A Performance Comparison of Cloud-Based Container Orchestration Tools

Conference Paper · November 2019

DOI: 10.1109/ICBK.2019.00033

Yao Pan
University of Melbourne
3 PUBLICATIONS **36** CITATIONS
SEE PROFILE

Francisco Brasileiro
Universidade Federal de Campina Grande (UFCG)
240 PUBLICATIONS **3,185** CITATIONS
SEE PROFILE

Glenn Tesla Jayaputera
University of Melbourne
25 PUBLICATIONS **218** CITATIONS
SEE PROFILE

A Performance Comparison of Cloud-based Container Orchestration Tools

Yao Pan[1], Ian Chen[1], Francisco Brasileiro[2], Glenn Jayaputera[1], Richard O. Sinnott[1]

[1]School of Computing and Information Systems, University of Melbourne, Melbourne, 3010 VIC
[2]Department of Computing and Systems, Federal University of Campina Grande, Campina Grande, PB, Brazil
Contact email: yao.pan@unimelb.edu.au.

***Abstract* -** Compared to the traditional approach of using virtual machines as the basis for the development and deployment of applications running in Cloud-based infrastructures, container technology provides developers with a higher degree of portability and availability, allowing developers to build and deploy their applications in a much more efficient and flexible manner. A number of tools have been proposed to orchestrate complex applications comprising multiple containers requiring continuous monitoring and management actions to meet application-oriented and non-functional requirements. Different container orchestration tools provide different features that incur different overheads. As such, it is not always easy for developers to choose the orchestration tool that will best suit their needs. In this paper we compare the benefits and overheads incurred by the most popular open source container orchestration tools currently available, namely: Kubernetes and Docker in Swarm mode. We undertake a number of benchmarking exercises from well-known benchmarking tools to evaluate the performance overheads of container orchestration tools and identify their pros and cons more generally. The results show that the overall performance of Kubernetes is slightly worse than that of Docker in Swarm mode. However, Docker in Swarm mode is not as flexible or powerful as Kubernetes in more complex situations.

***Keywords* - Kubernetes; Docker; Swarm; benchmarking; cloud computing**

I. Introduction

CLOUD computing has changed the face of Information Technology infrastructure provisioning. Large public cloud providers such as Amazon, Microsoft, and Google, as well as federated national platforms such as the National eResearch Collaboration Tools and Resources (NeCTAR – www.nectar.org.au) offer scalable and flexible on-demand access to compute, networking and storage resources to application developers and operators.

An Infrastructure-as-a-Service (IaaS) Cloud provider bundles and subsequently shares compute resources available in physical servers using technologies such as Virtual Machines (VMs), enabling multiple users to simultaneously access their own share of these aggregated resources. The implementation of VMs relies on a hypervisor that multiplexes the underlying hardware, allowing different operating system kernels to securely co-exist on the same physical server, isolated from each other. Containers offer another approach to implement virtualization at the operating system level. They provide a lightweight alternative to VMs, in which a common kernel is shared, whilst also providing isolation guarantees. A container engine is able to run applications whose binaries have been encapsulated into a container image together with the required library dependencies and configuration files. Different kernel process groups and namespaces are used in the single shared kernel to isolate different containers running on the same physical processing node [1]. According to a recent survey of 576 IT leaders [2], the two most widely-adopted container technologies are Docker (52 per cent) and Kubernetes [3] (30 per cent).

Due to their intrinsic portability, containers have been used to support the development, testing, deployment and management of complex Cloud-based applications. They provide an appropriate substrate to decompose large monolithic applications into independent micro-services [4], each packed into different container images. These micro-services can then be individually deployed and configured accordingly, to cooperatively provide the application's functionalities. The micro-service architecture also provides a convenient way to address non-functional requirements such as overall performance, high availability and scalability. In this case, instances of the individual micro-services are typically deployed on a cluster of processing nodes [5].

The management of complex applications deployed on multiple processing nodes calls for additional container orchestration functionalities that are not present in basic container engines. According to another recent survey [5], the investment in container orchestration tools has more than doubled year by year since 2015. In particular, this survey indicates that the two most popular container orchestration technologies are Docker in Swarm mode (from now on referred as Docker Swarm) [1] and Kubernetes. Another study [6] undertaken in 2017 also indicates Docker Swarm and Kubernetes are the two major players in container orchestration and it predicts increasing demand in 2018. Both tools leverage the same underlying container engine, namely Docker [1] [7] [8] [9].

As the name implies, Docker Swarm is part of the native Docker "ecosystem" [1]. As such, it provides a seamless and easy way of working with the Docker Application Programming Interface (API). Consequently, its deployment is simplified, since it more naturally integrates with the other components of the underlying Docker deployment. In turn, Kubernetes uses its own command line interface (CLI) languages such as *kubctl* CLI, which makes the containerization procedure more intricate, and requires a steeper learning curve to developers and operators. It also requires a larger number of extra components that need to be deployed, which not only makes deployment more complex, but also consumes extra resources. On the other hand, Kubernetes provides a richer set of features that have been developed and matured by Google as a result of more than 15 years of experience in running containers in production environments.

These two orchestration tools provide different features, and the different decisions taken in their design yield different overheads that can impact on different types of applications. There has been some effort in trying to assess the pros and cons of these tools (*e.g.* [10]), however there remains a lack of independent assessment that can inform developers and operators when choosing the container orchestration tool that best suits their needs.

In this paper, we report on experiments to measure the overheads incurred by these orchestration tools. We have designed and executed two sets of experiments. In the first, we assess the overheads incurred when an application that uses a single container is executed. This provides a lower bound on the overheads that should be expected when using orchestration tools. This experiment uses different configurations of the Phoronix Test Suite made available by OpenBenchmarking.org. The second set of experiments uses a realistic and complex application that requires multiple containers. This application was developed by the Melbourne eResearch Group (MeG - eresearch.unimelb.edu.au) at the University of Melbourne in the context of the ElectraNet LiDAR Data Processing project [11]. This uses targeted machine learning algorithms to identify particular patterns in point-based LiDAR data. In all cases, the overheads are estimated by comparing the performance of the system using the different orchestration tools against that of a system that executes the same experiments using only the underlying container engine software, *i.e.* without any orchestration support. All experiments were executed over the same infrastructure, provided by the Australia-wide NeCTAR Research Cloud.

The rest of the paper is structured as follows. We start with a discussion of the related work to contextualize the contributions made in this paper (Section 2). We then present the details of the container technologies considered (Section 3). Following this we present the benchmarking methodology used in this work to assess the overheads incurred by the container orchestration tools (Section 4). This is followed by a discussion of the results of the experiments that have been performed (Section 5). We conclude the paper with final remarks and identify directions for future work (Section 6).

II. Background and Related Work

Previous works measuring the performance of container technologies have mostly focused on either comparing the performance of running containerized applications against that achieved by running standard versions of the same applications in VMs, and/or physical servers, and/or assessing their isolation capabilities, i.e. how the performance of an application running in a container or on a VM is impacted by external load in the same node.

Quétier, Néri and Capello [12] were among the first to highlight the performance advantages of using kernel-level virtualization of computing resources. They showed that this approach provides much better start-up time for applications, thereby helping to achieve greater scalability.

Other works focused on assessing the performance penalties incurred by containers and VMs when particular physical resources are stressed. Morabito et al. [13] used a benchmarking tool for stressing CPU use. Their results showed that container-based solutions perform better than their VM-based counterparts. Felter et al. [14] ran the SysBench *oltp* benchmark on a single instance of a MySQL database management system, considering virtualized environments based on containers and VMs. Their results showed that the use of containers resulted in a very small degradation, when compared to a system without virtualization technology. On the other hand, with VMs, the overhead could be as high as 40%.

Xavier et al. [15] compared the performance isolation capabilities of a number of container technologies, and compared them against the performance isolation provided by state-of-the-art virtualization technology. They used benchmarking suites with stress tests for CPU, memory, disk and network. Their results showed that all container technologies, as well as the VM technology, suffer negligible interference with regards to the CPU. On the other hand, other resources suffered more interference when container technology was used. Xavier et al. [16] subsequently performed a deeper evaluation of the performance impact that disk-intensive containerized applications suffered under different stress scenarios. Their results showed that in some scenarios, performance degradation as great as 38% was possible, however, in other scenarios the impact could be negligible.

To the best of our knowledge, there is just one work assessing the overheads caused by the use of tools to orchestrate the execution of multiple containers [10]. This work assessed the startup time of containers for both Docker Swarm and Kubernetes in experiments undertaken with as many as thirty thousand containers on a cluster of one thousand nodes. The results showed that more than half of the Docker Swarm nodes could start a container in less than 0.5 seconds when the cluster was no more than 90% full, whilst over half of the Kubernetes nodes could only start a container in over 2 seconds when the cluster was over 50% full. Moreover, if the cluster of Kubernetes nodes was above 90% full, the startup time of containers could go up to 124 seconds and need manually intervention, while Docker Swarm nodes could retain start-up times under 1 second [10].

We complement previous state of the art by executing experiments that assess the overheads that Docker Swarm and Kubernetes give rise to for different classes of applications. We use a number of test suites from a well-known benchmarking tool to stress the use of different system resources, including CPU, memory, network, disk I/O, as well as a complete functional application. Moreover, previous experiments used an old version of Docker Swarm, which was a standalone project not completely integrated to the Docker engine. Since version 1.12, Docker Swarm has been built into the Docker engine. This is the version that we have used in the experiments that we report in this paper.

III. Container Orchestration Tools

In order to manage multiple containers concurrently, a centralized orchestration tool is typically required. Container orchestration tools can be broadly defined as frameworks that integrate, manage, control and/or schedule containers at scale. This is typically achieved across multiple processing nodes. Different types of container orchestration tools serve different purposes. For instance, *Docker Compose* is a container orchestration tool used for defining and running multi-container Docker applications via YAML scripts, however it does not have the capability for monitoring a cluster or managing containers distributed across networks. Docker Swarm and Kubernetes are two leading technologies that serve this purpose. Both orchestration solutions have their own pros and cons, which results in users facing the dilemma of choosing which solution to adopt. According to the Google Trends, the world-wide interest and searches of Kubernetes over the past three years is much higher than Docker Swarm, and this gap is increasing.

A. Docker Swarm

Docker Swarm was first released in 2016. It has been embedded in the Docker Engine since version 1.12.0. Fig. 1 shows the architecture of a simple Docker Swarm cluster.

A Docker Swarm cluster includes the following components and features:

- *Raft Consensus group* consists of the internal distributed state store in the manager node.
- *Internal Distributed State Store* is a built-in key-value store that is used to maintain the cluster state.
- *Manager Node* in Docker Swarm consists of:
  - an *API* that accepts commands and creates a new service based on the definition;
  - an *Orchestrator* that reconciles the service definition and creates tasks accordingly;
  - an *Allocator* that allocates IP addresses;
  - a *Scheduler* that schedules and assigns tasks to worker nodes, and
  - a *Dispatcher* that conducts check-in actions on worker nodes.

  When the manager node is promoted as a leader node, it is able to conduct orchestration and management tasks such as:
  - receiving the service definition from users;
  - replicating the number of tasks based on the service definition and dispatching to worker nodes;
  - performing orchestration and cluster management;
  - supporting ingress load balancing to expose the services to external, and
  - working with the internal distributed state store to maintain the state of the cluster for scheduling and distributing decisions.
- *Worker Nodes* receive and execute tasks directly from the manager node. Worker nodes also send the status of tasks and a heartbeat to the manager node, to check that the worker node is alive and able to accept tasks.
- A *Service* consists of one or more replica tasks.
- A *Task* refers to the combination of a single Docker container and commands of how it will be run.

In terms of fault tolerance, Docker Swam allows multiple manager nodes to co-exist in a cluster to recover from failures without any downtime. When the existing leader is down or not available, the raft consensus group will elect a new leader to conduct orchestration tasks.

B. Kubernetes

Kubernetes is a production grade container orchestration solution developed and released by Google in 2014. It is a modular system that allows different components and plugins to be included in a Kubernetes cluster.

The Kubernetes Master is the core of the Kubernetes orchestrator. In general, a functioning Kubernetes master consists of the following key components:

- *API Server* is the central component that acts as a management hub. It serves the Kubernetes API and communicates with all other components.
- *etcd* is a lightweight, key-value data store component that stores all configuration data including a registry of what is running and where it is running.
- the *Scheduler* maintains the availability, performance and capacity of the cluster.
- A *Controller Manager* monitors and watches *etcd* via the *API server*. It consists of several controllers, which run in separate processes, for instance:
  - *Node Controller*, which manages, discovers, monitors all nodes in the cluster, and responding to the health condition of nodes.
  - *Replication Controller*, which maintains the replicated objects in the cluster.
  - *Endpoint Controller*, ensures the service for each pod is always up-to-date, and manages endpoints.

- *Service Account and Token Controllers*, manages service accounts and API access tokens for new namespaces.

The Kubernetes nodes, also known as *worker nodes*, receive requests from the Master node. Each Kubernetes node consists of the following components:

- *Kubelet* is the node agent that runs on each user node.
- *kube-proxy* runs on each node to provide simple network proxying and load balancing.
- *Labels* is a key-value pair attached to objects such as services or nodes.
- *User pod* consists of container(s), which have been deployed or scheduled in groups.

Kubernetes supports multiple master nodes running simultaneously, but only one controller manager and scheduler can perform actions within one of the master nodes of the cluster. When the current master dies, a new master is elected.

IV. BENCHMARKING METHODOLOGY

To evaluate and benchmark the performance of the container orchestration tools, especially from the perspective of their associated overheads, we adopt a two-step strategy. In the first step we use a well-known benchmarking test suite to measure the performance of a single container running on a single worker node. This allows us to subsequently assess the overheads that are introduced at the worker node by each container orchestration tool. Then, we use an application that is implemented by multiple containers running over a cluster of Cloud-based worker nodes. This allows us to also assess the overheads that are introduced when containers need to communicate with each other. In both cases, the experiments are also executed in a system that is manually configured using only the underlying container technology, *i.e.* Docker, without any orchestration support. By comparing the performance of this baseline setting against the performance that is achieved when a given orchestration tool is used, it is possible to estimate the performance overheads associated with the tool. The next two subsections give more details about the benchmarks that were executed.

A. Phoronix Test Suite Benchmarks

Phoronix Test Suite (PTS - www.phoronix-test-suite.com) is a comprehensive, open-source, automated benchmark software for Linux and other operating systems. Several performance benchmarks were conducted by applying predefined benchmark profiles via PTS to a two-node Kubernetes cluster and a two-node Docker Swarm cluster. These benchmark scenarios cover CPU, memory, disk I/O, and the overall system performance.[1] Each benchmark suite consists of various benchmark tools to assess the accuracy of the acquired results. Each test profile in the benchmark suite was executed at least 5 times. If the standard deviation of the results collected in these executions exceeded 3.5%, then the test was executed several more times, until the produced results were within the required maximum standard deviation. As a baseline for the benchmark results, the same suites were also applied to a standalone Docker deployment without container orchestration tools. All tests were performed in the same kind of infrastructure provided by the NeCTAR Research Cloud. It is also noted that after the execution of a test, both the cluster and the containers used were removed and rebuilt for the execution of subsequent tests.

We executed different suites of applications to assess the overhead impact on processor performance:

- *The Timed File Compression Test Suite* executes file compression algorithms such as BZip2, Gzip, and LZMA; the same 512 MB file was used in each test.
- *The Computational Biology Test Suite* runs a series of computational biology tests; there are three individual tests in this suite: HMMER, MAFFT, and MrBayes.
- *The Audio Encoding Profile* evaluates the amount of time needed to encode a sample audio file in WAV format into WavePack, Ogg, LAME MP3, Flac, and APE audio formats.
- *The Video Encoding Profile* measures the amount of time needed to encode a sample video file into various formats by utilizing the Mencoder utility and the FFmpeg codec.
- *The SQLite Profile* measures the time to execute a pre-defined number of insertions on an indexed database.

The Memory Test Suite was used to perform the benchmark on memory. This suite includes tests designed to test the computer's system memory performance, speed and bandwidth:

- *RAMSpeed* is an open-source command line utility to measure cache and memory performance of computer systems. RAMSpeed/SMP is for benchmarking multi-processor machines running UNIX-link operating systems.
- The STREAM benchmark is a synthetic benchmark tool that measures sustainable memory throughput by utilizing four long vector operations. In this work, we used *Copy*, *Scale* and *Add* tests to compare memory throughput.[2]
- *CacheBench* was utilized in evaluating the performance of the memory and cache hierarchy. It focuses specifically on parameterizing the performance of possibly multiple levels of cache present on and off the processor.

For Disk I/O performance the *Asynchronous I/O Stress Test Profile* (AIO) and *IOzone Benchmark Test* were used:

- AIO is a form of input and output processing that permits other processing to continue before the transmission has finished. The AIO-Stress benchmark

[1] Unless specified, the default configuration was used in the tests.

[2] There is a general rule for running the *STREAM* benchmark whereby every stream array must be at least four times the size of all last-level caches used, and the minimum array size is 1 million elements. In this test, the array size was set to 100 million elements.

in PTS uses a 2GB test file with 64KB record size. During the execution of the benchmark, the program opens and starts a series of asynchronous disk input and outputs to a test file.

- *IOzone* is a filesystem benchmark tool that generates and measures a variety of file operations. It is useful for performing broad file system analysis. Read and write benchmark tests were executed to read and write 2GB, 4GB and 8GB files with 4KB and 1MB record sizes to evaluate the performance of the disk I/O.

B. *LiDAR Data Benchmarks*

In addition to the PTS benchmarking, a realistic application scenario was included in this work to evaluate the performance of the two container orchestration tools. This focused on processing of Light Detection and Ranging (LiDAR) data required for identification of encroachment of vegetation on power lines [11]. A complete LiDAR data processing platform was deployed to a nine-node Kubernetes cluster and a nine-node Docker cluster in Swarm mode to benchmark the two container orchestration tools. This was used to provide a baseline. The application was also deployed on the same multi-node infrastructure, but using only Docker, and having the required network configuration and other customizations of the environment done manually.

The experiment consisted in the processing of 3GB of LiDAR data in 45 individual Laser File Format (LAS) files. The same LiDAR data was processed 3 times in each setup.

C. *Experimental environment*

In this work, the NeCTAR Research Cloud was used as the underlying Cloud infrastructure for the experiments. NeCTAR is based on OpenStack technology and provides computing infrastructure, software and services that allow Australia's research community to store, access and analyze data, remotely, rapidly, and autonomously.

In OpenStack, the template for virtual hardware is called a "flavor". A flavor defines sizes for resources such as CPUs (number of virtual cores), memory, disk space, and so on. The flavors used in the experiments are given in Table 2.

Table 2: Instance flavors

FLAVORS	m1.medium	m1.large	m1.xlarge
PROCESSOR	Intel Core (Broadwell) @ 2.10GHz		
Core Count	2	4	8
Extensions	SSE 4.2 + AVX2 + AVX + RDRAND + FSGSBASE		
Cache Size	4096 KB		
Microcode	0x1		
MOTHERBOARD	OpenStack Foundation Nova v13.1.2		
Memory	8 GB	16 GB	32 GB
Chipset	Intel 440FX- 82441FX PMC		
Network	Red Hat Virtio device		
DISK	10 GB		
File-System	Ext4		
Mount Options	data=ordered errors=remount-ro realtime rw		

For conformity, all instances were running "NeCTAR Ubuntu 16.04 LTS (Xenial) amd64" image as the host OS. The details of this image are shown in Table 3.

Table 3: Instance image

Name	NeCTAR Ubuntu 16.04 LTS (Xenial) amd64
Size	287.0 MB
Container Format	BARE
Disk Format	QCOW2
Architecture	X86_64
OS Distro	Ubuntu
OS Version	16.04
Kernel	4.4.0-92-generic (x86_64)
System Layer	KVM

The OpenStack Network service (Neutron) provides network connectivity and addressing in the NeCTAR Cloud. There are three types of networks in Neutron: external networks, provider networks and tenant networks. A provider network is created by the Cloud administrator. It is mapped to and runs on top of the external network. Each instance is allocated an IP address from the provider network (qh2-uom) by default. The tenant network is provisioned by users and isolated from other tenants. Two tenant networks (Kubernetes and Swarm) were created for each environment in the experiments to link all the instances via private IP addresses in the same environment. These were isolated from other environments.

The container engine, container orchestration tools, and benchmarking tools used in the experiment and their versions are listed in Table 4.

Table 4: The software and their versions

Name	**Type**	**Version**
Docker	Container Engine	17.06.0-ce
Docker-compose	Container Engine	1.16.1
Kubernetes	Container Orchestration Tool	1.8.0
Phoronix Test Suite	Benchmark Suite	7.4.0

In this work the experimental platform consisted of one m1.medium instance (manager nodes), one m1.large instance and seven m1.xlarge instances (worker nodes). The details for the instances in the Phoronix Test Suite benchmark and in the LiDAR experimental environment are presented in Table 5.

Table 5: Instances used in the PTS experimental environment

Name	**Flavor**	**Availability Zone**
PTS Manager node	m1.medium	melbourne-qh2-uom
PTS Worker node	m1.large	melbourne-qh2-uom
LiDAR Manager node	m1.medium	melbourne-qh2-uom
LiDAR Worker node 1-8	m1.large	melbourne-qh2-uom

The size of the root disk in NeCTAR m1 flavor was set to 10 GB. A 25 GB persistent volume storage was attached to each of the instances for Docker storage. Two 512 GB persistent volume storages were created in each experimental environment for persisting the database and SFTP server in LiDAR experiment.

D. *Comparison methodology*

We used statistical inference to compare systems that used orchestration tools (Docker Swarm and Kubernetes) with a baseline system that used neither of these tools (Docker). We ran a number of experiments and measured the performance of the system – processing time, and throughput, depending of the benchmark used. Since the experiments were executed over a controlled environment, we assume that the sampled data follows a normal distribution. For each benchmark we took the mean of the measurements in the replicated runs as an estimate of the performance of the system.

For each benchmark, we then used hypothesis testing to infer whether a system that used a particular orchestration tool performed worse than the baseline system. We considered different null and research hypotheses, depending on whether the benchmark measured the processing time or throughput. For example, let $\bar{x}_b$ and $\bar{x}_o$ be the mean time for the sample executions of the baseline system for a particular benchmark, and the mean time for the sample executions of one of the systems that uses an orchestration tool respectively. For the benchmarks that measure the processing time, the null (H_0) and alternative hypothesis (H_a) are formulated as follows:

$$H_0\!:\bar{x}_o - \bar{x}_b = 0;$$
$$H_a\!:\bar{x}_o - \bar{x}_b > 0.$$

For those that measure throughput, the hypothesis is formulated as follows:

$$H_0\!:\bar{x}_b - \bar{x}_o = 0;$$
$$H_a\!:\bar{x}_b - \bar{x}_o > 0.$$

Since the variances of the two independent samples are not too different, we have used pooled variances. For the case where we have evidence that the null hypothesis can be rejected, *i.e.* we have statistically meaningful evidence that the system using an orchestration tool introduces a significant overhead, we calculate an estimate of the maximum value of the overhead for the benchmarks that measure processing time (resp. throughput) by computing the difference between $\bar{x}_b$ and $\bar{x}_o$ (resp. $\bar{x}_o$ and $\bar{x}_b$) and divide by $\bar{x}_b$.

In summary, we report in our results the mean performance of the baseline system, together with the confidence intervals. When there is statistical evidence that a system that uses an orchestration tool to execute a particular benchmark performs worse than the baseline system, we the estimate of the maximum value for this overhead. In all cases, we report results with at least 95% confidence level.

V. Benchmarking Results and Discussion

A. Results

As explained in the previous section, for the case when the application is encapsulated in a single container we have executed two types of benchmarks. The first type of benchmark was useful to understand the impact of the orchestration tools in the processing times of the containerized application. The results for these benchmarks are presented in Table 6.

Table 6: Average time to execute the benchmark for single-container application

Name	Baseline: mean execution time for Docker (second)	Docker Swarm		Kubernetes	
		Decision on the null hypothesis	**Estimated maximum overhead (%)**	**Decision on the null hypothesis**	**Estimated maximum overhead (%)**
Parallel BZIP2	18.76± 0.79	Failed to reject	N/A	Failed to reject	N/A
Gzip	18.4± 1.42	Failed to reject	N/A	Failed to reject	N/A
LZMA	387.32± 7	Failed to reject	N/A	Failed to reject	N/A
HMMER Search	27.27± 0.51	Failed to reject	N/A	Rejected	**4.2%**
MAFFT Alignment	8.93± 0.29	Failed to reject	N/A	Failed to reject	N/A
MrBayes Analysis	821.69± 6.35	Rejected	**3.5%**	Rejected	**3.5%**
Audio Encoding (WavPack)	11.65± 0.19	Failed to reject	N/A	Rejected	**6.4%**
Audio Encoding (Ogg)	10.82± 0.29	Failed to reject	N/A	Failed to reject	N/A
Audio Encoding (LAME MP3)	18.01± 0.18	Failed to reject	N/A	Rejected	4.6%
Audio Encoding (FLAC)	11.24± 0.25	Failed to reject	N/A	Failed to reject	N/A
Audio Encoding (APE)	14.01± 0.24	Failed to reject	N/A	Failed to reject	N/A
Video Encoding (Mencoder)	30.78± 0.48	Failed to reject	N/A	Failed to reject	N/A
Video Encoder (FFmpeg)	21.73± 0.26	Failed to reject	N/A	Failed to reject	N/A
SQLite	97.53± 3.27	Failed to reject	N/A	Failed to reject	N/A

The other PTS benchmarks executed were used to understand the impact on the throughput achieved for memory access and disk I/O by the containerized applications. The results for these benchmarks are presented in Table 7.

Table 7: Average throughput achieved by the benchmark for single-container application

Name	Baseline: mean execution time for Docker (second)	Docker		Kubernetes	
		Decision on the null hypothesis	**Estimated maximum overhead (%)**	**Decision on the null hypothesis**	**Estimated maximum overhead (%)**
Integer (Add)	19621.22± 485.44	Failed to reject	N/A	Failed to reject	N/A
Integer (Copy)	17703.15± 169.64	Failed to reject	N/A	Rejected	**4.5%**
Integer (Scale)	17383.17± 318.67	Failed to reject	N/A	Failed to reject	N/A
Floating-Point (Add)	20075.7± 513.63	Failed to reject	N/A	Rejected	**6.8%**
STREAM (Add)	37859.8± 131.52	Failed to reject	N/A	Rejected	**1.5%**
STREAM (Copy)	53819.58± 288.1	Rejected	**1.8%**	Rejected	**2.0%**
STREAM (Scale)	34151.98± 136.66	Failed to reject	N/A	Rejected	**2.1%**
CacheBench	2175.94±	Failed to	N/A	Rejected	**6.6%**

Name	Baseline: mean execution time for Docker (second)	Docker		Kubernetes	
		Decision on the null hypothesis	Estimated maximum overhead (%)	Decision on the null hypothesis	Estimated maximum overhead (%)
(Read)	46.83	reject			
CacheBench (Write)	14814.4± 309.37	Failed to reject	N/A	Rejected	**7.9%**
AIO-Stress (Random Write)	1111.99± 111.04	Failed to reject	N/A	Failed to reject	N/A
IOZone (Read 8GB/4K)	2399.97± 231.56	Failed to reject	N/A	Failed to reject	N/A
IOZone (Read 8GB/1MB)	4060.92± 561.08	Failed to reject	N/A	Failed to reject	N/A
IOZone (Read 4GB/4K)	2452.28± 175.29	Failed to reject	N/A	Failed to reject	N/A
IOZone (Read 4GB/1MB)	3852.33± 498.5	Failed to reject	N/A	Failed to reject	N/A
IOZone (Read 2GB/4K)	2439.46± 246.22	Failed to reject	N/A	Failed to reject	N/A
IOZone (Read 2GB/1MB)	3964.38± 265.2	Failed to reject	N/A	Failed to reject	N/A
IOZone (Write 8GB/4K)	587.8± 24.27	Failed to reject	N/A	Failed to reject	N/A
IOZone (Write 8GB/1MB)	673.31± 31.21	Failed to reject	N/A	Rejected	**12.3%**
IOZone (Write 4GB/4K)	505.45± 26.81	Failed to reject	N/A	Failed to reject	N/A
IOZone (Write 4GB/1MB)	529.07± 43.76	Failed to reject	N/A	Failed to reject	N/A
IOZone (Write 2GB/4K)	418.22± 17.67	Failed to reject	N/A	Failed to reject	N/A
IOZone (Write 2GB/1M)	453.03± 28.3	Failed to reject	N/A	Failed to reject	N/A

Finally, we present in Table 8 and Fig. 1 the results for the LiDAR Data processing application, which comprised multiple containers.

Table 8: Average time to execute the multi-container application

Name	Baseline: mean execution time for Docker (second)	Docker		Kubernetes	
		Decision on the null hypothesis	Estimated maximum overhead (%)	Decision on the null hypothesis	Estimated maximum overhead (%)
LiDAR	15,913.67± 248.38	Failed to reject	N/A	Rejected	**8.3%**

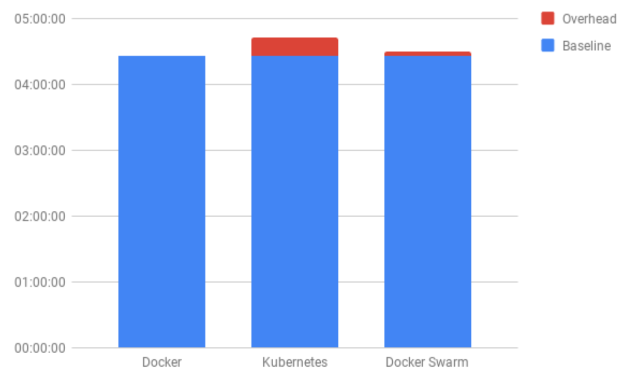

Fig. 1 Orchestration overhead observed in LiDAR Experiment

B. *Discussion*

The results of the experiments for the applications running in a single container show that, when compared to the performance achieved by the baseline Docker system, there are no meaningful overheads associated with Docker Swarm in all, but two benchmarks. Even when the overheads are meaningful, they are quite low: maximum of 3.5% for MrBayes Analysis and 1.8% for STREAM (copy). On the other hand, there are meaningful overheads associated with the execution of Kubernetes in as many as 1/3 of the experiments. The values of the maximum overheads, however, are also reasonably low: the highest is 12.3% for the IOZONE (Write 8GB/1MB), but the mean value is only 5.2%, and the median is 4.5%.

Considering the results of the experiments with the multi-container application, again, we cannot say that the performance of Docker Swarm is worse than that of Docker, while in the worst case, it is estimated that the performance of Kubernetes is as much as 8.3% worse than that of Docker.

In comparison to a native Docker deployment, Docker Swarm requires an additional "Swarm-wide" overlay network for ingress load balancing and service discovery. The experiments show that further overheads are low. Linux IPVS is often used in Docker for load balancing. This has been in the Linux kernel for many years and proven to be one of the most efficient load balancers. Compared to Docker Swarm, Kubernetes requires a number of additional components to form a cluster. While these components are not necessarily expensive to run, such overheads may be noticeable when multi-container applications are deployed.

VI. Conclusions and Future Work

Interest and adoption of container technology has been growing rapidly. Just as a capable virtual machine monitoring and management tool is crucial for hypervisor-based environments, monitoring and management of containers is equally essential. In this work, a thorough comparison and performance benchmark of the two most popular container clustering and orchestration tools were described. Docker Swarm as a native Docker component is relatively easy to deploy and configure with other existing

Docker components whilst Kubernetes, a third-party Docker container orchestration tool, is relatively more comprehensive and mature in terms of functionalities and experiences in mainstream production environments. While the additional Kubernetes components are not necessarily expensive to run, the additional infrastructure required to run them can be expensive when it comes to small sized clusters. For larger deployments, Kubernetes' auto-scaling features make maximum use of resources, and the overheads of running Kubernetes are reduced.

Docker Swarm comes with the latest Docker engine and works with other existing Docker tools. This makes it a lightweight and out-of-box orchestration solution. However, its functionality is limited by what is available in the Docker API, and hence it only provides limited fault tolerance capabilities. Kubernetes comes with several years of expert experience in production deployment environments and hence provides comprehensive features and support. However, it is not compatible with the existing Docker CLI and composition tools and requires additional skill sets to install, configure and manage.

Regarding the outcome of the performed experiments and evaluations, Docker Swarm shows comparable performance with Docker. In only two of the experiments performed were the overheads of Docker Swarm noticeable, however these were very low. On the other hand, Kubernetes showed slightly higher overheads in some of the performed experiments, however the estimated maximum overheads were reasonably low.

Both Docker Swarm and Kubernetes can manage Docker container clusters efficiently and run many of the same services, but they can require different technical approaches. The generated results in this work provide some direction and should be considered when designing container-based Cloud environments and choosing the right container clustering and orchestration tool.

On 5th June 2018 Amazon announced that its managed Kubernetes service, Amazon Elastic Container Service for Kubernetes, would be generally available. This means that all major cloud service providers can now have integrated Kubernetes as part of their own cloud container services. The performance of a Kubernetes-based cloud service from different cloud service providers would be something that can be investigated in future work.

ACKNOWLEDGMENT

The authors would like to thank the National eResearch Collaboration Tools and Resources for providing the (free) access to and use of the Research Cloud resources that were used in this paper. Acknowledgments are also made to the Australian Government Research Training Program Scholarship for supporting the work, and the LiDAR project team at the Melbourne eResearch Group for the support in utilizing the LiDAR application in this paper.

REFERENCES

[1] J. Turnbull, The Docker Book: Containerization Is the New Virtualization, Turbull Press, 2014.

[2] Diamanti, "2018 Container Adoption Benchmark Survey," 28 July 2018. [Online]. Available: https://diamanti.com/wp-content/uploads/2018/07/WP_Diamanti_End-User_Survey_072818.pdf.

[3] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing,* vol. 1, no. 3, pp. 81 - 84, Sept 2014.

[4] S. Newman, Building Microservices: Designing Fine-Grained Systems, 1st Edition ed., O'Reilly Media, 2015, pp. I-XVIII, 1-259.

[5] K. Zhanibek, R.O. Sinnott, *A Performance Comparison of Microservice Hosting Technologies for the Cloud*, Future Generation Computing Systems, August 2016, Volume 68, March 2017, Pages 175-182.

[6] Portworx, "2017 Annual Container Adoption Survey: Huge Growth in Containers," Portworx, 12 April 2017. [Online]. Available: https://portworx.com/2017-container-adoption-survey/.

[7] G2 Crowd, "Digital Platforms Trends 2018," G2 Crowd, 2017. [Online]. Available: https://blog.g2crowd.com/blog/trends/digital-platforms/2018-dp/.

[8] B. Ruan, H. Huang, S. Wu and H. Jin, "A Performance Study of Containers in Cloud Environment," in *Asia-Pacific Services Computing Conference*, 2016.

[9] G. Lahmann, T. McCann and W. Lloyd, "Container Memory Allocation Discrepancies: An Investigation on Memory Utilization Gaps for Container-Based Application Deployments," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Orlando, 2018.

[10] H. A. Farias, D. Ortiz, C. Núñez, M. Solar and M. Bugueno, "ChiVOLabs: cloud service that offer interactive environment for reprocessing astronomical data," in *SPIE Astronomical Telescopes + Instrumentation 2018*, Austin, 2018.

[11] J. Nickoloff, "Evaluating Container Platforms at Scale," 9 March 2016. [Online]. Available: https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c.

[12] Melbourne eResearch Group, "ElectraNet LiDAR Data Processing," The University of Melbourne, [Online]. Available: https://eresearch.unimelb.edu.au/projects/.

[13] B. Quétier, V. Neri and F. Cappello, "Scalability Comparison of Four Host Virtualization Tools," *Journal of Grid Computing,* vol. 5, no. 1, pp. 83-98, 19 Spet 2006.

[14] R. Morabito, J. Kjällman and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*, Tempe, 2015.

[15] 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)*, Philadelphia, 2015.

[16] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange and C. A. F., "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Belfast, 2013.

[17] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos and K. J. Matte, "A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds," in 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turku, 2015.