CoMA: Resource Monitoring of Docker Containers

Lara Lorna Jiménez, Miguel Gómez Simón, Olov Schelén, Johan Kristiansson, Kåre Synnes and Christer Åhlund

Dept. of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Luleå, Sweden

Keywords: Docker, Containers, Containerization, OS-level Virtualization, Operating System Level Virtualization, Virtualization, Resource Monitoring, Cloud Computing, Data Centers, Ganglia, sFlow, Linux, Open-source, Virtual Machines.

Abstract: This research paper presents CoMA, a Container Monitoring Agent, that oversees resource consumption of operating system level virtualization platforms, primarily targeting container-based platforms such as Docker. The core contribution is CoMA, together with a quantitative evaluation verifying the validity of the measurements reported by the agent for three metrics: CPU, memory and block I/O. The proof-of-concept is implemented for Docker-based systems and consists of CoMA, the Ganglia Monitoring System and the Host sFlow agent. This research is in line with the rising trend of container adoption which is due to the resource efficiency and ease of deployment. These characteristics have set containers in a position to topple virtual machines as the reigning virtualization technology in data centers.

1 INTRODUCTION

Traditionally, virtual machines (VMs) have been the underlying infrastructure for cloud computing services(Ye et al., 2010). Virtualization techniques spawned from the need to use resources more efficiently and allow for rapid provisioning. Native virtualization (Type I) (VMware Inc, 2007b)(VMware Inc, 2007a) is the standard type of virtualization behind cloud services. There are several established platforms that offer this type of virtualization such as, Xen hypervisor (Barham et al., 2003), Linux Kernel Virtual Machine (KVM) (Tafa et al., 2011) and VMware (VMware Inc, 2007a). Full-virtualization, Para-virtualization and Hardware-assisted virtualization are different techniques that attempt to enhance the effectiveness of VMs, with varying degrees of success and certain tradeoffs. However, none of these techniques are on par with today's expectations in the cloud computing industry. It has been demonstrated that VMs introduce a significant overhead that does not allow for an optimized use of resources (Xu et al., 2014). The unfulfilled potential for improvement of VMs is where OS-level virtualization comes in.

OS-level virtualization has become popular in recent years by virtue of its resource efficiency. This light-weight type of virtualization executes processes quasi-natively (Felter et al., 2014),(Xavier et al., 2013). On top of a shared Linux kernel, several of what are generally referred to as "containers" run a series of processes in different user spaces (Elena Reshetova, 2014). In layman's terms, OS-level virtualization generates virtualized instances of kernel resources, whereas hypervisors virtualize the hardware. Moreover, containers run directly on top of an operating system, whereas VMs need to run their OS on top of a hypervisor which creates a performance overhead (Xu et al., 2014). The downside of containers is that they must execute a similar OS to the one that is hosting the containers. There are various implementations of OS-level virtualization, with differences in isolation, security, flexibility, structure and implemented functionalities. Each one of these solutions is oriented towards different use cases. For example, *chroot() jails* (Elena Reshetova, 2014) are used to sandbox applications and Linux containers (LXC) [1] are used to create application containers.

In March 2013, the Docker platform was released as an open-source project based on LXC and a year later, the environment was moved from LXC to libcontainer [2]. Docker is based on the principles of containerization, allowing for an easy deployment of applications within software containers as a result of its innovative and unique architecture (Felter et al.,

[1] https://linuxcontainers.org/

[2] http://www.infoq.com/news/2013/03/Docker

DOI: 10.5220/0005448001450154
In *Proceedings of the 5th International Conference on Cloud Computing and Services Science* (CLOSER-2015), pages 145-154
ISBN: 978-989-758-104-5

2014). Docker implements certain features that were missing from OS-level virtualization. It bundles the application and all its dependencies into a single object, which can then be executed in another Docker-enabled machine. This assures an identical execution environment regardless of the underlying hardware or OS. The creation of applications in Docker is firmly rooted in the concept of versioning (Docker Inc, 2014b). Modifications of an application are committed as deltas (Docker Inc, 2014c), which allows roll backs to be supported and the differences to previous application versions to be inspected. This is an exceptional method of providing a reliable environment for developers. Furthermore, Docker promotes the concept of reusability, since any object that is developed can be re-used and serve as a "base image" to create some other component. Another essential aspect of Docker is that it provides developers with a tool to automatically build a container from their source code.

The main difference between a Docker container and a VM is that while each VM has its own OS, dependencies and applications running within it, a Docker container can share an OS image across multiple containers. In essence, a container only holds the dependencies and applications that have to be run within them. For example, assuming a group of containers were making use of the same OS image, the OS would be common to all containers and not be duplicated contrary to the case of a VM topology.

Docker has become the flagship in the containerization technology arena since its release (Felter et al., 2014) (Docker Inc, 2013). This open-source project has gained much notoriety in the field of cloud computing, where major cloud platforms and companies (e.g. Google, IBM, Microsoft, AWS, Rackspace, Red Hat, VMware) are backing it up. These companies are integrating Docker into their own infrastructures and they are collaborating in Docker's development. Recently, a few alternatives to Docker have cropped up, such as Rocket [3], Flockport [4] and Spoonium [5].

An adequate monitoring of the pool of resources is an essential aspect of a cloud computing infrastructure. The monitoring of resources leads to improved scalability, better placement of resources, failure detection and prevention, and maintenance of architectural consistency, among others. This is relevant for VMs, and it is just as applicable to OS-level virtualization. Out of this need to monitor containers, within the paradigm of OS-level virtualization platforms, the following research questions have been addressed in this paper:

- How could an OS-level virtualization platform be monitored to obtain relevant information concerning images and containers?

  This paper presents an investigation of this issue as well as an implementation of a Container Monitoring Agent.

- Is the resource usage information about the running containers reported by our Container Monitoring Agent valid?

  This paper details a quantitative evaluation of the validity of the measurements collected by the proposed Container Monitoring Agent.

The rest of the paper is organized as follows. Section 2 presents state-of-the-art research related to the monitoring of containers and virtual machines. Section 3 explains the different components of the monitoring system architecture. Section 4 presents the results obtained. Section 5 discusses the research questions. Finally, Section 6 presents the conclusions of the paper and discusses the possibilities for future work.

2 RELATED WORK

There is an active interest in industry and in research to build monitoring solutions (Ranjan and Tai, 2014). In (Kutare et al., 2010) the term monalytics is coined. Monalytics refers to a deployment of a dynamically configurable monitoring and analytics tool for large-scale data centers, targeting the XEN hypervisor. One of the monalytics' topologies defined, matches the architecture chosen for the monitoring solution provided in this paper. The research of (Meng et al., 2012) is centered on providing a state monitoring framework that analyzes and mitigates the impact of messaging dynamics. This technique ensures the trustworthiness of the measurements collected by a distributed large-scale monitoring tool on XEN-based virtual machines. Most existing monitoring research for data center management target virtual machines. To the best of our knowledge, at the time of writing this paper, there were no research papers on the topic of monitoring the Docker platform.

When this monitoring solution for Docker was developed, there were a lack of open-source implementations to monitor Docker. However, recently, several other monitoring systems for Docker have appeared.

Datadog agent (Datadog Inc, 2014) is an open-source tool developed by *Datadog Ink* which monitors Docker containers by installing an agent within the host where the Docker platform is running. It is

[3]https://coreos.com/blog/rocket/
[4]http://www.flockport.com/start/
[5]https://spoon.net/docs

an agent-based system which requires metrics to be pushed to the *Datadog cloud* thereby making the task of monitoring entirely dependent on *Datadog's cloud*. Unlike the *Datadog agent*, the monitoring agent for the Docker platform presented in this paper is not only open-source, but also independent of any particular collector. It can be integrated within different monitoring architectures after the proper configuration is done.

cAdvisor is an open-source project created by Google Inc (Bryan Lee, 2014) to monitor their own *lmctfy* containers (Google Inc, 2014). Support to monitor the Docker platform was later added to it. Therefore, this monitoring tool provides Docker metrics, which are shown in real time but are not stored for more than one minute. This feature may be useful to test container performance but, due to the small data time frame displayed, it is not possible to get a historical of the metrics collected.

The Host sFlow agent (InMon Inc, 2014) is an open-source tool to monitor the resource consumption of a host. It has recently incorporated the option to monitor the Docker platform, making it a viable open-source monitoring solution for Docker. However, this agent adheres to the sFlow standard [6], which enforces constraints on the information it is able to send as there is no dedicated sFlow structure for Docker. By contrast, the solution provided in this paper does not have limitations on the metrics that can be obtained. The monitoring agent presented here can be modified to select which metrics, out of all the available, to monitor.

As shown above, only the solution proposed in this paper manages to provide a monitoring module for Docker that is open-source, does not adhere to a particular collector or monitoring framework provided some configuration is done, and allows for the selection of a certain subset of metrics from all the available ones.

3 SYSTEM ARCHITECTURE

To monitor Docker containers, three separate modules could be employed: our Container Monitoring Agent (CoMA), a metrics' collector and a host monitoring agent. The solution proposed in this paper, to create a distributed Docker monitoring system consists of: CoMA, the Ganglia Monitoring System and the Host sFlow agent.

In Figure 1, a possible layout of the proposed monitoring architecture is shown. This figure represents three clusters. Each one of the monitored hosts represented in Figure 1 have the structure presented in Figure 2.

[6]http://www.sflow.org/developers/specifications.php

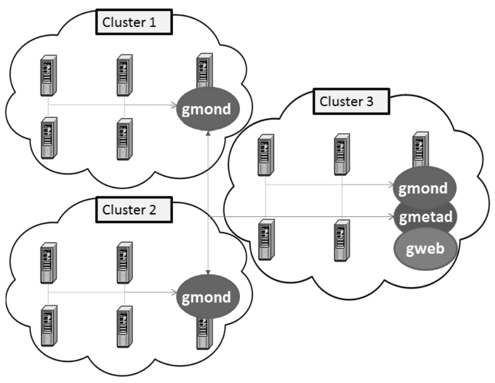


Figure 1: Cloud topology.

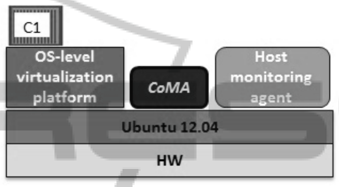


Figure 2: General overview of the host.

3.1 CoMA: CONTAINER MONITORING AGENT

We have developed CoMA, the agent that monitors containers of OS-level virtualization platforms such as Docker. CoMA retrieves information about the containers and images in the Docker platform. It also tracks the CPU, memory and block I/O resources being consumed by the running containers. CoMA can be accessed as an open-source project at *https://github.com/laraljj/CoMA*.

The collection of Docker metrics in CoMA is accomplished via two modules. One module makes requests to the platform's Remote API (Docker Inc, 2014a) to collect data about the containers and images. These include: the number of Go routines being executed by the Docker platform; the images that have been created in each host and information about these (e.g. size and virtual size); the number of containers that have been created and from which image they have been built; the status of all the containers in the host (i.e. whether they are running or stopped).

The second module obtains information about resource usage and resource limitations of the running containers. These metrics are obtained by accessing the control groups (cgroups) feature of the Linux kernel (Paul Menage, 2014), which accounts and sets limits for system resources within different subsystems. The container resources monitored by this module include CPU, memory and block I/O. The number of measurements recorded by CoMA vary according to the number of containers that have been deployed. For the Docker platform as a whole, there are 18 CPU related metrics. Per container, 18 CPU related metrics, 24 memory related metrics, and 100 block I/O related metrics are measured. This means that when a single container is running, there are a total of 160 measurements available, 142 of these are container specific and 18 of these are related to the Docker platform itself. Therefore, when two containers are running there will be 302 measurements, 142 measurements for one container, 142 measurements for the other container and 18 measurements for the Docker platform. The metrics that are being reported by CoMA can be selected according to the needs of each specific deployment, so that only the values of those metrics are dispatched to the collector, instead of all of them.

3.2 Complementary Components

3.2.1 The Ganglia Monitoring System

The Ganglia Monitoring System (Massie et al., 2012) is an open-source distributed monitoring platform to monitor near real-time performance metrics of computer networks. Its design is aimed at monitoring federations of clusters. The Ganglia Monitoring System was selected as collector due to its capacity to scale, its distributed architecture and because it supports the data collection of the Host sFlow agent. However, in the interest of fitting the requirements of a different system, a stand-alone collector could be used instead.

The system is comprised of three different units: *gmond*, *gmetad* and *gweb*. These daemons are self-contained. Each one is able to run without the intervention of the other two daemons. However, architecturally they are built to cooperate with each other.

Gmond is a daemon that collects and sends metrics from the host where it is running to *gmetad*. This is not a traditional monitoring agent, as it does not sit passively waiting for a poller to give the order to retrieve metrics. *Gmond* is able to collect metric values on its own, but *gmond*'s built-in metrics' collection may be replaced by the Host sFlow agent. This setup implies that, for the architecture chosen, *gmond* acts as in-between software layer for the Host sFlow agent and *gmetad*.

Gmetad is a daemon running a simplified version of a poller, since all the intelligence of metric retrieval lays, in our case, with the Host sFlow agent and *gmond*. *Gmetad* must be made aware of from which *gmonds* to poll the metrics. *Gmetad* obtains the whole metric dump from each *gmond*, at its own time interval, and stores this information using the RRD-tool (i.e. in "round robin" databases).

Gweb is Ganglia's visualization UI. It allows for an easy and powerful visualization of the measurements collected, mostly in the form of graphs. New graphs combining any number of metrics can be generated, allowing the visualization of metrics to be customized depending on individual needs.

3.2.2 The Host sFlow Agent

The Host sFlow agent was selected as the host monitoring agent to track the resources consumed by the OS in the host running the OS-level virtualization platform. Monitoring resources both at the host level and at the virtualization platform level makes it possible to compare the values of the metrics for soundness checks, tracking problems at both levels.

The Host sFlow agent may retrieve information from within an OS running on bare metal or from within the hypervisor if the aim is to monitor virtual machines. This agent can be run in multiple OSs and hypervisors. The agent itself obtains the same metrics as *gmond*'s built-in collector does. The difference between these two solutions is that the sFlow standard, used by the Host sFlow agent to relay metrics, is considerably more efficient than *gmond*. This is because each sFlow datagram carries multiple metric values, which reduces the number of datagrams that need to be sent over the network. For example, monitoring 1,000 servers with *gmond* would create the same network overhead as 30,000 servers with the sFlow protocol (Massie et al., 2012). The sFlow protocol's efficiency justifies the usage of the Host sFlow agent in this monitoring system.

4 EVALUATION

The primary evaluation objective is to assess the validity of the values of the metrics collected with CoMA. Validity in this context means to establish, for the metrics reported by CoMA, whether the measured values reflect the real values. Given the numerous metrics reported about CPU, memory and block I/O, a small subset of these metrics has been selected for the evaluation. This validity assessment is carried out on user CPU utilization, system CPU utilization, memory usage and number of bytes written to disk. User CPU utilization and system CPU utilization refer to the percentage of CPU that is employed to execute

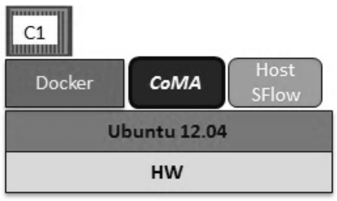


Figure 3: Scenario 1, no workload (baseline).

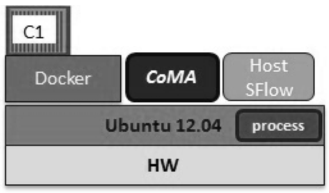


Figure 4: Scenario 2, workload on host OS.

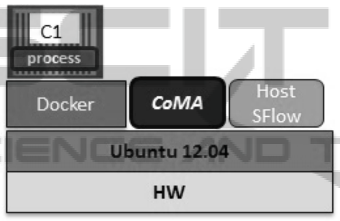


Figure 5: Scenario 3, workload in container.

code in user space and in kernel space, respectively. It should be noted that for all tests, Ubuntu 12.04 LTS and the Docker platform 1.3.1 have been run on the same Dell T7570 computer. This computer runs an Intel Pentium D 2.80 GHz (800 MHz) and 2 GB of RAM at 533 MHz.

The evaluation presented collects for each host: host-specific metrics (reported by the Host sFlow agent) and the metrics from the Docker platform (reported by CoMA). The purpose of collecting the values for both sets of metrics was to compare and contrast the host's resource consumption against the resource consumption of the Docker platform, that is, against the resource consumption of the collection of containers and images within the host. The objective of this comparison is to offer a reliable overview of the system from a resource usage perspective. Comparing both sets of metrics, a system administrator can pinpoint the origin of a particular misbehavior by determining if the issue is due to the Docker platform (i.e. a specific container) or due to some other problem within the host but independent of the containers.

4.1 Validity of CPU and Memory Measurements

Three different scenarios have been set up to assess

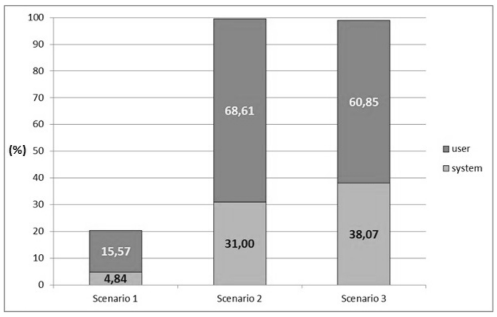
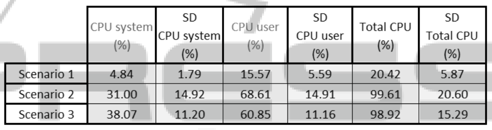


Figure 6: Host CPU utilization reported by the Host sFlow agent.

Table 1: Host CPU utilization reported per scenario. Total CPU is the aggregation of user CPU and system CPU. Standard Deviation (SD).



	CPU system (%)	SD CPU system (%)	CPU user (%)	SD CPU user (%)	Total CPU (%)	SD Total CPU (%)
Scenario 1	4.84	1.79	15.57	5.59	20.42	5.87
Scenario 2	31.00	14.92	68.61	14.91	99.61	20.60
Scenario 3	38.07	11.20	60.85	11.16	98.92	15.29

the collected values of the memory and CPU-utilization metrics. For all scenarios, 6 rounds, each one of 30 minutes have been run. Scenario 1 (Figure 3) presents a baseline of the CPU and memory usage while the OS is executing Docker, which runs a single unlimited container executing */bin/bash*, the Host sFlow Agent and CoMA. The container was not actively used during this scenario. Scenario 2 (Figure 4) is set up like Scenario 1, except for the fact that a workload generator was executed natively in the OS. This means that the process did not run containerized. *Stress-ng* [7] has been used to generate load on both CPU and memory. In order to create load on the CPU, two workers were launched so that there would be a worker per core. Each CPU worker executed *sqtr(rand())* to generate load. To stress the memory, five workers were started on anonymous mmap, each one of these workers was set to 420MB. Scenario 3 (Figure 5) has been laid out exactly like Scenario 2, the only difference being that the *stress-ng* processes were executed containerized.

4.1.1 CPU: A Single Container

The data obtained from each scenario were processed. Figure 6 shows user CPU, system CPU and total CPU utilization reported at the host level for each scenario. Figure 7 displays CPU utilization pertaining to the process or processes running within that single container deployed in the three scenarios.

In order to determine that the measurements of the CPU metrics collected with CoMA are valid, the data

[7]http://kernel.ubuntu.com/ cking/stress-ng/

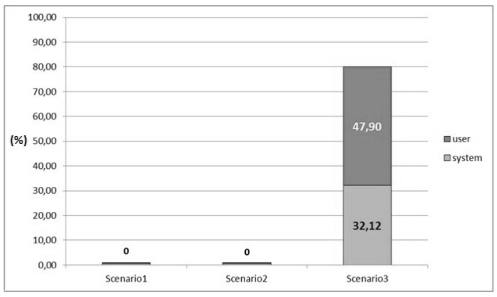


Figure 7: Container CPU utilization reported by CoMA.

Table 2: Container CPU utilization reported by CoMA per scenario. Note that there are no processes running in Scenario 1 and Scenario 2.

	CPU system (%)	SD CPU system (%)	CPU user (%)	SD CPU user (%)	Total CPU (%)	SD Total CPU (%)
Scenario 1	-	-	-	-	-	-
Scenario 2	-	-	-	-	-	-
Scenario 3	32.12	9.44	47.90	9.77	78.41	2.78

obtained from Scenario 3, which can be visualized in Figure 6 and 7, has been compared. The total CPU of Scenario 1 (Table 1), aggregated to the total CPU of the container reported by CoMA in Scenario 3 (Table 2), should resemble the total CPU reported by the host in Scenario 3 (Table 1). The aggregation of those first two values (20.42% and 78.41%) results in a total CPU of 98.83% and a standard deviation of 6.5. The total CPU utilization of the whole host in Scenario 3 is 98.92% with a standard deviation of 15.29. These results verify that the values of the CPU metrics gathered by CoMA are valid.

The CPU utilization data retrieved from this evaluation allows for other noteworthy observations to be made. In Figure 6 and Table 1 a small difference of 0.69% can be ascertained, between running the *stress-ng* processes natively (Scenario 2) or containerized (Scenario 3). The disparity that exists between these two scenarios is due to several reasons. First, the intrinsic variable nature of the data collected has a direct impact on the results attained. However, its irregularity is acceptable as the standard deviations calculated demonstrate, since these are reasonable and valid for these data. Second, the *stress-ng* processes themselves may be accountable for a certain variation.

It can also be noticed that there seems to be a tendency in the way these *stress-ng* processes are executed. When *stress-ng* was run within a container more system CPU utilization was accounted for compared to when *stress-ng* was run natively. The effect is the exact opposite when it comes to user CPU utilization, as can be visualized in Figure 6. This last observation has been verified by computing the correlation coefficient between system CPU utilization and user CPU utilization. A nearly perfect negative correlation of -0.99 was obtained for Scenario 2 and -0.98 for Scenario 3.

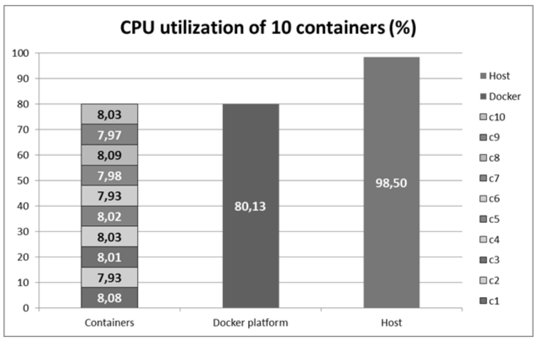


Figure 8: Total CPU utilization of 10 containers, where each container runs the same process generating a symmetric CPU load across all containers. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

Table 3: Total CPU utilization and standard deviations (SD) for Figure 8.

Total CPU utilization (%)						
Host		Docker platform		Containers		
Total CPU	SD Total CPU	Total CPU	SD Total CPU	Name of container	Total CPU	SD Total CPU
98.50	3.30	80.13	1.10	c1	8.08	0.23
				c2	7.93	0.18
				c3	8.01	0.26
				c4	8.03	0.22
				c5	8.02	0.38
				c6	7.93	0.30
				c7	7.98	0.24
				c8	8.09	0.28
				c9	7.97	0.20
				c10	8.03	0.28

4.1.2 CPU: Multiple Containers

These scenarios prove that the CPU utilization retrieved by CoMA, of one container, is valid. However, whether the agent is able to properly report the CPU metrics for multiple simultaneously running containers should also be demonstrated. For this purpose, two tests were carried out. For the first test, ten containers ran the exact same *stress-ng* process to generate load on the CPU with two workers, one per core. In accordance with the default process scheduler in Linux, the Completely Fair Scheduler (CFS) [8], the expected outcome of this test is for each container to employ a similar portion of the total CPU. As it can be observed in Figure 8 and Table 3, each container is granted an average of around 8% of the CPU. The aggregation of each container's total CPU utilization adds up to 80.07% which almost matches

[8]http://lwn.net/Articles/230501/

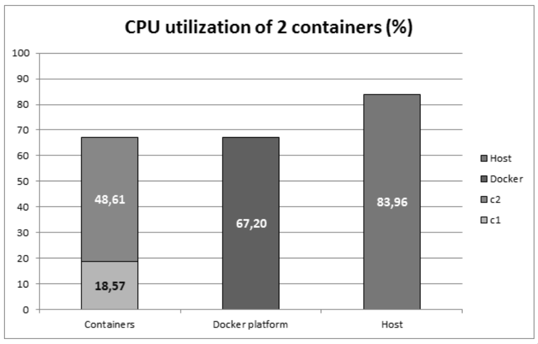


Figure 9: Total CPU utilization of 2 containers with asymmetric processes running in each container. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

Table 4: Total CPU utilization and standard deviations (SD) for Figure 9.

Total CPU utilization (%)						
Host		Docker platform		Containers		
Total CPU	SD Total CPU	Total CPU	SD Total CPU	Name of container	Total CPU	SD Total CPU
83.96	1.46	67.20	0.65	c1	18.62	0.28
				c2	48.61	0.59

the total CPU utilization of the whole Docker platform (80.13%) as reported by CoMA.

This test shows that each container reports its own set of CPU measurements independently and is able to do so effectively. However, a different test was carried out to verify this by running asymmetric processes in two containers. Each container ran the *stress-ng* process with different settings so as to generate an uneven load across the two containers. As represented by Figure 9 and Table 4, CoMA reported just that. A container used 48.61% of the total CPU whilst the other container employed 18.57% of the total CPU. Both containers together used 67.18%, which resembles the value (67.20 %) reported by CoMA of the total CPU utilization of the Docker platform.

4.1.3 Memory: A Single Container

The memory data captured for all scenarios is displayed in Figure 10. It should be mentioned that there is a greater fluctuation in the memory-reported values than in the CPU values. This phenomenon is due to the manner in which Linux manages its memory. The memory management methodology applied by Linux varies according to the needs of the OS at any given time. This adds another layer of complexity when analyzing the metrics collected. The host's memory usage in Scenario 1 (360.73MB) aggregated to the container's memory usage reported by CoMA in

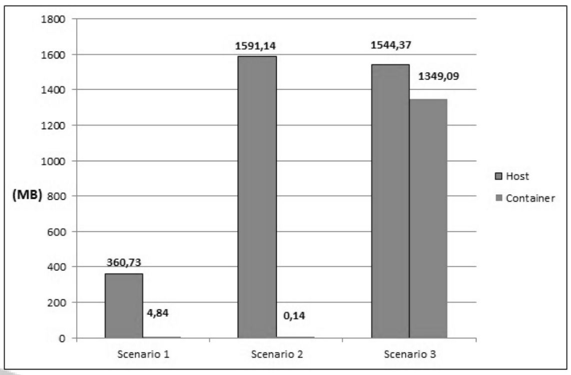


Figure 10: A comparison of the memory usage reported by the host and the container per scenario.

Table 5: Memory usage and standard deviations (SD) for Figure 10.

	Host-reported memory		Container-reported memory	
	Memory usage (MB)	SD Memory usage (MB)	Memory usage (MB)	SD Memory usage (MB)
Scenario 1	360.73	11.35	4.84	0.02
Scenario 2	1591.14	224.43	0.14	0.17
Scenario 3	1544.37	209.47	1349.09	184.57

Scenario 3 (1349.09MB), should be somewhat similar to the host's memory consumption in Scenario 3 (1544.37MB). In this case there is a difference of around 165MB. As it has been explained before, this discrepancy is caused by the memory management enforced by Linux, as well as by the error introduced in the averaging process of the results.

4.1.4 Memory: Multiple Containers

Much like it happens with CPU, the previous scenarios establish that the memory metrics provided by CoMA are valid for a single container. The two CPU tests performed with multiple simultaneously running containers, were also carried out for memory. As it can be observed in Figure 11 and Table 6, when the same process is running in each container, the memory usage value presented by CoMA per container has a greater variability than that observed in the same test for CPU utilization. As it has been previously explained, these fluctuations are due to the changeable nature of how memory is managed by the OS. However, each container's memory usage is close to 152MB. The aggregated memory usage of all 10 containers adds up to 1519.92MB. The Host sFlow agent reports a memory usage of 1773.62MB for the whole host during this test. The difference of 253.70MB between these last two values, represents the memory being employed by the OS to run non-containerized processes. A second test, where two containers were

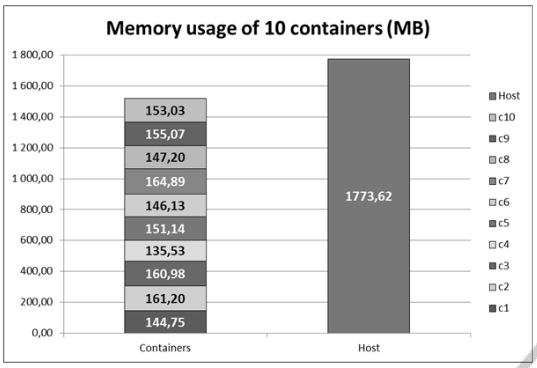


Figure 11: Memory usage of 10 containers, where each container runs the same process. This generates a symmetric memory use across all containers. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

Table 6: Memory usage and standard deviations (SD) for Figure 11.

Memory (MB)				
Host		Containers		
Memory usage	SD Memory usage	Name of container	Memory usage	SD Memory usage
1773.62	59.57	c1	144.75	20.60
		c2	161.20	20.87
		c3	160.98	11.56
		c4	135.53	15.96
		c5	151.14	11.86
		c6	146.13	14.54
		c7	164.89	14.50
		c8	147.20	12.54
		c9	155.07	20.74
		c10	153.03	27.69

configured to make a disparate use of memory was also carried out. Figure 12 and Table 7 reflect the results obtained, which are consistent with the values gathered when running a symmetric memory load on 10 containers.

4.2 Validity of block I/O Measurements

To evaluate whether the block I/O measurements gathered by CoMA were solid, the I/O tool *fio* [9] was used to write 1000MB directly to the ext4 filesystem mounted by the host by making use of the –v flag (Docker Inc, 2014d) on the container. In order to achieve this, *fio* was configured to initiate five workers, each worker performing random write operations of 200 MB in the shared folder between the host and the container.

The test of writing 1000MB to disk was executed at 12:00 and it finished by 12:07. As Figure 13 shows,

[9]http://freecode.com/projects/fio

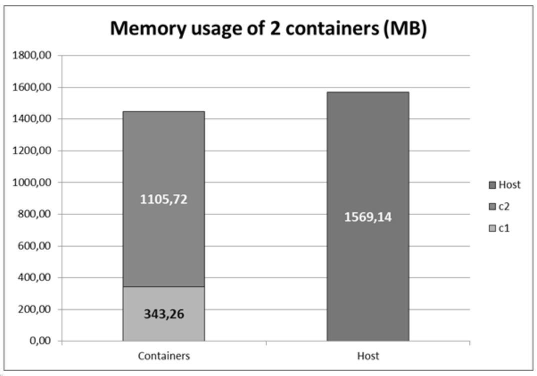


Figure 12: Memory usage of 2 containers with asymmetric processes running in each container. CoMA reports the values for the containers and for the Docker platform. The Host sFlow agent reports the values for the host.

Table 7: Memory usage and standard deviations (SD) for Figure 12.

Memory (MB)				
Host		Containers		
Memory usage	SD Memory usage	Name of container	Memory usage	SD Memory usage
1569.14	194.24	c1	343.26	65.01
		c2	1105.72	130.16

exactly 1000MB were reported to have been written during that time.

A separate test was created, following the same principle previously explained, to write to disk from three simultaneously running containers. *Fio* was configured for each container with a disparate number of workers and file sizes. The first container spawned two workers, each of which had to write 300MB to the shared folder. The second container initiated three workers, each with a file size of 250MB. The third container started five workers, where each worker had to write 200MB to disk. For each container, the number of bytes that CoMA reported were written to disk was exactly right, down to the last byte. The first container took 20 minutes to write the 600MB to disk. The second and third container took around 16 minutes to write 750MB and 1000MB to disk, respectively. The time taken for each container to complete the task of writing these files to memory is closely linked to the number of workers running and the number of containers writing to disk.

5 DISCUSSION

This section discusses CoMA as well as the evaluation results obtained in terms of the research questions proposed.

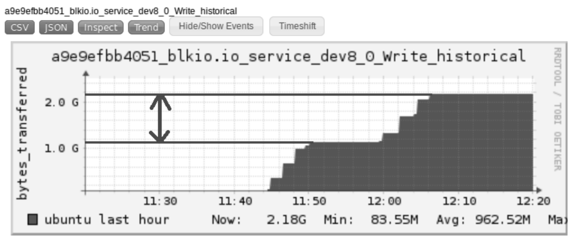


Figure 13: Bytes written to disk by container.

How could an OS-level virtualization platform be monitored to obtain relevant information concerning images and containers?

CoMA retrieves information about the usage of CPU, memory and block I/O of running containers from the Linux kernel's control groups. It also fetches the data concerning images and containers using Docker's Remote API.

Is the resource usage information about the running containers reported by our Container Monitoring Agent valid?

The evaluation provides validation across the following three blocks of metrics: CPU, memory and block I/O. The most complex metric to validate was the memory usage of a container. This is due to the way memory is managed in an OS, which causes the memory usage baseline in Scenario 1 to account a slightly overestimated usage. The authenticity of all the measurements that can be collected with CoMA could not be tested because of the number of metrics that CoMA is able to gather. Nevertheless, since the values are being reported by the Linux kernel, assessing at least one metric from each group of metrics is sufficient to establish the validity of CoMA. It should be mentioned that CoMA can be modified to only dispatch a subset of desired metrics.

CoMA's CPU utilization is dependent on the test-bed that has been set up. This means that CoMA's resource usage is contingent on the hardware that has been employed, the number of containers that had been deployed, the number of metrics being sent and the sampling rate set for CoMA. This last value can be configured to obtain measurements closer or further away from real-time monitoring, depending on the requirements. There are certain tradeoffs in the selection of the sampling rate. A higher sampling rate would mean obtaining more accurate measurements in terms of time, but more resources would be used in order to monitor the platform. It is worth mentioning that CoMA itself consumes around 15.25% of CPU with a standard deviation of 5.87 for the specific test-bed presented in the evaluation section. This number may seem high, but it is relative to the hardware being employed. An Intel Pentium D 2.8GHz and 2GB RAM at 533MHz was used in this case. Had conventional cloud computing hardware been used, this percentage would be much lower. Moreover, in this test-bed all available metrics are collected, if fewer of them were collected the percentage of CPU used would decrease. It should also be mentioned that the monitoring solution itself shall be optimized so as to minimize its impact.

It has been previously mentioned that CoMA could be employed to monitor similar OS-level virtualization platforms. For this to happen, said OS-level virtualization platform would have to account resource usage in a similar fashion to Docker, i.e. using the Linux kernel's control groups. However, the information pertaining to the containers and images that is collected through Docker's Remote API, is specific to the Docker platform itself.

6 CONCLUSION AND FUTURE WORK

Monitoring the resource consumption of OS-level virtualization platforms such as Docker, is important to prevent system failures or to identify application misbehavior. CoMA, the Container Monitoring Agent presented in this paper, reports valid measurements as shown by our evaluation. It currently tracks CPU utilization, memory usage and block I/O of running containers. CoMA could be configured to gather a subset of the available metrics to suit the monitoring needs of a particular system or application. This paper has presented a possible implementation solution of CoMA to build a distributed and scalable monitoring framework, using the open-source projects Ganglia and the Host sFlow agent.

It would be positive to monitor the network usage of the containers, since this feature has not yet been implemented in CoMA. Moreover, establishing thresholds on certain metrics collected by CoMA to trigger alarms or actions would be beneficial. Also, assessing CoMA's behavior when numerous containers are deployed on commonly used hardware in data centers is required. This would be a proper test-bed to gauge CoMA's performance in a realistic cloud computing scenario.

Another area for further research would be to employ machine learning techniques on the values collected, to maximize resource usage by modifying each container's resource constraints based on the needs of the running containers. There is also the possibility of applying data analytics on the information captured by CoMA to build an autonomous system for container placement within a cloud or across clouds.

There are new and upcoming OS-level virtualizations platforms that could rival Docker, such as Rocket. CoMA could be also employed and evaluated with these recent virtualization platforms.

REFERENCES

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177.

Bryan Lee (Accessed: 2014). cAdvisor monitoring tool. http://blog.tutum.co/2014/08/07/using-cadvisor-to-monitor-docker-containers/.

Datadog Inc (Accessed: 2014). Docker-ize Datadog with agent containers . https://www.datadoghq.com/2014/06/docker-ize-datadog/.

Docker Inc (2013). What is Docker technology ? https://www.docker.com/whatisdocker/.

Docker Inc (Accessed: 2014a). Docker remote API. https://docs. docker.com/ reference/api/docker_remote_api/.

Docker Inc (Accessed: 2014b). Docker working with LXC. https://docs.docker.com/faq/.

Docker Inc (Accessed: 2014c). File sytem architecture of the Docker platform . https://docs.docker.com/terms/layer/.

Docker Inc (Accessed 2014d). Volume system with Docker. https://docs.docker.com/userguide/dockervolumes/.

Elena Reshetova, Janne Karhunen, T. N. N. A. (2014). Security of os-level virtualization technologies.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2014). An updated performance comparison of virtual machines and linux containers. *technology*, 28:32.

Google Inc (Accessed: 2014). lmctfy: Let Me Contain That For You . https://github.com/google/lmctfy.

InMon Inc (Accessed: 2014). HostsFlow monitoring tool . http://host-sflow.sourceforge.net/.

Kutare, M., Eisenhauer, G., Wang, C., Schwan, K., Talwar, V., and Wolf, M. (2010). Monalytics: Online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 141–150, New York, NY, USA. ACM.

Massie, M., Li, B., Nicholes, B., Vuksan, V., Alexander, R., Buchbinder, J., Costa, F., Dean, A., Josephsen, D., Phaal, P., and Pocock, D. (2012). *Monitoring with Ganglia*. O'Reilly Media, Inc., 1st edition.

Meng, S., Iyengar, A. K., Rouvellou, I. M., Liu, L., Lee, K., Palanisamy, B., and Tang, Y. (2012). Reliable state monitoring in cloud datacenters. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 951–958, Washington, DC, USA. IEEE Computer Society.

Paul Menage (Accessed: 2014). Control Groups (cgroups) Documentation . https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. Available since: 2004.

Ranjan, R., B. R. L. P. H. A. and Tai, S. (2014). A note on software tools and techniques for monitoring and prediction of cloud services softw: Pract. exper., 44: 771–775.

Tafa, I., Beqiri, E., Paci, H., Kajo, E., and Xhuvani, A. (2011). The evaluation of transfer time, cpu consumption and memory utilization in xen-pv, xen-hvm, openvz, kvm-fv and kvm-pv hypervisors using ftp and http approaches. In *Intelligent Networking and Collaborative Systems (INCoS), 2011 Third International Conference on*, pages 502–507.

VMware Inc (2007a). Understanding Full Virtualization, Paravirtualization and Hardware Assist. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf. Accessed: 2014 (white paper).

VMware Inc (2007b). Virtualization Overview. http://www.vmware.com/ pdf/ virtualization.pdf. Accessed: 2014 (white paper).

Xavier, M., Neves, M., Rossi, F., Ferreto, T., Lange, T., and De Rose, C. (2013). Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240.

Xu, F., Liu, F., Jin, H., and Vasilakos, A. (2014). Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31.

Ye, K., Huang, D., Jiang, X., Chen, H., and Wu, S. (2010). Virtual machine based energy-efficient data center architecture for cloud computing: A performance perspective. *IEEE-ACM International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing*, 0:171–178.