# Evaluation of container orchestration systems for deploying and managing NoSQL database clusters

Eddy Truyen, Dimitri Van Landuyt, Bert Lagaisse, Wouter Joosen
imec-DistriNet, KULeuven
{firstname}.{lastname}@cs.kuleuven.be

Matt Bruzek
Red Hat
mbruzek@redhat.com

***Abstract*—Container orchestration systems, such as Docker Swarm, Kubernetes and Mesos, provide automated support for deployment and management of distributed applications as sets of containers. While these systems were initially designed for running load-balanced stateless services, they have also been used for running database clusters because of improved resilience attributes such as fast auto-recovery of failed database nodes, and location transparency at the level of TCP/IP connections between database instances. In this paper we evaluate the performance overhead of Docker Swarm and Kubernetes for deploying and managing NoSQL database clusters, with MongoDB as database case study. As the baseline for comparison, we use an OpenStack IaaS cloud that also allows attaining these improved resilience attributes although in a less automated manner.**

***Keywords*-Container orchestration; Performance evaluation; NoSQL databases**

## I. Introduction

Recently, there has been an increasing industry adoption of Docker containers for simplifying the deployment of software[14], [20], [30]. Almost in parallel, container orchestration middleware, such as Docker Swarm[9], Kubernetes[5], Mesos[19], and OpenShift 3.0[18] have arisen which provide support for automated container deployment, scaling and management.

NoSQL databases are an important technology trend for supporting the storage of various kinds of data structures with various trade-offs between consistency, availability and network-partition tolerance[4]. The configuration and maintenance of the data tier of a distributed software service is a challenging task however. There are a lot of configuration parameters with inter-dependencies. Ensuring consistency of these configuration parameters between various development, testing and production environments is of key importance in contemporary DevOps environments. Docker's approach to packaging and distributing container images has the potential to offer a simple consistency management solution as a specific Docker image is a portable and light-weight software package that encapsulates a specific performance-tuned configuration of a NoSQL database.

Companies increasingly consider running Dockerized NoSQL databases, not only in development environments, but also in cloud-based production environments[31], [8]. In large-scale deployments, container orchestration systems are then a logical necessity for managing important non-functional requirements, such as resilience, resource cost-efficiency, and elasticity[5].

It has been shown that containerizing NoSQL databases yields a number of benefits[3], [31], [28]. More specifically, (a) fast auto-recovery of failed database instances can be achieved and (b) support for location transparency of TCP/IP connections between database instances even when they are migrated to another node. These two advantages ensue because of several common features of container orchestration systems: (i) a highly customizable central scheduler that places containers on nodes according to user-specified placement constraints, (ii) automated node failure dectection and automated container eviction, (iii) integration with services for distributed persistent volumes for storing the disk data blocks of database instances in a more dependable fashion such that disk data blocks can be recovered after a node failure, (iii) virtual networks for containers where each container has a separate IP address, and (iv) a built-in service proxy such that each client-visible application service can be reached via the same IP address, even after a container is evicted to another node.

These features are common among many container orchestration systems including Kubernetes, Docker Swarm and Mesos[1], although little is known about the performance overhead of these features.

In this paper, we quantify the performance overhead of using a container orchestration system for deploying and managing NoSQL database clusters and compare these costs to deployments of these databases directly on top of an IaaS cloud platform. We evaluate the performance overhead of 2 container orchestration systems – Docker Swarm and Kubernetes – with MongoDB as database case study. As baseline, we use an OpenStack IaaS cloud in a closed lab. In this IaaS platforms, the above described features are also supported although in a less automated way.

This paper is structured as follows. Section 2 summarizes the existing work on performance evaluation of container technologies. Section 3 gives an up-to-date overview of the most popular container orchestration systems and their support for running database clusters. Section 4 presents the experimental design and quantitative findings of the performance evaluation. Section 5 discusses the implications of these findings.

[1]Kubernetes v1.2+, the Swarm integrated mode of Docker v1.12+ and Mesos v1.0.0+ comprise support for all four features.

## II. Related work

Existing research[15], [17], [30] has focused most attention on comparing the performance of a single container against a single virtual machine, both running directly in Linux on top of a bare-metal machine. Sharma et al.[30] evaluate the so-called hybrid model where a containerized Redis instance, inside a VM, is compared to Redis directly installed in another VMs and show that the hybrid model performs slightly better.

Literature on design and evaluation of container orchestration systems (CO systems) originates mostly from Apache[19] and Google[32]. Verma et al. [32] shows that the Borg system, a predecessor of Kubernetes, supports improved resource utilization in terms of number of machines needed for fitting a certain workload on.

Existing studies of containers consistently report and confirm that containers trade improved cost-efficiency for reduced security isolation in comparison to virtual machines[14], [34]. For this reason, cloud providers offer the hybrid VM+Container model in order to protect their infrastructural assets. However, little research exists on the performance evaluation of CO systems that are deployed on top of such virtualization-based cloud platform. Kratzke et al. [21] has already studied the overhead of the virtual networks between containers in public cloud providers and concludes that operating container clusters with highly similar core machine types is the best strategy to minimize the data-transfer rate-reducing effects of containers in several different public cloud providers. As such, we take the findings of this work as premise for our experiments.

## III. Overview of container orchestration systems

Docker Swarm (*Swarm*)[9], Kubernetes (*K8s*)[24], Mesos[1], and OpenShift 3.x[18] are the most popular container orchestration systems for running production-grade services in private OpenStack Clouds, according to a recent survey among OpenStack cloud administrators[29]. As OpenShift is based on K8s, we focus our attention on the other three CO systems.

K8s and Swarm support deploying and managing service- or job-oriented workloads. Mesos is a low-level orchestration sytems that supports scheduling differerent CO frameworks on top of a shared cluster of machines[19]. The core architectural pattern underlying K8s, Swarm and Mesos-based systems is very similar: a container cluster consists of Master and Worker nodes and the Master offers an API for declarative configuration of desired distributed applications.

First we describe their common features that are relevant to database clusters. Secondly, we describe how these features can be used for deploying database clusters.

### *A. Common CO features relevant to database clusters*

**1. Highly customizable scheduler :** All CO systems allow to restrict the placement decision of their default scheduling algorithm by means of various user-specified placement constraints. These user-specified constraints support placing inter-dependent application containers and data close or far from each other in the network topology in order to meet application-specific requirements.

**2. Node failure detection and container eviction:** All CO systems support tolerance against node fail-stop errors. When a node has been identified as down by the master and has not returned to a normally functioning mode within a configurable timeout, all containers on that failed node are evicted and will be restarted on another node that matches the placement constraints of these containers. It is also possible to wait for the failed node to return by specifying a placement constraint that only matches with that node only.

**3. Persistent volumes**: In all CO systems, containers are by default stateless; when a container dies or is stopped, any internal state is lost. Therefore, so called persistent volumes can be attached to store the persistent state of the container. Persistent volumes can be implemented by services for attaching block storage devices to guest virtual machines such as OpenStack's Cinder service, distributed file systems such as NFS, cloud services such as GCE Persistent Disk or just be directory on the host machine.

**4. Virtual networks for containers**: In order to improve the original port-to-port linkage model of Docker containers, K8s has introduced from its inception an improved container networking model[16]: each container can be reached via a separate virtual cluster IP address, which simplifies service discovery and connection management. Virtual network software is used to provide a layer 3 network between the containers and to control how network traffic between containers is transported between nodes of the cluster. Note that containers have transient IP addresses: when a container is stopped or migrated to another node, its cluster IP address is released.

In response, Docker v1.9 improved its networking model with IP addresses per container. Mesos also introduced support for IP addresses per container since Mesos v0.25[2]. However, the specific implementation details have been changed multiple times. As such, Mesos support for virtual networks has not been used by any Mesos-based CO system, except DC/OS [27] very recently.

In K8s and Docker Swarm, multiple alternative networking solutions exist that are available as modular plugins. In K8s, the flannel network software is an often used and well-working solution [23] while Docker Swarm uses Docker's default *overlay* network solution [10]. Both CO systems also support the Weave NET networking solution[33] that has been used in the performance evaluation (see Section 4) to keep the evaluated database deployments as identical as possible.

**5. Services**: CO systems allow exposing the service of a running container to a client via a Service object, which embodies a stable cluster IP address, a network protocol and one or more ports. The cluster IP address of a service is named stable because clients of the service can depend on location transparency at the level of TCP/IP connections: when a container is restarted, the new container gets another cluster IP address, but this change is invisible for clients as they remain connected to the unchanged cluster IP address of the service. Requests to the Service IP address are processed by a local or distributed service proxy which forwards requests to the transient IP addresses of the containers of the service, using a load-balancing algorithm.

A service can also have an external IP address in order to enable access by external clients outside of the container cluster. This external IP addresss is either supported by an external load balancer or by so-called NodePorts, which is a cluster-wide unique port number on which the service can be reached.

All CO systems also support an internal DNS service that enables lookup of the cluster IP address of a service based on its name that is specified in its Service configuration.

**6. Host ports:** Finally, it is also possible to attach HostPorts to containers so that clients can directly send network request to a container without requiring to pass through service proxies and the virtual network. This allows clients to use their own load balancer solution, and possibly avoid the known performance overheads of the virtual network layer[22], [21]. Note that in the case of Kubernetes and Mesos, which adopt the Container Networking Interface specification, a separate **portmap plugin**[6] must be installed on each node of the cluster in order to enable communication to and from host ports.

## *B. Support for database clusters*

CO systems were initially designed for deploying and managing stateless applications and therefore proved less suited for databases. The essence of the difficulty is that the built-in service proxy and replication scheme of CO systems are in conflict with the existing load balancing and replication algorithms of database cluster software systems such as MongoDB and Cassandra. In all CO systems a Service typically is offered by a pool of identically configured container replicas that are scheduled across multiple nodes. The service proxy implements round-robin load balancing across these container replicas or can be configured to forward requests from the same the client IP to the same replica (i.e, sticky sessions). However, putting a round-robin or client-IP based loadbalancer in front of any type of database cluster is not going to work because the database cluster already implements its own load balancing scheme that is based on how data is replicated across multiple database instances. In order to overcome this problem, four constraints have to be taken into account for defining a correct container-based configuration of a database cluster:

1) Separate Service objects need to be defined for each database container in the cluster such that each database container has a separate Service IP address; exactly one container should thus run per Service IP so that all requests are forwarded to that container. An alternative for such unique Service IP addresses per databases is registering the transient cluster IP address of each database container as a DNS record in the internal DNS service of the CO system. In K8s, for this purpose, a Headless Service object can be defined, which is a Service object without a stable cluster IP address.
2) Each database instance needs to be associated with a uniquely identified persistent volume.
3) Database instances are created and destroyed in an ordered fashion. For example in master-slave architectures such as MongoDB, primary database instances are always created first, and thereafter secondary instances
4) Database instances need to be placed on different nodes and close to the data storage location of the persistent volumes. Therefore, the cluster scheduler, which controls the placement of containers on nodes, needs to be configured accordingly by means of user-specified placement constraints.

K8s version 1.5+ offers support for the concept of StatefulSet[25] which is a higher-level configuration abstraction for automating the deployment of database clusters such that the above four constraints can be automatically applied. The most innovative addition of StatefulSets is that when a new database container is added to the StatefulSet, a new persistent volume is automatically provisioned. However, it is also possible to adhere to the above four constraints in Swarm and Mesos Marathon by means of a shell script that first creates a persistent volume and than creates a new database container. With StatefulSets, the default service networking option is to use Headless Service objects for each database instance (i.e. each instance is exposed as a stable DNS name).

## IV. PERFORMANCE EVALUATION

This section is structured as follows. Section IV-A motivates and defines the goal of the performance evaluation and presents the different variants of MongoDB database deployments that will be evaluated. Thereafter Section IV-B presents the research questions. Then Section IV-C specifies the overall experimental design and the OpenStack-based testbed. Finally, Section IV-D presents the experimental findings for MongoDB. The MongoDB experiments presents a systematic study for all YCSB workloads A-F. The code and data of these experiments are made available on-line [13].

### *A. Goal and compared database deployments*

Orchestration actions are exceptional for database clusters. After all, databases are relatively long-running systems and only in case of failures or workload changes specific actions such as container eviction or scaling/up or down is necessary. As such, we consider the *constant overhead of the CO features during normal operation as the biggest additional cost over time.*

The goal of this paper is thus to measure the performance overhead of CO systems during normal operations while the container orchestration system does not need to take any orchestration actions. The aboxe six CO features, as presented in Section III-A, cannot be evaluated in isolation however. After all, the five first features are built-in by default in all CO systems and are actually already used in K8s and Swarm for running the container orchestration software itself such as the internal DNS service.

We expect that the overhead of the first three features of CO systems (i.e., customizable scheduling, container eviction on node failure detection and persistent volumes) to be low. First, the operational cost of the cluster scheduler, which only runs on the master node of the container cluster, can be shared with all other applications running in the cluster. Moreover, for a particular application – in this case a database cluster – the scheduler will only perform orchestration actions at concrete points of time such as deployment of a container, node failure or a scaling action. Secondly, node failure detection is implemented as a simple and thus lightweight healt checking protocol between master and worker nodes; moreover container eviction is also an exceptional orchestration action that will only be triggered in case a node has been detected as failed and a configurable timeout period has passed. Third, the overhead of using persistent volumes in Docker is also expected to be low because the drivers for these persistent volumes bypass the AUFS or OverlayFS storage drivers used by Docker and therefore do not incur the known potential overheads for write-heavy workloads incurred by these storage drivers [30], [11].

The overhead of the 4th and 5th features (i.e., virtual network software and service proxies for enabling TCP/IP location transparency) may however introduce a non-significant performance overhead, as already indicated by prior research[22], [21]. We expect however that some of this overhead may be avoided by exposing database containers via the combination of the sixth feature (i.e., host ports) and a stable floating IP address or stable VM hostname which can be provisioned by the underlying cloud provider.

Different configuration options for setting up a stable TCP/IP endpoint of a single database instance thus ensue. Names for denoting the different TCP/IP endpoint types are specified by the following BNF grammar:

$$\begin{aligned}
< EndpointType > &::=< CloudProvisioned > \\
&\quad | < ClusterProvisioned > \\
< CloudProvisioned > &::=< StableVmIP > HostPort \\
< StableVmIP > &::= FloatingIP \mid Hostname \\
< ClusterProvisioned > &::= ServiceIP[ServicePort] \\
&\quad | \, ContainerDnsName[ContainerPort] \\
&\quad | < StableVmIP > NodePort
\end{aligned}$$

We aim to compare four kinds of variants for deploying the same database cluster: (i) non-containerized deployments where each database is directly installed in the Linux OS of the VM and exposed via a *CloudProvisioned* TCP/IP endpoint (i.e. a *FloatingIP* or *Hostname* in combination with a *HostPort*), (ii) a Docker-only deployment without NAT where each database container runs in HOST mode[2] and is also exposed via a *CloudProvisioned* endpoint, (iii) Container-Orchestrated (CO) deployments where each database container is also exposed via a *CloudProvisioned* endpoint, and (iv) CO deployments where each database container is exposed via a *ClusterProvisioned* endpoint (i.e., either *ServiceIP*, *ContainerDNSName* or *NodePort*). Names for all possible configurations of database deployments that will be compared can thus be represented by the following BNF grammar:

$$\begin{aligned}
< DB > &::=< NodePlatfom >< CloudProvisioned > \\
&\quad | < CO >< ClusterProvisioned > \\
< NodePlatform > &::= VM\_ \mid DockerOnly \mid < CO > \\
< CO > &::= Swarm \mid K8
\end{aligned}$$

For *CO-based* database deployments, we will only evaluate the performance overhead of K8s and Docker Swarm. We have excluded Mesos from the experiments because at the time of experiments the container networking solution of Mesos has not yet been used in any Mesos-based CO framework (see Section III-A for an up-to-date overview of the current state of affairs in Mesos). We have installed Kubernetes v1.7.2 using the kubeadm v1.7.2 tool[26]. We installed Docker Swarm integrated mode as part of Docker engine 17.04.0˜ce-0˜ubuntu-xenial.

### B. Research questions

Given the above range of possible deployments, we refine the goal of the paper by investigating three research questions (RQ):

**RQ1.** What is the overhead of *Container Orchestrated (CO)* database deployments with only *CloudProvisioned* endpoints in comparison to *VM*-based or *DockerOnly* deployments with also *CloudProvisioned* endpoints? We expect that response latency will increase because more CPU time will be spent by CO-based deployments in processing soft interrupts that are required for activating the virtual bridge to redirect VM-level network traffic to the virtual ethernet interface (veth) of the database container[12]. Therefore we will only test CPU-intensive or in-memory database workloads for answering this questions.

**RQ2.** What is the overhead of *CO*-based deployments with *ClusterProvisioned* endpoints in comparison to *CO*-based deployments with *CloudProvisioned* endpoints? We expect that a higher response latency because the aforementioned virtual bridge now needs to redirect VM-level network I/O for the database container **twice**: one time from the VM-level network interface of the local node to a veth of the virtual bridge, a second time from another veth of the virtual bridge to the veth of the database container[12].

**RQ3.** Can the performance overhead of *CO*-based database deployments with *ClusterProvisioned* endpoints be reduced by choosing a different network plugin?

### C. Experimental design and testbed

We have used the well-known YCSB benchmark for evaluating database clusters (see table I) [7] for running the performance evaluation experiments. We have ran the core workload types A, B, C, D and F. Each of these workload

[2]i.e., the networking stack of the node's operating system is directly used for connecting to containers, so the overhead of Network Address Translation (NAT) is avoided.

| Workload type | Operations | Record selection |
|---|---|---|
| A - Update heavy | Read: 50%, Update 50% | Zipfian |
| B - Read heavy | Read: 95%, Update: 5% | Zipfian |
| C - Read only | Read: 100% | Zipfian |
| D - Read latest | Read: 95%, Insert: 5% | Latest |
| F - Read-modify-write | Read 50%, Rmw: 50% | Zipfian |

Table I
WORKLOADS FROM THE CORE PACKAGE OF YCSB BENCHMARK [7]

types combines insert, read and update operations according to a specific ratio (e.g 90% reads, and 10% updates). Read operations select data records according to a either a Zipfian or a Latest distribution[3].

Our testbed is an isolated part of a private OpenStack cloud, version Liberty. The OpenStack cloud consists of a master-slave architecture with two controller machines and droplets, on which VMs can be scheduled. The droplets have Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz processors and 64GB DIMM DDR3 memory with Ubuntu xenial, while the master controller is an Intel(R) Xeon(R) CPU E5-2430 @ 2.20GHz machine with Ubuntu xenial. CPU cores are exclusively reserved for a single VM. Each droplet has two 10Gbit network interfaces. Means and standard deviations of multiple ping tests between the different VMs on different droplets are on average 0.694ms and 0.306 ms respectively.

Each database deployment is setup as a database cluster with 3 database instances. All database instances store their state on a local directory of the VM, which in turn is stored on an attached Openstack Cinder volume. Swarm and Kubernetes deployments respectively use a local Docker volume and a hostPath volume; moreover placement constraints are specified so that after a node fails, the database instance will be rescheduled to that same node whenever it returns into a healthy state. This is the most simple auto-recovery strategy. Evaluating auto-recovery strategies that involve using distributed persistent volumes is outside the scope of this paper. Finally, all database deployments are configured so that write operations need to be acknowledged by at least two database instances, while read operations need only be processed by the invoked database instance. We thus create network traffic between database instances for every write operation. As such, any overhead of the virtual network layer will be more pronunciated for these write operations.

The three database instances of a cluster run in three separate VMs with 2 vCPUs and 4 GB of RAM. All VMs run Ubuntu xenial 4.4.0-112-generic. Swapping is turned off. Each of these VMs runs on a separate droplet which is kept the same for all experiments. The YCSB VM is also deployed on a separate, yet always the same droplet with a c4m8-flavored VM.

In general, each experiment is setup as a scalability stress test: For each deployment, for all workloads A to F, 15 runs of $10^5$ requests are performed. For each consecutive run, the number of threads in the YCSB VM is augmented with 5

[3]In a Zipfian distribution, popular records are read or updated most. In a Latest distribution, the most recently updated records are read most

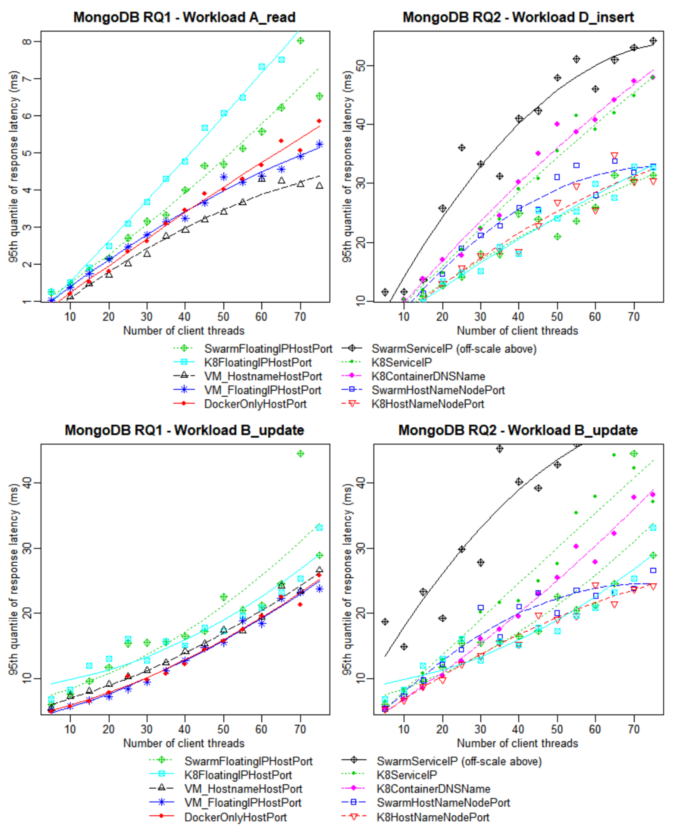

Figure 1. MongoDB scalability graph for RQ1 and RQ2

additional threads. We measure the mean, median and 95th quantile of response latency at the YCSB VM.

## D. MongoDB Results

We use MongoDB version 3.2.4 in all database deployments. The size of the dataset (23 MB) is chosen very small so all operations can be performed in memory. As such, the YCSB workloads will mostly stress the CPU as this is intended for answering the research questions.

**Main findings for RQ1.** All compared deployments use a *HostPort* as endpoint configuration. The only differ according to the used NodePlatform (*VM*, *DockerOnly*, *Swarm* or *K8s*) and the type of *CloudProvisioned* endpoint (FloatingIP or HostName). The *DockerOnly* deployment uses directly the network stack of the node's OS. *K8* deployments are configured with the Weave Net network plugin[33]. In Swarm, no separate virtual network needs to be installed for exposing containers via a *HostPort*. Instead, traffic for host ports is redirected by the default installed virtual network `docker_gwbridge`[12].

The experimental findings are presented as a series of scalability graphs (see Figure 1), a summarizing bar plot (see Figure2) and a data analysis table (see Table II). For example, the four graphs in Figure 1 compare database deployments with respect to a specific RQ for a specific pair of a YCSB Workload and an operation (insert, read, update, etc.). For each deployment in such graph, the measurements of the 95th

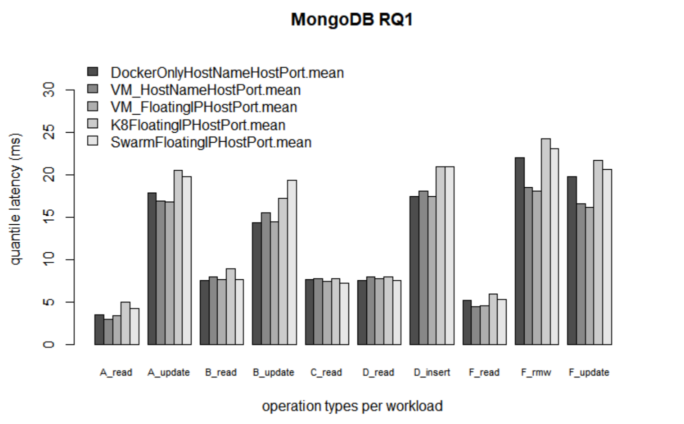

Figure 2. Summarizing bar plot for RQ1

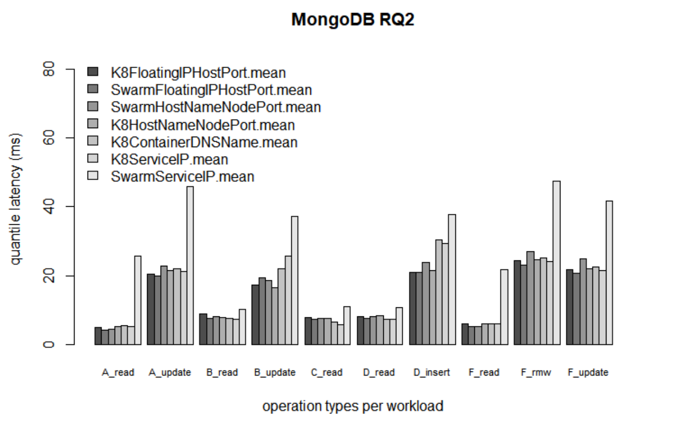

Figure 3. Summarizing bar plot for RQ2

| Operation | Update-heavy workloads (A, F) | Read-heavy workloads (B,C,D) |
|---|---|---|
| Read | 3.7,VM_HostNameHostPort<br>4.0,VM_FloatingIPHostPort<br>4.3,DockerOnlyHostName<br>4.8,SwarmFloatingIPHostPort<br>5.5,K8FloatingIPHostPort | no reoccuring rankings found; thus no significant differences found between deployments |
| Update/ Insert | 16.5,VM_FloatingIPHostPort<br>16.7,VM_HostNameHostPort<br>18.8,DockerOnlyHostName<br>20.2,SwarmFloatingIPHostPort<br>21.1.K8FloatingIPHostPort | 15.9,VM_FloatingIPHostPort<br>15.9,DockerOnlyHostName<br>16.8,VM_HostNameHostPort<br>19.1,K8FloatingIPHostPort<br>20.2,SwarmFloatingIPHostPort |

Table II
DATA ANALYSIS TABLE FOR MONGODB RQ1. COMMON RANKINGS OF DEPLOYMENTS. EACH DEPLOYMENT IS PRE-ANNOTATED WITH THE MEAN OF THE OBSERVED 95TH QUANTILES OF RESPONSE LATENCY (MS) FOR THAT PARTICULAR TYPE OF WORKLOAD AND TYPE OF OPERATION.

quantile of the response latency for each of the 15 runs of the scalability experiment are plotted as datapoints, and a regression line is generated in R using the `lm()` function with a standard quadratic equation. Furthermore, all graphs for all (workload,operation) pairs are summarized by computing the mean of the 15 datapoints of each deployment (which is represented as a bar plot in Figure 2). Finally, we can identify patterns of reoccurring rankings of deployments from this barplot and classify them. For MongoDB, we could clearly classify three different patterns of common rankings between deployments. Table II gives an overview of these three patterns and shows that they directly map to different combinations of two workload types (read-heavy vs write-heavy workloads) and two operations types (read vs update/insert). For each of these 3 common patterns of ranking, we can then also quantify the performance overhead of CO-based deployments with respect to VM-based deployments by taking the average of the means of each related workload and operation pair (see Figure 2).

*Results for RQ1:* With respect to read operations, *DockerOnly* in host mode does not introduce any significant overhead. The performance overhead of the *CO*-based deployments is significant for update-heavy workloads A and F (37% for *K8* and 20% for *Swarm*), but not significant for read-heavy workloads (B,C,D) because no common rankings could be found.
With respect to insert or update operations, the performance overheads of *Docker*, *Swarm* and *K8* for update-heavy workloads are respectively 13%, 22% and 28%. For read-heavy workloads, *Docker* performs better than the VM-based deployments (-5%), while *Swarm* and *K8* respectively incurred a performance overhead of about 27% and 20%.
We also studied the impact on scalability. Only, for read operations of workload A and update operations of workload B, we observed a deterioration of the scalability of the *K8* and *Swarm* deployments (see two left graphs of Figure 1). The overhead of the CO frameworks is so substantial because the CO frameworks don't allow containers to have access to the underlying host's network stack (i.e., host mode networking) as is possible in the *DockerOnlyHostNameHostPort* deployment.
**Main findings for RQ2** (see Figure 3 and Table III)**.** The compared deployments are all based on a *CO*-based Node-Platform (*Swarm* or *K8*) with either *CloudProvisioned* or *ClusterProvisioned* endpoints. For the latter type of endpoints, the Weave NET plugin has been used in *Swarm* and *K8*, except in the *SwarmHostNameNodePort* deployment where a NodePort is connected to the default `ingress` network, which uses Swarm's default `overlay` network plugin[12].

*Results for RQ2:* First of all, the performance overhead of the *SwarmServiceIP* deployment is exceptionally large across all workloads.
For all the other deployments, we can identify four patterns of reoccuring rankings of deployments in terms of the 95th response latency. With respect to read operations, there is no significant performance overhead of the CO deployments with *ClusterProvisioned* endpoints for update-heavy workloads (A,F); for read-heavy workloads (B,C,D), the *K8ServiceIP* and *K8ContainerDNSName* deployments perform slighly better than the CO deployments with *CloudProvisioned* endpoints.
With respect to insert or update operations, we structure the summary of the observed performance overheads per CO

| Operation | Update-heavy workloads (A, F) | Read-heavy workloads (B,C,D) |
|---|---|---|
| Read | 4.8,SwarmFloatingIPHostPort | 6.7,K8ServiceIP |
| | 4.9,SwarmHostNameNodePort | 7.1,K8ContainerDNSName |
| | 5.5,K8FloatingIPHostPort | 7.5,SwarmFloatingIPHostPort |
| | 5.5,K8ServiceIP | 7.9,K8HostNameNodePort |
| | 5.6,K8HostNameNodePort | 7.9,SwarmHostNameNodePort |
| | 5.7,K8ContainerDNSName | 8.2,K8FloatingIPHostPort |
| | 23.6,SwarmServiceIP | 10.6,SwarmServiceIP |
| Update/ Insert | 20.2,SwarmFloatingIPHostPort | 18.9,K8HostNameNodePort |
| | 21.1,K8FloatingIPHostPort | 19.1,K8FloatingIPHostPort |
| | 21.3,K8ServiceIP | 20.2,SwarmFloatingIPHostPort |
| | 21.8,K8HostNameNodePort | 21.3,SwarmHostNameNodePort |
| | 22.2,K8ContainerDNSName | 26.1,K8ContainerDNSName |
| | 23.8,SwarmHostNameNodePort | 27,5,K8ServiceIP |
| | 43.8,SwarmServiceIP | 37.5,SwarmServiceIP |

Table III
DATA ANALYSIS TABLE FOR RQ2

deployment and endpoint type: the performance overhead of the *SwarmHostNameNodePort* deployment is about 18% for update-heavy workloads but only 5% for read-heavy workloads. For *K8* deployments, the performance overhead of *K8ServiceIP* and *K8ContainerDNSName* is substantially larger for read-heavy workloads (40%) than for update-heavy workloads (3%). At the same time, the *K8HostNameNodePort* deployment performs the best with respect to update operations in read-heavy workloads.

With respect on the impact on scalability, the *SwarmServiceIP* deployment exhibits serious scalability problems as many datapoints for this deployment went off-scale in comparison to the other deployments (e.g see the left graph of Figure 1). For update/insert operations of read-heavy workloads B and D, we also observed a small deterioration of the scalability of the *K8ServiceIP* and *K8ContainerDNSName* deployments (see right graphs of Figure 1).

**Main findings for RQ3** (see Figure4)**.** The compared deployments are all *Swarm* or *K8*-based deployments with *ClusterProvisioned* endpoints. The deployments differ in the used network plugin. For K8, we compare the often used **Flannel** network plugin with the Weave NET plugin. For Swarm we compare the default **overlay** network plugin with the Weave NET plugin.

*Results for RQ3:* With respect to read operations, we structure the findings for *K8* and *Swarm* separately. For *K8*, there are no significant differences between the Weave NET and the flannel plugin for both update-heavy and read-heavy workloads.

For *Swarm*, despite the large overhead of the *SwarmServiceIP_weave* deployment in RQ2, the default overlay plugin consistently performs 7% worse than the Weave Net plugin for read operations of read intensive workloads (B, C, D). This overlay plugin also exhibits a deteriorated scalability behavior (see bottom graph of Figure 4). On the other hand, with respect to update operations (not including insert operations), the *SwarmServiceIP_overlay* deployment excells across all workloads. It performs twice as fast as the *SwarmServiceIP_weave* deployment and about 9% faster than the *K8ContainerDNSName_weave* deployment.

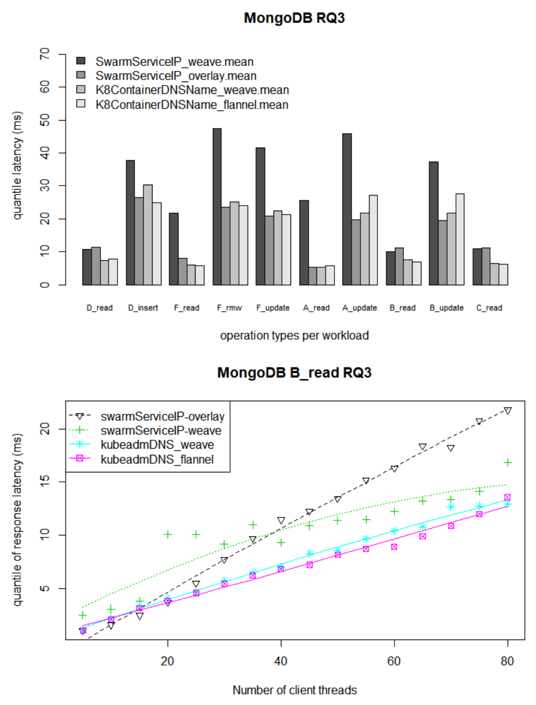

Figure 4. Comparison of CO-based MongoDB deployments with different network plugins

## V. DISCUSSION

In this paper we have evaluated the performance overhead of container orchestration systems for deploying and managing MongoDB database clusters on top of an OpenStack cloud. As base line we compare with pure VM-based deployments of these databases. It was first shown that for **CPU-bound database workloads** (such an in-memory MongoDB database deployment), the performance overhead of container orchestration (CO) systems is quite significant in comparison to a *DockerOnly* deployment that runs in host mode (i.e. to have access to the underlying host's network stack). Secondly, for some YCSB (workload, operation) pairs, the scalability of CO-based databases is deteriorated. Thirdly, CO-based databases with *CloudProvisioned IP endpoints and HostPorts* cause in general less performance overhead than CO-databases with *ClusterProvisioned endpoints*; it is expected that this overhead can be further reduced substantially by allowing containers of services to run in host mode.

There are however a number of disadvantages of *CloudProvisioned* endpoints: (i) automated support for TCP/IP location transparancy, by re-assigining the stable *FloatingIP* address or *HostName* from a failed VM to a replacing VM, still needs to be implemented by the underlying cloud orchestration platform; (ii) an additional operational management cost includes the avoidance of host port conflicts with other services running on the same node; (iii) another management cost is that container access control policies must be correctly specified in order to regulate which containers are allowed access to the host's networking stack.

*ClusterProvisioned* endpoints, on the other hand, come with a fully automated and self-contained approach to TCP/IP location transparency without operational management costs such as avoidance of host port conflicts and controlling access to the underlying host's networking stack. As such efforts to reduce the performance overhead of CO systems with *ClusterProvisioned* endpoints are worthwhile. With this end in view, an interesting finding with Kubernetes is that different *ClusterProvisioned endpoints* exhibit a different performance overhead. These variations in performance overhead have been found to correlate with the specific workload type (read-heavy vs update-heavy) and the type of operation (read or update operation). More specifically, *ServiceIP and ContainerDNSName* endpoints perform on the same par as *CloudProvisioned* endpoints, except for update operations in read-heavy workloads where *NodePorts* are better performing. For Docker Swarm, the most dominating factor that determines the performance overhead of *ClusterProvisioned* endpoints is the choice of network plugin.

## Acknowledgment

We thank Vincent Reniers for developing the script generator for running multiple YCSB experiments and R-scripts for statistical analysis of YCSB results. This research is partially funded by the Research Fund KU Leuven and the IWT-SBO DeCoMAdS project.

## References

[1] Apache. Mesos Documentation. http://mesos.apache.org/documentation/latest/index.html, 2018. Accessed: February 14 2018.

[2] Kapil Arya. Networking for Mesos-managed containers. https://github.com/apache/mesos/blob/0.25.0/docs/networking-for-mesos-managed-containers.md. Accessed: August 16 2017.

[3] E. Bekas and K. Magoutis. Cross-layer management of a containerized nosql data store. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1213–1221, May 2017.

[4] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[5] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[6] Container Network Interface (CNI) specification. Port-mapping plugin. https://github.com/containernetworking/plugins/tree/master/plugins/meta/portmap, accessed: 22-05-2018, 2018.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, pages 143–154, 2010.

[8] Sandeep Dinesh. Running MongoDB on Kubernetes with StatefulSets. http://blog.kubernetes.io/2017/01/running-mongodb-on-kubernetes-with-statefulsets.html, 2017. Accessed: May 24 2017.

[9] Docker. Swarm mode overview. https://docs.docker.com/engine/swarm/. Accessed: February 14 2018.

[10] Docker. Use overlay networks. https://docs.docker.com/network/overlay/. Accessed: February 14 2018.

[11] Docker. Use the OverlayFS storage driver – OverlayFS and Docker performance. https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/#overlayfs-and-docker-performance, 2017. Accessed: August 9 2017.

[12] Gary Duan. How Docker Swarm Container Networking Works – Under the Hood. https://neuvector.com/blog/docker-swarm-container-networking/, 2017. Accessed: March 2 2018.

[13] Eddy Truyen et al. Container orchestration for deployment and management of NoSQL database clusters. https://goo.gl/EeGPR8. Accessed: August 18 2017.

[14] Miguel G. Xavier et al. A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 299–306, 2014.

[15] Miguel G. Xavier et al. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 253–260, 2015.

[16] Tomasz Napierala et al. Networking design proposal. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/network/networking.md, 2018. Accessed: May 22 2017.

[17] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[18] Red Hat. OpenShift Container Platform 3.7 Documentation. https://docs.openshift.com/container-platform/3.7/welcome/index.html. Accessed: February 14 2018.

[19] Benjamin Hindman, Andy Konwinski, A Platform, Fine-Grained Resource, and Matei Zaharia. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI 2011)*. USENIX Association, 2011.

[20] Nane Kratzke. A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms. *Open Journal of Mobile Computing and Cloud Computing*, 1(2):17–30, 2014.

[21] Nane Kratzke. About microservices, containers and their underestimated impact on network performance. In *Proceedings of CLOUD COMPUTING 2015 (6th. International Conference on Cloud Computing, GRIDS and Virtualization)*, pages 165–169, 2015.

[22] Nane Kratzke and Peter-Christian Quint. How to operate container clusters more efficiently? some insights concerning containers, software-defined-networks, and their sometimes counterintuitive impact on network performance. *International Journal On Advances in Networks and Services*, 8(3&4):203–214, 2015.

[23] Kubernetes. Cluster Networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/#flannel. Accessed: August 16 2017.

[24] Kubernetes. Kubernetes Documentation. https://kubernetes.io/docs/home. Accessed: February 14 2018.

[25] Kubernetes. StatefulSets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. Accessed:March 6 2018.

[26] Kubernetes. Using kubeadm to Create a Cluster. https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/, 2018. Accessed: February 16 2018.

[27] Mesosphere. CNI plugin support. https://docs.mesosphere.com/1.10/networking/virtual-networks/cni-plugins/, 2018. Accessed: February 14 2018.

[28] Stephen Nguyen. Ridiculously Fast MongoDB Replica Recovery. https://dzone.com/articles/ridiculously-fast-mongodb-replica-recovery-part-1, 2016. Accessed: June 30 2017.

[29] OpenStack. OpenStack User Survey Report November 2017. https://www.openstack.org/assets/survey/OpenStack-User-Survey-Nov17.pdf. Accessed: January 23 2018.

[30] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16. ACM, 2016.

[31] MongoDB (TM). Running MongoDB as a Microservice with Docker and Kubernetes. https://www.mongodb.com/blog/post/running-mongodb-as-a-microservice-with-docker-and-kubernetes, 2016. Accessed: August 2 2017.

[32] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. *Eurosys*, 2015.

[33] weaveworks. Weave Net. https://www.weave.works/oss/net/, 2018. Accessed: February 14 2018.

[34] Mingwei Zhang, Daniel Marino, and Petros Efstathopoulos. Harbormaster: Policy Enforcement for Containers. *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 355–362, 2015.