applied to the daemon level not to a single container. For the migration, we simply migrate the container which requires more resources when there is no sufficient resources to reply its demand. The idea is to improve this mechanism in order to have more intelligent system that decides which container to migrate: the one which currently requires more resources, or the one which has less activity, etc.

VII. Related Work

Elasticity is a major research challenge in the field of cloud computing. Several different approaches have been proposed for the elasticity at VMs level such as [3], [5], [6]. However, with the appearance of Docker containers and their widespread popularity among cloud providers, some researches are dedicated to this field. Kukade et al. [12] proposed a design for developing and deploying microservices applications into Docker containers. The elasticity is achieved by constantly monitoring memory load and number of requests by an external master. Once certain thresholds are reached, the master node invokes scaling agent. The scaling agent permits to horizontally spin in or out the container instances. Haldy et al. [13] worked on container live migration technique and proposed a framework called MultiBox. MultiBox is a mean for creating and migrating containers among different cloud providers. It makes use of Linux cgroups to create containers and migrate the source containers to those newly created ones. Hoenisch et al. [16] proposed a control architecture that adjusts VMs and containers provisioning. DoCloud [14] is a horizontal elastic cloud platform based on Docker. It permits to add or remove Docker containers to adapt Web application resource requirements. In DoCloud, a hybrid elasticity controller is proposed that uses proactive and reactive model to scale out

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.

and proactive model to scale down. Monsalve et al. [24] proposed an approach that controls CPU shares of a container, this approach uses CFS scheduling mode. Nowadays, Docker can use all the CPU shares if there is not concurrency by other containers. Paraiso et al. [15] proposed a tool to ensure the deployability and the management of Docker containers. It allows synchronization between the designed containers and those deployed. In addition, it allows to manually decrease and increase the size of container resource. These works either handle horizontal elasticity or manual vertical elasticity. [23] proposed horizontal and vertical autoscaling technique based on a discrete-time feedback controller for VMs and containers. This approach is limited to Web applications. In addition, the application requirements and metadata must be precisely defined to enable the system to work. It also adds overhead by inserting agents for each container and VM. Kubernetes and Docker Swarm are orchestration tools that permit container horizontal elasticity, they allow also to set limit on containers during their initial creation. Our proposed approach supports automatic vertical elasticity for Docker containers and live migration if there is no enough resources.

VIII. Conclusion & Perspectives

ElasticDocker is an elasticity controller to dynamically grow or shrink Docker resources according to workloads. If there is no more resources on the host machine, we migrate the container to another host. This migration technique is based on CRIU functionality in Linux systems. Through experimental evaluations we have shown that ElasticDocker significantly increases end-user QoE and performance, reduces customer's expenses and makes a better resource utilization. In addition, the migration downtime is very small and the application state is maintained. The experiments also demonstrate that fine-grained adaptation capabilities of ElasticDocker greatly improve performance when compared to Kubernetes autoscaling.

We envision extending this work in several ways: (i) coordinating container elasticity and hosting VM elasticity, (ii) enhancing ElasticDocker with features to support predictive approaches in order to anticipate workloads and rapidly scale up resources, (iii) extending the proposed platform to complement vertical elasticity with horizontal elasticity (what we name "diagonal" elasticity).

Acknowledgment

This work is supported by the OCCIware research and development project (http://www.occiware.org) funded by French Programme d'Investissements d'Avenir (PIA). Likewise, this work is also funded by Scalair company (http://www.scalair.fr).

References

[1] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza, "Elasticity in Cloud Computing: a Survey," *Annals of Telecommunications*, pp. 1–21, 2015.

[2] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 23–27.

[3] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth, "Towards faster response time models for vertical elasticity," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, ser. UCC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 560–565.

[4] F. Paraiso, P. Merle, and L. Seinturier, "soCloud: a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds," *Computing*, vol. 98, no. 5, pp. 539–565, 2016.

[5] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth, "Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints," in *2015 International Conference on Cloud and Autonomic Computing (ICCAC)*, Sept 2015, pp. 69–80.

[6] W. Dawoud, I. Takouna, and C. Meinel, "Elastic virtual machine for fine-grained cloud resource provisioning," in *Global Trends in Computing and Communication Systems*. Springer, 2012, pp. 11–25.

[7] Docker Inc., "What is Docker?" Web site https://www.docker.com/what-docker.

[8] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2015.

[9] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud," *Advanced Science and Technology Letters*, vol. 66, pp. 105–111, 2014.

[10] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*. IEEE, 2012, pp. 423–430.

[11] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 171–172.

[12] P. P. Kukade and G. Kale, "Auto-Scaling of Micro-Services Using Containerization," *International Journal of Science and Research (IJSR)*, pp. 1960–1964, 2013.

[13] J. Hadley, Y. El Khatib, G. Blair, and U. Roedig, *MultiBox: lightweight containers for vendor-independent multi-cloud deployments*, ser. Communications in Computer and Information Science. Springer Verlag, 11 2015, pp. 79–90.

[14] C. Kan, "DoCloud: An elastic cloud platform for Web applications based on Docker," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, Jan. 2016, pp. 478–483.

[15] F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-Driven Management of Docker Containers," in *9th IEEE International Conference on Cloud Computing (CLOUD)*, San Francisco, United States, Jun. 2016, p. 8.

[16] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, "Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers," in *Service-Oriented Computing*. Springer, 2015, pp. 316–323.

[17] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[18] Yang, Chen, "Checkpoint and Restoration of Micro-service in Docker Containers," 2015.

[19] RUBiS, Website http://www.cs.nyu.edu/~totok/professional/software/rubis/rubis.html.

[20] Kubernetes, Website https://kubernetes.io.

[21] Red Hat, Inc., Web site https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-cpu.html.

[22] Checkpoint/Restore, Website https://criu.org/Checkpoint/Restore.

[23] Baresi, Luciano and Guinea, Sam and Leva, Alberto and Quattrocchi, Giovanni, "A Discrete-time Feedback Controller for Containerized Cloud Applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 217–228.

[24] J. Monsalve, A. Landwehr, and M. Taufer, "Dynamic CPU Resource Allocation in Containerized Cloud Environments," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 535–536.

Authorized licensed use limited to: PES University Bengaluru. Downloaded on February 13,2023 at 08:22:57 UTC from IEEE Xplore. Restrictions apply.