Can Linux Containers Clustering Solutions offer High Availability?

Ali Kanso, Hai Huang
IBM T.J. Watson Research Center
IBM Research
Yorktown Heights, New York
{akanso, haih}@us.ibm.com

Abdelouahed Gherbi
Department of Software and IT Engineering,
Ecole de Technologie Superieure (ETS)
Montreal, Canada
abdelouahed.gherbi@etsmtl.ca

Abstract—Linux containers offer a lightweight virtualization solution that is based on sharing the Linux kernel among multiple containerized environments. Container clustering solutions manage containerized applications across multiple hosts. Such solutions maintain the high availability (HA) of those applications, by monitoring their health and reacting to their failures. In this paper, we examine the effectiveness of the HA mechanisms offered by such solutions, and discuss the limitations that still need to be addressed.

Keywords—Linux Containers; High Availability; Kubernetes; Testing; Clustering; Container Management.

I. INTRODUCTION

Linux containers are gaining significant interest in the cloud computing space. They are perceived as a lightweight replacement of virtual machines (VMs) [1]. Due to their negligible runtime overhead, they can have a much higher deployment density per physical host. Although virtual machines are still widely used in the Infrastructure as a Service (IaaS) space, we see Linux containers dominating the Platform as a Service (PaaS) landscape [2]. The need for management and orchestration systems capable of scheduling, deploying, updating and scaling such applications is crucial. This is especially true with the rise of the microservices architecture, where the software components composing an application are loosely coupled and have independent lifecycles and deployments [19]. By containerizing such components, we can leverage the container clustering solutions to accelerate the development and operations of microservices. Kubernetes [3] emerged as one of the first solutions to managed containerized applications across multiple hosts. Backed by Google, and used to power their Google Cloud Platform’s container service (Google Container Engine [4]), Kubernetes is getting the attention of a wide spectrum of users who are using it in their cloud environment (e.g. Red Hat’s OpenShift [5]).

The motivation of this work is to test the effectiveness of Kubernetes in managing HA. Kubernetes is one of many orchestration services for managing containers. The rationale behind selecting Kubernetes is its advanced HA features. In fact, Kubernetes offers three levels of monitoring to ensure the high availability of deployed applications. The first is at the host level, where Kubernetes monitors the nodes of the cluster through its distributed agents. The second is at the container level, where Kubernetes ensures through the container engine that the containers are healthy. And finally, Kubernetes introspects the containers through health checking probes to ensure the software components executing inside of the container are healthy.

In this work we present a test plan for examining the responsiveness of Kubernetes HA mechanisms, we benchmark its performance, and discuss its limitations. Accordingly, this paper is organized as follows: in Section 2 we discuss in more details the Kubernetes deployment model and its resource structural model. In Section 3 we present our testing framework. In Section 4 we discuss our experimental results. Section 5 surveys the related work and technologies, and finally we conclude the paper in Section 6.

II. BACKGROUND

In this section we present the conceptual model according to which Kubernetes manages the applications, and then present the architectural model of Kubernetes.

A. Kubernetes Conceptual Model

The finest level of granularity for the container management in Kubernetes is the pod. A pod consists of at least one regular container providing a given functionality and the infrastructure container. The latter is the container created by Kubernetes to hold the network and IPC (inter-process communication) namespaces shared by all the containers of the pod. This allows the restart of any other container in the pod while preserving the network setup in the infrastructure container. The deletion of the pod entails the deletion of all its containers.

A Kubernetes service represents a set of pods and the policy by which to access them. A replication controller ensures that a configurable number of pod replicas are running at any point in time in the cluster. It is also responsible for scaling up/down the number of replicas according to a scaling policy. The deletion of the replication controller entails the deletion of all its pods. Figure 1 shows a simplified unified modeling language (UML) class diagram of the Kubernetes resource model.

B. Kubernetes Architecture

A Kubernetes cluster comprises one or more Kubernetes master nodes, and the Kubernetes worker nodes, also referred
to as Minions. Each Minion runs a container engine, (in our case, it is Docker[7]), to instantiate the containers of a pod. The Kubelet is the Kubernetes agent that runs on each minion, and executes the instructions of the master. The Kubelet also monitors the pods on its node and reports back periodically to the master. Kubernetes supports the concept of kube-proxy, which acts as a distributed multi-tenant load-balancer deployed on each Minion. On the master side, Kubernetes has three main components: (1) the API-server, which has two main functions: first it provides the frontend to the shared state through which all other components interact. Second, it serves the CRUD (create, read, update, delete) operations on the Kubernetes objects. (2) The scheduler is responsible for the placement of the pods according to fitness and priority filters. (3) The controller-manager watches the shared state of the cluster through the API-server and makes the necessary changes in order to shift the current state towards the desired state. For instance, in case of a minion node failure, the controller-manager will ensure the pods that are part of a replication controller are re-instantiate on other healthy minions. Finally the shared cluster state is preserved in a distributed, strongly consistent, key/value store[2].

---

1 Kubernetes also supports other container engines such as Rocket.
2 Currently etcd [6] is the preferred key/value store for Kubernetes
3 Kubernetes currently recommends no more than 40 pods per node, however
4 Currently etcd [6] is the preferred key/value store for Kubernetes

---

**III. THE TESTING FRAMEWORKS**

In this section we present our testing framework. In particular, we explain our testing methodology and test plan, and then present our deployment environment.

**A. Testing methodology**

Our testing methodology consists of four steps as shown in Figure 2. Those steps are executed for each run of each experiment.

1) In the first step, in order to ensure that a preceding failure does not impact the system behavior in a subsequent test, we “sanitize” our testing environment after each run. To achieve that, we define the normal state of the system based on the initial deployment before any fault is injected, and then we take a snapshot of the system at this state. After each test we restore the system to the initial state.

2) In the second step we inject a fault at the desired level of testing. A failure is simulated by a Kill, Halt, or other abrupt termination commands that do not allow the graceful termination of an entity.

3) We instrument our failure injection scripts to time-stamp the failure events, and then we monitor the logs (Docker, Kubernetes, etcd, System Journal, operating system log) for the subsequent events following the failure.

4) Finally, we analyze the timings in the chain of events and draw conclusions.

---

**B. Test plan**

Our objective is to define a comprehensive test plan that considers the key failures in the environment that can affect the availability of the applications. Our plan is based on four pillars. We test the failures affecting the pods, the host, the network and finally those affecting the Kubernetes master as shown in Figure 3.

---

**Fig. 1. A simplified UML class diagram of the Kubernetes resources.**

**Fig. 2. Testing methodology**

**Fig. 3. Test plan overview**
C. Testing environment

Our testing environment consists of three master nodes (for HA) connected to a cluster of three replicas of the etcd key/value store and three minions. The nodes are connected via two networks: a management network and a data network. We deployed all of the Kubernetes components inside pods (except for the Kubelet, since it is needed to start the pod). The rational behind this is to leverage the Kubelet monitoring and recovery (locally) capabilities for those components. We used Kubernetes version 1.1.7 in our tests. The Kubelet, the Docker engine, and the overlay daemon are configured as Systemd services with auto-restart. Figure 4 shows an overview of our test environment deployment. We configured the monitoring intervals for the controller-manager and the Kubelet syncing with the master to their minimum values (1second) in order to test the maximum responsiveness we can get out of the system.

Currently, the minions do not support the notion of load balancing the interactions among multi-masters. Hence in our deployment we hide the masters behind a virtual IP, where the minions (and the clients) have the illusion of communicating with a single server.

IV. BENCHMARKING RESULTS

This section illustrates the various experimental results we collected in our testing. The presented results are the average of 25 runs for each experiment. It should be noted that there is more than one way to deploy Kubernetes. Some deployments can favor performance, e.g. by collocating the master components with etcd on the same host and avoiding the network delay. Others can favor manageability and reliability, by containerizing the components and placing the etcd on dedicated hosts. Therefore, the experimental results might vary according to different deployments, and configuration settings. The objective here is to give an idea of the expected HA system behavior.

A. Pod failure testing

The concept of pod is a logical entity within Kubernetes, in order to simulate its failure we abruptly kill the containers within the pod. We distinguish between the following events during the recovery:

- *Failure detected by the Kubelet*: this is the duration it takes the Kubelet to realize that one of the containers under its management is faulty.
- *Container created*: this is the duration it takes Docker to create the container after its failure.
- *Container active*: this is the duration it takes for the container to become active. (Note that our test containers are based on Ubuntu images, which includes our scripts installed inside to time stamp their activation).
- *Failure reported to master*: this is the duration it takes for the master to realize that a failure has occurred.
- *Recovery reported to master*: this is the duration it takes for the master to realize that all the containers of the pod have successfully recovered.

Table I shows the testing results of the regular container failure.

<table>
<thead>
<tr>
<th>Action</th>
<th>Duration</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure detected by the Kubelet</td>
<td>0.39 s</td>
<td>0.12 s</td>
</tr>
<tr>
<td>Container created</td>
<td>1.37 s</td>
<td>0.51 s</td>
</tr>
<tr>
<td>Container active</td>
<td>2.10 s</td>
<td>0.42 s</td>
</tr>
<tr>
<td>Pod Failure reported to the master</td>
<td>1.78 s</td>
<td>0.80 s</td>
</tr>
<tr>
<td>Pod Recovery reported to the master</td>
<td>2.24 s</td>
<td>0.54 s</td>
</tr>
</tbody>
</table>

For the infrastructure container failure, we see a slight increase in the recovery time. This is due to the fact that when the infrastructure container fails, all the containers in the pod are restarted after its recovery.

<table>
<thead>
<tr>
<th>Action</th>
<th>Duration</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure detected by the Kubelet</td>
<td>0.57 s</td>
<td>0.23 s</td>
</tr>
<tr>
<td>Container created</td>
<td>2.17 s</td>
<td>0.41 s</td>
</tr>
<tr>
<td>Container active</td>
<td>2.54 s</td>
<td>0.56 s</td>
</tr>
<tr>
<td>Failure reported to the master</td>
<td>2.25 s</td>
<td>0.61 s</td>
</tr>
<tr>
<td>Recovery reported to the master</td>
<td>3.04 s</td>
<td>0.32 s</td>
</tr>
</tbody>
</table>
It should be noted here that with the container failure, the recovery can sometimes be faster than the time it takes the Kubernetes master to be notified of the failure, hence relying solely on the Kubernetes master to get the recovery duration would result in longer perceived outages than the actual outage.

The Kubelet failure would cause a disruption in the communication between the master and the minion, from this perspective Kubernetes will consider this failure as a node failure (discussed in the next subsection) and failover all the pods to another node. However, we noticed that the containers of those pods still execute on the minion with the faulty Kubelet. Therefore, there is a need for a fencing mechanism that allows the forceful termination of such containers before the failover, especially if certain processes in those containers are locking sensitive resource like database entries for instance.

The container engine (i.e. Docker) failure will cause the immediate termination of all the containers on the minion. However, we noticed that the Kubernetes master takes a longer duration to realize the failure. Although the Kubelet becomes aware of the Docker failure almost instantly, it would take the master a significantly longer duration to declare the node in a “not ready” state and trigger the failover of the pods.

B. Host failure

A Kubernetes node can be deployed on a physical server or in a virtual machine (VM). We simulate the host failure by abruptly powering it off, or by causing the operating system to immediately go into a halted state. Almost all types of the host failure incur similar outage durations to the ones shown in Table III.

Table III. Host Failure Recovery Times

<table>
<thead>
<tr>
<th>Action</th>
<th>Duration</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure detected by the master</td>
<td>4.65 s</td>
<td>2.7 s</td>
</tr>
<tr>
<td>Container created</td>
<td>5.92 s</td>
<td>0.31 s</td>
</tr>
<tr>
<td>Container active</td>
<td>6.70 s</td>
<td>0.42 s</td>
</tr>
<tr>
<td>Recovery reported to the master</td>
<td>9.1 s</td>
<td>0.76 s</td>
</tr>
<tr>
<td>System back to normal (all pods in running state, none in terminating)</td>
<td>34.24 s</td>
<td>4.24 s</td>
</tr>
</tbody>
</table>

C. Network Failure

The network failure can be (1) at the Kubernetes level, e.g. when the Kubernetes proxy fails on the node, or the overlay network daemon fails, or (2) at the system level e.g. the network card, or the switch/router is faulty. We deploy the Kube-proxy within a pod that is constantly monitored by the Kubelet, hence the failure of the containers is swiftly handled as if it was a container failure. As for the overlay daemon failure, (in our deployment) it has a co-dependency on the Docker engine, since Docker only assigns IP addresses in accordance to the ones specified by the overlay. Therefore this failure entails the shutdown of Docker, and hence it is handled as a Docker failure.

On the other hand, when a network interface fails it is handled as a host failure; the same applies to the failure of the switch/router connecting the Kubernetes nodes. This will put the master in a panic state, where each master node thinks the entire cluster is down. Therefore it is highly recommended to have a redundant physical network setup in order to avoid this scenario.

D. Kubernetes Master Components Failure

In the Kubernetes master failure, we focus on the failure of the components on the master side that are essential for the recovery of the pods. We considered the key/value store as part of the master, since Kubernetes stores all its configuration and runtime states in that store.

Not all the components of the Kubernetes master have the same impact on availability in case of their failure. In fact, although a failure at the master level does not directly impact the running pods, it has two main impacts: (1) it disrupts the service provisioning where the client requests (create, delete, scale, etc.) may not be serviced, and (2) it will disable the failover recovery in case of a subsequent failure at the minion level. Local failures (requiring a restart not a failover) can still be handled by the Kubelet.

Table IV summarizes the impact of the failure of each of the Kubernetes master components. It clearly shows that the API-server is the most critical component. As for the pods with predefined nodes (not requiring the scheduler) and no replication controller (not requiring the controller-manager) they can still be created while the scheduler and the controller-manager are faulty.

Table IV. Impact of the Master Components Failure

<table>
<thead>
<tr>
<th>Kubernetes Component</th>
<th>Failure detected</th>
<th>Fail-over possible</th>
<th>Pods creation with predefined nodes</th>
<th>Scaling up possible</th>
<th>Scaling down possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller-manager failure</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>API-server failure</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Scheduler failure</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ETCD failure</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

E. Resource consumption

As discussed earlier, the monitoring intervals in Kubernetes are configurable. Setting those intervals to their minimum values is not recommended at the normal state of execution as it increases the utilization of computing and memory resources. In this subsection we examine the impact of relaxing those intervals from one second to five second on the resource utilization.
We also examine the impact of having the maximum recommended pods per node (forty) versus an average of ten pods per node. Figure 5 illustrates the results of this experiment, where the master node has 1.5 GB of memory and one CPU of 2.2 Ghz. We can clearly see that there is no substantial increase in the resource utilization, even with smaller monitoring intervals and a larger number of pods.

**Controller-Manager Resource Utilization**

Fig. 5. Resource consumption of the controller manager

V. RELATED WORK

The related work devoted to the enhancement of the HA features of Linux containers clustering solutions is mainly defined by the related technologies rather than the academic research. This is normal since at the time of the writing, container clustering solutions are mainly driven by development and have not reached a stable level of maturity yet. Therefore, we first present the related technologies and then briefly survey the literature.

Docker Swarm [9] is a recently launched project as the native clustering tool for Docker. The basic architecture of Swarm is fairly similar to that of Kubernetes: a cluster has a Swarm manager and each host runs a Swarm agent and one host runs a Swarm manager. The manager is in charge on the orchestration and scheduling of containers on the hosts. The Swarm scheduler can reschedule faulty containers on other nodes. However, this HA feature is still at the experimental level [10]. Swarm can also be deployed in an HA mode at the master level using tools implementing a consensus protocol such as Raft (etcd, Consul [11]) or ZAB (Zookeeper Atomic Broadcast [12]) to handle fail-over to a standby manager. At the time of the writing of this paper, Kubernetes (with its controller-manager) is a more mature product from a HA perspective.

Mesos [13] is a highly scalable cluster manager that can manage multiple frameworks sharing the same cluster resources. While it is not designed as a container manager, it works at the job level, and a job can consist of launching a container. Mesos uses the Marathon framework (among others) [14] as the scheduler. Multiple Mesos masters can be used. The scheduler uses ZooKeeper to locate the current Mesos master to which it can submit the tasks. Marathon is capable of starting, monitoring and scaling applications. At the architectural level Mesos follows a master/slave deployment where a slave agent is deployed on each of the cluster node to run the Mesos jobs.

Fleet [15] is a cluster management tool that builds on top of systemd, giving the illusion of a distributed system across a cluster of machines. Fleet takes systemd unit files as input, and then schedules them on the machines in the cluster. The unit file will typically specify a container to run. At the architectural level, Fleet has one active engine per cluster and an agent running on each node. The agent takes care of starting units and reporting state. In case of failure the engine will reschedule those units on other machines. Fleet operates at a lower level than Kubernetes, since it tightly coupled with systemd.

In [16], the authors from Google present a case study of the networking in container-based clusters. In particular, they explore the networking setups of Kubernetes. The authors do not discuss in the paper this container clustering solution from performance or high availability perspective. Borg is the Google's cluster manager that has the capability to support hundreds of thousands of jobs. In [17], the authors present the system architecture and the most important features of Borg. In particular, they describe the support for applications high availability, which is claimed to reach 99.999% in practice. In [18], the authors discuss the performance gains obtained using Docker containers to run parallel tasks in comparison with the KVM hypervisor in a cluster computing environment.

VI. CONCLUSION

HA is characteristically achieved when the system availability reaches the 5 nines (i.e. the system is available 99.999% of the time). This leaves an outage margin of roughly 315 seconds per year. As an availability manager, Kubernetes is capable of recovering the containers within seconds. Moreover, Kubernetes can be highly available itself by replicating its master. Kubernetes is still under heavy development, and we still see room for improvement within Kubernetes, such as better fencing (for failure isolation), and more built-in HA features such as implementing a dynamic approach for the minions to identify their master (similarly to Mesos). Other related emerging container clustering solutions support the HA mechanisms that are crucial for service recovery in case of failure. Therefore we believe that the reliability and availability of containerized services will continue to improve as container clustering and management solutions continue to evolve.

REFERENCES


---

1 Kubernetes currently recommends no more than 40 pods per node, however this is not a hard constraint, as we have successfully deployed hundreds of pods per node.

Red Hat OpenShift platform as a service, available online: https://www.openshift.com/ (accessed on February 25, 2016).

etcd distributed, consistent key-value store, available online: https://github.com/coreos/etcd / (accessed on February 25, 2016).


Docker Swarm, clustering for docker, available online: https://docs.docker.com/swarm/ (accessed on February 25, 2016).


Consul, a tool for discovering and configuring services, available online: https://www.consul.io/ (accessed on February 25, 2016).


Victir Marmol, Rohit jnagal and Tim Hockin, “Networking in Containers and Container Clusters”, Proceedings of netdev 0.1, 2015, Ottawa, Canada.

