Kubernetes Container Orchestration

Udhav Sethi
University of Waterloo
udhav.sethi@uwaterloo.ca

Xiyang Feng
University of Waterloo
x74feng@uwaterloo.ca

ABSTRACT

Containerization is taking hold in the datacenter. To leverage the benefits of an isolated environment provided by containers while utilizing resources efficiently, orchestration becomes necessary. Kubernetes is a container orchestration system that allows better tracking, scheduling and operationalizing of containers at scale and eliminates infrastructure complexity. We study and evaluate Kubernetes performance against native application deployment for different workloads. We also conduct a case study on running MongoDB on Kubernetes and discuss the benefits and trade-offs.

KEYWORDS

Kubernetes, Containers, Distributed Systems

1 INTRODUCTION

In traditional application deployments, applications developed in one computing environment often run into bugs and errors when deployed in another environment. The differences in network topology, storage management, and security policies are some of the variables that lead to issues in the application execution. Containerization attempts to solve this problem by providing an isolated runtime environment to run the application. The application is bundled together with its dependencies, libraries, binaries, and configuration files into a single package and is run inside a container. Thus, containerization abstracts away the differences in the underlying infrastructure.

Since each container operates independently of others, containers need to be monitored and orchestrated to track all the moving parts and ensure their efficient functioning. Kubernetes is an open-source container-orchestration layer for automating application deployment, scaling, and management.[2] It groups containers that make up an application into logical units for easy management and discovery. Kubernetes enables deployment of a cluster that is highly available, uses resources efficiently, self-heals in case of failures, and scales automatically.

In this paper, we study and evaluate the Kubernetes system. We try to understand its basic functioning, scheduling, and fault-tolerance mechanisms. For evaluation, we first compare the performance of the Kubernetes system in basic tasks, such as ping, CPU intensive and memory intensive workloads against native machine deployments. We then conduct a case study on running the MongoDB application on Kubernetes and investigate the motivation and challenges involved in deployment of such a system. For evaluation, we conduct failover testing on the cluster and also benchmark the performance of the system for the YCSB (Yahoo! Cloud Serving Benchmark) workload.

2 KUBERNETES ARCHITECTURE

Kubernetes is built to manage a large number of containers running in a distributed system resiliently. The various components that form the Kubernetes architecture are outlined in Figure 1 and described in the following sections.

2.1 Kubernetes Clusters

Kubernetes coordinates a cluster of computers running containerized applications and automates the distribution and scheduling of application containers across a cluster in a more efficient way. A Kubernetes cluster consists of a master and workers. The master coordinates all activities in the cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates. A worker machine runs the application. The Kubernetes architecture is outlined in Figure 1.

2.2 Deployments

A Deployment instructs Kubernetes how to create application instances onto individual Nodes in the cluster. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster. This provides a self-healing mechanism to address machine failure or maintenance.

2.3 Pods

A Pod is a Kubernetes abstraction for a "logical host" that represents a group of one or more application containers, and some shared resources (such as storage, IP address, ports) for those containers. Containers in a Pod are always co-located and co-scheduled, and run in a shared context on the same Node. A Pod is an atomic unit on the Kubernetes platform.

2.4 Nodes

A Node is a worker machine in Kubernetes and may be either a virtual or physical machine. Every Node has a Kubelet, an agent for managing the pods running on the node and communicating with the Kubernetes master, and a container runtime, responsible for pulling the container image from a registry, unpacking the container, and running the application.

2.5 Services

A Service is an abstraction which defines a logical set of Pods and enables loose coupling between them. Services enable external traffic exposure, load balancing and service discovery for those Pods.

2.6 Scaling

Horizontal Scaling is accomplished by changing the number of replicas in a Deployment, which creates new Pods and schedules them to Nodes with available resources. Kubernetes also supports autoscaling of Pods, where the number of pods are automatically

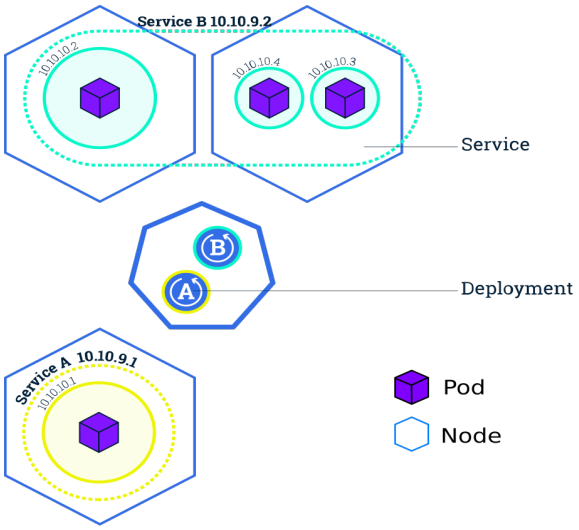

Figure 1: Kubernetes Architecture

scaled up based on observed CPU utilization or other application-provided metrics.

2.7 Updates

Kubernetes provides rolling updates, which allows Deployments to be updated with zero downtime by incrementally updating old Pod instances with new ones. If a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update.

3 NETWORKING

Kubernetes dictates particular requirements on any networking implementation.[7]. It is specific about its choices on how Pods are networked. We study these networking models in the sections below.

3.1 Container-to-Container Networking

Each pod running on a node is assigned its own network namespace. All the containers within a pod are assigned the same IP address and port space through the namespace of the Pod that encapsulates them. Since they reside in the same namespace, they are able to communicate with each other through localhost.

3.2 Pod-to-Pod Networking

Since all the pods have their own network namespace, they communicate with other pods using the IP address assigned to them. The mechanism differs based on if the two pods reside on the same node or different nodes.

3.2.1 Pods on same Node. The network namespace assigned to a pod isolates it to its own networking stack. Since all the pods in a node are connected to the root namespace of the node, they go through it to communicate with each other. For this, they use a network bridge that maintains a forwarding table between sources and destinations and connects the namespaces of each of the pods to the root namespace.

3.2.2 Pods across Nodes. In Kubernetes, the IP address of a pod is always visible to other pods in the network. The pods residing on different nodes can reach each other through their IP addresses. A pod that wants to communicate with a pod on another node forwards the packet to the network bridge of the root namespace, from where it enters the network. The network forwards the packet to the root namespace of the destination Node, where it is routed through the bridge to the namespace of the destination pod.

3.3 Pod-to-Service Networking

Pods are mortal. Pods may be added to or removed from nodes while scaling the application or in case of node failures. A Kubernetes Service solves this problem by tracking the changing pod IP addresses. A service is assigned a virtual IP address which does not change over time. To reach the set of pods associated with a service, the traffic is forwarded to the virtual IP of the service. Kubernetes maintains an internal load balancer that distributes the load among these pods. Thus, in terms of networking, the service acts like a black box that encapsulates all the pods associated with it.

3.4 Internet-to-Service Networking

A Kubernetes Service is generally created by specifying a load balancer which is provided by the cloud controller. The IP address of this load balancer is used to direct external traffic to the service. To get traffic from a service out to the Internet, it is forwarded to an Internet gateway which forwards it to the the public Internet.

4 SCHEDULING

The scheduling framework in Kubernetes attempts to schedule a Pod in two phases, the scheduling cycle and the binding cycle.

Every Pod that is newly created or unscheduled gets added to a Pod queue. In the scheduling cycle, a pod is dequeued from the queue and the scheduler is responsible for finding the optimal Node for that Pod to run on. Since every Pod and every container in the Pod may have different requirements, it is important to first find the nodes that are capable of running the Pod, and then choosing one of them. The scheduler selects a Node for the Pod in 2 steps: filtering, and scoring. These are described in sections 4.1 and 4.2 below.

4.1 Filtering Step

The filtering step finds the set of Nodes where it's feasible to schedule the Pod. The scheduler follows a defined set of filtering policies to find the list of suitable Nodes. Filtering policies are hard constraints and cannot be violated while finding feasible nodes. Some of these policies are listed below:

- Matching the hostname specified by the Pod.
- Checking if the Node meets the resource requirements like CPU and Memory specified by the Pod

- Checking if the ports requested by a Pod are free on the Node.
- Matching the Volumes requested by the Pod to the mounted volumes on the Node.
- Fitering out the Nodes reporting storage/memory/PID pressure.
- Fitering out nodes with completely full filesystem.
- Marking a node tainted so that no pods can schedule onto it unless a Pod explicitly tolerates the taint.

4.2 Scoring Step

Once a list of suitable nodes is determined, the scoring step scores the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. Scoring policies are soft constraints and are followed on a best-effort basis. Some of these policies are a part of the Kubernetes scheduler's default behavior, while others are user-configurable. Some commonly used policies are listed below:

- Favouring nodes with fewer/more requested resources.
- Spreading pods among nodes.
- Prioritizing nodes according to pod affinity/anti-affinity.
- Balancing out the resource utilization of nodes.
- Favouring nodes that already have the container images for that Pod cached locally.
- Co-locating services that communicate frequently.
- Simply allocating equal weights to all nodes.

On having selected an optimal Node for running a Pod, the binding cycle applies that decision to the cluster. A scheduling or binding cycle can be aborted if the Pod is determined to be unschedulable or if there is an internal error. The Pod will be returned to the queue.

5 FAILURE HANDLING

Kubernetes features a self-healing mechanism so that the system remains highly available. The Replication Controller feature helps Kubernetes achieve high availability through failures. We will talk about the Replication Controller feature and the various failure scenarios in the following sections.

5.1 Kubernetes Replication Controller

The Replication Controller feature of Kubernetes is responsible for managing the pod lifecycle through supervision of multiple pods across multiple nodes.[3] A Replication Controller ensures that a specified number of pod replicas are up and running at any given time. Depending upon the number of pods specified for the application, the Replication Controller adds more pods or terminates extra pods.

The motivation behind replicating application containers is to achieve reliability, scalability, and load-balancing for the application. The goal of Replication Controller is that the services and their clients should remain oblivious to the underlying process of maintaining a fixed number of replicas in the cluster. The Replication Controller makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply updating the replicas field.

5.2 Worker Failure

The kubelet on each worker node posts a node status to the master every fixed interval of time. The frequency is determined by the node status update frequency (default=40s). If the node is unresponsive for a grace period (default=40s), the node is marked as unhealthy. Once the node is marked unhealthy for longer than the pod eviction grace period (default=5m), all the pods on the node are marked for eviction. Thereafter, no traffic is routed to the pods marked for eviction.

If a node crashes, the above mechanism will kick in and the pods scheduled on the failed node will be evicted from the cluster. In such a scenario, the Replication Controller tries to ensure that the desired number of pods matches its label selector and are operational. It adds the required additional pods to the available healthy nodes in the cluster so that the cluster state is restored.

5.3 Master Failure

The master is responsible for managing and coordinating the cluster. If the master goes down, the cluster is no longer able to create new resources, reschedule pods among worker nodes, or respond to worker node failures. Since the other services such as DNS, load balancing etc. continue to function, the application continues to function normally in the absence of failures. The auto-scaling, self-healing nature of the cluster is no longer operational.

5.4 Network Partition

In case of a network partition between the master and a worker node, no traffic is able to be sent to or received from the worker node. In such a scenario, a similar mechanism to worker node failure causes the Replication Controller to perceive those pods to be terminated or deleted. The unreachable node is dropped from the cluster and all its pods are rescheduled to other hosts.

6 EXPERIMENTS

We designed and conducted a set of basic experiments to evaluate the performance of Kubernetes cluster. The application containing these tests was deployed in 3 different environments - a Kubernetes cluster, a Docker container, and bare metal. For each test, we ran the experiment 100 times and recorded the latency values for the average, 10th percentile and 90th percentile case. The experiments are outlined as follows:

6.1 Ping Test

In the ping test, a simple "ping" request is sent to the application. The application responds with a simple acknowledgement. This test helps us get an idea of the overall overhead introduced by Kubernetes in the request response path.

6.2 Memory Intensive Test

We send a request to the application to perform a memory-intensive task. The request specifies an amount of memory to be allocated and also defines the time interval `t` for which to define it. The application creates and fills a data structure to occupy the specified size of memory, sleeps for `t` seconds, and then frees the memory. For our tests, we recorded latencies for requests with memory values

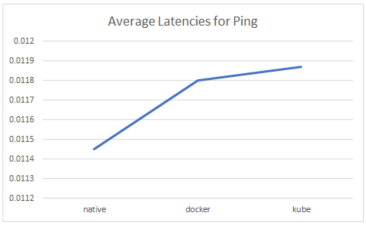

Figure 2: Ping Tasks

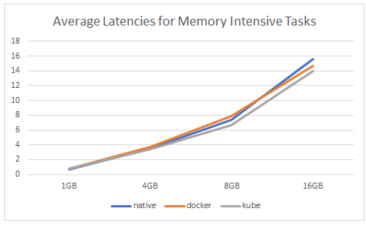

Figure 3: Memory Intensive Tasks

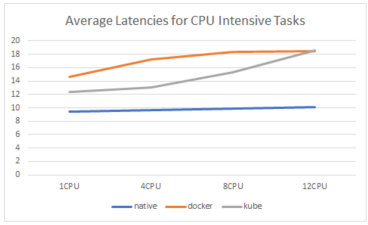

Figure 4: CPU Intensive Tasks

of 1GB, 4GB, 8GB, and 16GB. Each size of memory is allocated for `t=0` seconds, which means the memory is freed right at the end of the time needed to allocate it.

6.3 CPU Intensive Test

For the CPU intensive workload, we compute large exponents of a number. We also specify the number of CPUs we wish to run our computation on. For `n` number of CPUs, the computation is done `n` times parallely. We record latencies for requests with 1, 4, 8, and 12 CPUs.

6.4 Results

We observe latency overhead in the Kubernetes deployment of the application as opposed to bare metal. This overhead can be attributed to the runtime performance cost of Docker containers, caused due to port mapping between the host and the container. The results are outlined in Figures 2, 3 and 4.

7 CASE STUDY - RUNNING MONGODB ON KUBERNETES

MongoDB is a powerful cross-platform NoSQL database application. It is widely used by applications that require flexible data model, scalability and strong performance. Ever since the birth of Kubernetes, there has been a line of work on deploying and running MongoDB with Kubernetes done by the open source community. This section will first look into the motivation and challenges of running MongoDB on Kubernetes. It will then cover a detailed deployment process as well as a performance comparison with various workload between native MongoDB cluster and Kubernetes MongoDB cluster.

7.1 What Kubernetes can offer

As one of the most widely used orchestration frameworks, Kubernetes combines the advantages of containers and container scheduling system [5]. Below are some benefits of using containerized applications.

- **Isolation**. In Kubernetes, each container runs an application in a completely isolated environment from other containers so that developers do not need to worry about the conflict of configuration between applications. Such isolation also improves the stability of the cluster since the crash of one application cannot affect applications running in a separate container.
- **Automated Deployment**. Kubernetes supports elastic scaling of container instances. In case developers want to add/remove containers or do an upgrade on the application, Kubernetes can perform scaling and rolling upgrade automatically without taking down the whole cluster.
- **Automated Rescheduling of Failed Containers**. As described in Section 5, the powerful failure recovery mechanism is arguably the most important motivation of running MongoDB on Kubernetes. Consider a native MongoDB cluster with replication set of 3 nodes, in case of node failure, the database administrator has to manually create a new MongoDB instance, configure it with the same setup as other node in the replication set and join it back to the cluster which could be quite a complicated process. However, by using Kubernetes, such node failures can be handled automatically without any admin interference.

7.2 Challenges

Running MongoDB on Kubernetes is not trivial [6]. This section looks into some of the additional considerations introduced by Kubernetes.

- **Stateful Application**. MongoDB database itself is a stateful application which means a node needs to remember its previous state especially for the data after coming back from failure. However, in contrast, containers are mortal. Everything including storage will disappear when a container goes down. In order to solve this, a persistent volume needs to be attached to MongoDB application containers.
- **Network Communication**. Nodes within the same MongoDB replication set need to communicate with each other in order to perform back up and synchronization. This implies that every node needs to know each other's network address. However, in a normal Kubernetes service, a pod will be assigned a different ID as well as IP address after coming back from failure which is problematic for communication within the replication set.
- **MongoDB Initialization**. After setting up an individual MongoDB instance, developers still need to initialize the MongoDB replication set by executing command inside mongo shell on one the instances. This indicates that developers need to find a way to directly talk to the application running inside the container.

Cluster Domain	Service	StatefulSet
cluster.local	default/ngnix	default/web
StatefulSet Domain	**Pod DNS**	**Pod Hostname**
nginx.default.svc.cluster.local	web-{0..N-1}.nginx.default.svc.cluster.local	web-{0..N-1}

Table 1: Example Network ID

7.3 Kubernetes StatefulSet

In order to address the challenges mentioned above, Kubernetes introduces a new concept called StatefulSet [4] which is a workload API object used to manage stateful applications. StatefulSet guarantees a stable and unique network together with a stable and persistent storage for the pods running in the cluster.

7.3.1 Stable Network ID. Once a pod is created, it will be assigned with a unique integer ordinal within the StatefulSet. The pod then derives its hostname by following the pattern *$(statefulset name)-$(ordinal)*. A StatefulSet also has a Headless Service that controls the domain of its pod. The domain of this service is given in the form of *$(service name).$(namespace).svc.cluster.local*. And each Pod will be assigned a DNS of the form of *$(pod name).$(service domain)*. Table 1 shows an example Stable Network ID for a Kubernetes StatefulSet.

7.3.2 Stable Storage. Kubernetes creates one persistent volume for each persistent volume claim. The pod will always be mounted on the persistent volume associated with its persistent volume claim even after coming back from crash. Details about creating persistent volume in Kubernetes will be addressed in the next section.

7.3.3 Limitations. The current implementation of Kubernetes StatefulSet still has the following limitations.

- **Storage Creation** Currently, there are two ways to create persistent storage for a StatefulSet. One is to provision storage through Persistent Volume Provisioner which means in order to enable dynamic storage allocation, developers have to configure the cluster with the Provisioner's service first. The other way is to pre-provision enough static persistent volume and allocate to the MongoDB application running in a StatefulSet which does not scale well. Both methods require enormous time and effort for the administrator to configure.
- **Storage Recollection** Removing a StatefulSet does not delete its associated persistent storage. Kubernetes claims that this is desirable since data safety is more important than resource recollection. However, in the case that developers do not want the associated storage, deletion has to be done manually.
- **StatefulSet Deletion** Deleting a StatefulSet does not guarantee the termination of all pods within the StatefulSet. The solution suggested by Kubernetes is to first scale down the StatefulSet to size 0 and then delete. This will make sure that all pods are terminated prior to deletion.

7.4 Deployment Setup

We utilize 4 nodes named yellow 12-15 on the University of Waterloo SYN cluster, each of which runs Ubuntu 16.04. All the nodes share 64GB of memory, a 12 core CPU and similar network stack so that they can communicate with each other freely. Two MongoDB replication sets are deployed on yellow 13-15. Yellow 12 is used as the testing server so no application runs on it.

We first deploy a native MongoDB replication set using 3 nodes with yellow 14 as the primary set. All configurations are set to default. User can use MongoDB service by accessing the address *yellow14:27017*. Note that MongoDB does not allow direct writing on secondary replicas so exposing primary replication address is sufficient.

We then deploy an identical MongoDB replication set with Kubernetes. Yellow 13 is used as the Kubernetes master and yellow 14 is chosen as the primary replication which exposes its service as type NodePort. For the Kubernetes deployment, we use the StatefulSet approach with dynamic storage allocation as described in the previous section. User can use MongoDB service by accessing *yellow13:$(exposed service port)*.

7.5 Failover Test

A failover test is conducted on Kubernetes MongoDB replication set in order to understand how Kubernetes handles pod failure in a MongoDB replication set.

The initial state is that 3 pods, each of which have a MongoDB instance, named db-0, db-1 and db-2 are running in the cluster with db-0 being the primary replication. When we take down the pod db-0, MongoDB notices that the primary replication goes down and elects db-1 as the new primary replication. Meanwhile, Kubernetes notices that one of the pods goes down so it will start the failover mechanism by creating a new pod and joining it back to the cluster as a secondary replication. The new pod will have the same network identity within the StatefulSet. In the final state after failure recovery, there are 3 pods with the same name db-0, db-1 and db-2, and db-1 serves as the primary replication.

7.6 Benchmark

Apart from failover test, we also run various workloads on the two clusters. This section will first introduce the workload followed by an evaluation of the performance of two clusters.

7.6.1 YCSB workload. The Yahoo! Cloud Serving Benchmark (YCSB) [1] is a widely used benchmarking tool to measure database application performance. We test 6 different workloads as described in the following.

- **Workload A: Update heavy workload**. 50% of read and 50% of write. Application example: session store recording recent actions.
- **Workload B: Read mostly workload**. 95% of read and 5% of write. Application example: photo tagging.

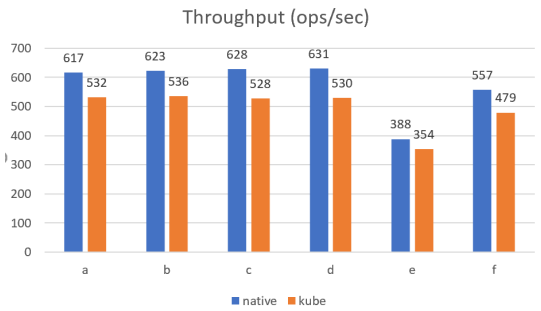

Figure 5: Throughput

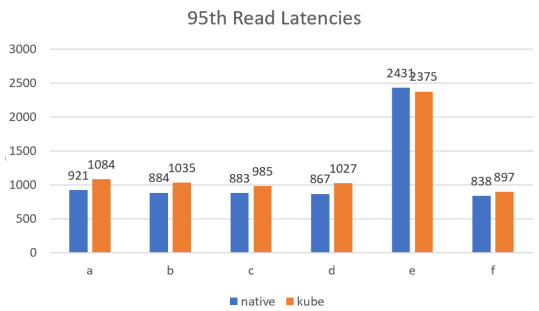

Figure 6: 95 Percentile Read Latency

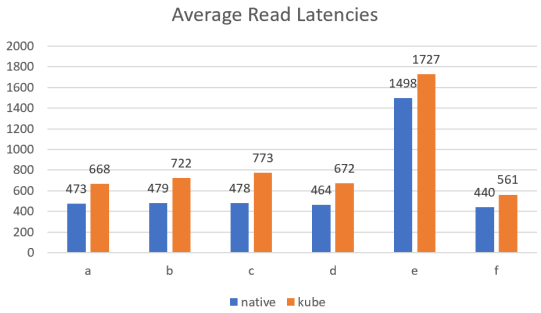

Figure 7: Average Read Latency

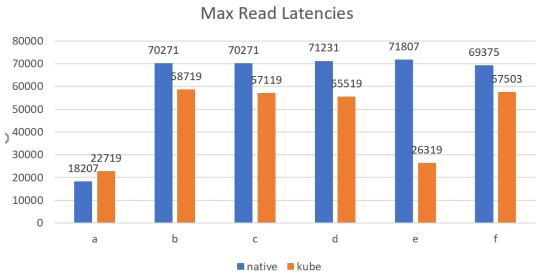

Figure 8: Maximum Read Latency

- **Workload C: Read only workload**. 100% of read. Application example: user profile cache.
- **Workload D: Read latest workload**. Newly inserted records are more likely to be read. Application example: user status update.
- **Workload E: Short ranges**. Perform range queries only. Application example: geographic location scan.
- **Workload F: Read-modify-write**. Read a record, modify it and write back to the database. Application example: record update.

7.6.2 Results. Figures 6 and 7 show the 95 percentile read latency and average read latency on two clusters. Since the maximum latency is pretty high in our test, it may be a better reflection of cluster performance. We observe that Kubernetes in general has a higher read latency in most of the cases. The only exception is the range query workload (workload e) where linear scan become the dominating time. The results also match our previous result in the ping experiment where Kubernetes introduce certain amount of communication overhead compared to native clusters. Due to the communication overhead, the throughput in terms of operations per second for Kubernetes cluster is also smaller than native cluster as shown in Figure 5. It seems that developers are sacrificing a small part of performance in exchange for the powerful features provided by Kubernetes.

We also observe a result that is beyond our expectation where Kubernetes has a lower tail latency compared with native cluster as shown in Figure 8. We think this result may be due to our experiment setup. Since the native MongoDB cluster is running on 3 different nodes, when the primary replication wants to direct the read to a secondary replication, it has to jump between machines. While in Kubernetes cluster, although it also runs on 3 different nodes, the master node is configured as a non-work node to achieve better isolation. Therefore, 2 pods actually run on the same machine and have less communication overhead. As a result, tail latency is smaller in Kubernetes cluster.

8 CONCLUSION

We study Kubernetes as a container orchestration layer and dissect its features that enable it to provide high availability and easy deployment of distributed containerized applications. Through our experiments and benchmarks, we conclude that while Kubernetes has a performance impact on the application, it provides the tools needed to deploy quickly while staying available. These ideas improve the overall velocity of reliable software deployment.

REFERENCES

[1] Brian F. Cooper. 2010. *Yahoo! Cloud Serving Benchmark.* https://github.com/brianfrankcooper/YCSB
[2] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.".
[3] Kubernetes. 2019. *ReplicationControllers.* https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/
[4] Kubernetes. 2019. *StatefulSets.* https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[5] MongoDB. 2017. *Enabling Microservices Containers Orchestration Explained.* https://webassets.mongodb.com/mongodb_microservices_containers_orchestration.pdf
[6] Andrew Morgan. 2016. *Running MongoDB as a Microservice with Docker and Kubernetes.* https://www.mongodb.com/blog/post/running-mongodb-as-a-microservice-with-docker-and-kubernetes
[7] Kevin Sookocheff. 2018. *A Guide to the Kubernetes Networking Model.* https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/