2018 IEEE 11th International Conference on Cloud Computing

Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned

Leila Abdollahi Vayghan
Engineering and Computer Science
Concordia University
Montreal, Canada
l_abdoll@encs.concordia.ca

Mohamed Aymen Saied
Engineering and Computer Science
Concordia University
Montreal, Canada
m_saied@encs.concordia.ca

Maria Toeroe
Ericsson Inc.
Montreal, Canada
maria.toeroe@ericsson.com

Ferhat Khendek
Engineering and Computer Science
Concordia University
Montreal, Canada
ferhat.khendek@concordia.ca

***Abstract*— Microservices represent a new architectural style where small and loosely coupled modules can be developed and deployed independently to compose an application. This architectural style brings various benefits such as maintainability and flexibility in scaling and aims at decreasing downtime in case of failure or upgrade. One of the enablers is Kubernetes, an open source platform that provides mechanisms for deploying, maintaining, and scaling containerized applications across a cluster of hosts. Moreover, Kubernetes enables healing through failure recovery actions to improve the availability of applications. As our ultimate goal is to devise architectures to enable high availability (HA) with Kubernetes for microservice based applications, in this paper we examine the availability achievable through Kubernetes under its default configuration. We have conducted a set of experiments which show that the service outage can be significantly higher than expected.**

高可用

outage
断停期

Keywords*— *Microservices; Containers; Orchestration; Docker; Kubernetes; Failure; Availability

I. Introduction

A microservice is built around a separate business functionality, it runs in its own process and communicates through lightweight mechanisms, often using APIs [1, 2, 3]. Microservices, composing an application, can be written in different programming languages and use different storage technologies. Microservices can address the drawbacks of the monolithic approach, where the application is a single deployable unit suffering from "dependency hell" and creating barriers for scalability and high availability [3, 4]. Microservices increase velocity and quality. By being small, they can restart faster after upgrade or for failure recovery. Microservices are loosely coupled and failure of one microservice will not affect other microservices of the application. These factors impact the availability of applications as they decrease downtime [3]. Moreover, the fine-grained architecture makes scaling flexible as each service can evolve at its own workload pace. To leverage all these benefits, one needs to use technologies aligned with the characteristics of this architectural style.

整体的

速率

Containers are lightweight and start up faster than virtual machines (VMs) [5]. Thus, the containerization of microservices can help to speed up the restart after upgrade or for failure recovery. In our experiments, we use Docker [6], the leading container platform. There is also a need for an orchestration platform to manage the deployment and operations of containers. Kubernetes [7] is an open source platform that manages Docker containers in a cluster. Along with the automated deployment and scaling of containers, Kubernetes provides healing by automatically restarting failed containers and rescheduling them when their hosts die [7]. This capability improves the application's availability.

The architectural style of microservices is being adopted by practitioners and investigated from different perspectives by researchers in academia as well. In [5], the authors assessed the effectiveness of the HA mechanisms offered by Kubernetes while setting its monitoring intervals to their minimum values. The authors concluded that Kubernetes still needs improvement to provide HA. Such a configuration is not recommended and may lead to false node failure detection. In our study, we investigate the HA effectiveness of Kubernetes under its default configuration. availability?

In this paper, we present an architecture for deploying microservice based applications with Kubernetes in a private cloud. Our ultimate goal is to devise architectures to enable high availability with Kubernetes for microservice based applications. Therefore, we perform a quantitative evaluation of the availability achievable through Kubernetes under its default configuration. For this purpose, we have conducted experiments to measure availability metrics in different failure scenarios. We briefly discuss and analyze our findings.

The rest of the paper is organized as follows. Section II introduces the architecture for deploying containerized applications with Kubernetes in a private cloud. In Section III we present our availability experiments. The analysis of the results is discussed in Section IV. We conclude in Section V.

II. Deploying Containerized Applications in a Kubernetes Cluster Running in a Private Cloud

Kubernetes can run in a cluster in a public or a private cloud. However, it is important to understand that deploying applications to a Kubernetes cluster running in a private cloud requires more effort than it does for a public cloud. The main difference is in the way of exposing the application externally. Below, we will discuss exposing applications deployed in a Kubernetes cluster running in a private cloud. Kubernetes runs on all VMs and creates a unified view of the cluster. For simplicity, the application here is composed of only one microservice.

Using Kubernetes' ingress resource is a structured way to expose services. Fig. 1 shows a generic architecture with ingress exposing the service in a cluster running in a private cloud to the outside world. In this case, a service of type Cluster IP is created to redirect requests to the pods and will be used as the backend of the ingress resource. Also, an ingress controller [9] is needed in the cluster in order to redirect the incoming requests to the ingress resource, which later will be redirected to the appropriate backend service. The ingress controller is deployed as one pod using a deployment controller. We define a rule for the deployment controller to always schedule the ingress controller on a specific node. In Kubernetes, it is possible to connect a pod directly to a port on the node hosting the pod. Since we have defined a rule to keep the ingress controller pod on the same node, it is safe to connect it to a port on its host so it can receive requests from outside of the cluster on the public IP of that node. Adapting the ingress controller to a Kubernetes cluster running in the private cloud is not an easy task and there is no sufficient documentation on how to use ingress controllers in these types of clusters. Although the role of each Kubernetes architectural component is described in [8], understanding how to put them together is not intuitive. It requires lots of trials and errors to figure out the ways these components work together in practice. The aforementioned architecture is a result of our understanding of these components' roles.

修改

直觉的

III. Evaluation of the Availability of Microservice Based Applications Deployed with Kubernetes

断供期

Availability is a nonfunctional requirement which is measured as the outage time over a given period [10]. High availability is achieved when the system is available at least 99.999% of the time, i.e. no more than around 5 minutes outage per year [11]. Some characteristics of microservices and containers such as being small and lightweight naturally contribute to improving availability when supported by Kubernetes' healing capability [12]. In this section, we describe the experiments we conducted to evaluate from an availability perspective the deployment of a microservice based application in a Kubernetes cluster running in a private cloud (as shown in Fig. 1).

Kubernetes reacts to a pod failure by automatically starting a new pod and therefore it is expected to improve the availability of the service provided by the pod. The common practice to evaluate Kubernetes' reaction to failure is to simulate failures through administrative operations (e.g. delete the pod or the node) using the Kubernetes command line interface (CLI) and then observe how fast a new pod replaces the failed one [13]. Due to the use of Kubernetes' administrative operations, such a "failure" is not a spontaneous event that Kubernetes needs to detect and react to. Instead, the operation is executed by Kubernetes in due order often in a graceful manner. Therefore, these operations cannot reflect common execution failure scenarios, which are anything but graceful and happen spontaneously as a result of external failure events (e.g. process or physical node crash). Drawing conclusions based on such administrative operations would not be accurate. Hence, it is important to identify and simulate (external) execution failure scenarios properly and measure the availability in these cases before making conclusions.

随机的
自发的

以前实验的问题：以前通过管理上的操作来模拟一些错误，以测试k8s之反应
但这些错误不是原生的，不是k8s需要检测和处理的错误

A. Availability metrics

The metrics we use to evaluate Kubernetes from availability perspective are defined hereafter.

测试指标

1) *Reaction Time:* The time between the failure event we introduce and the first reaction of Kubernetes that reflects the failure event was detected.

2) *Repair Time:* The time between the first reaction of Kubernetes and the repair of the failed pod.

3) *Recovery Time:* The time between the first reaction of Kubernetes and when the service is available again.

4) *Outage Time:* The duration in which the service was not available. It represents the sum of the reaction time and the recovery time.

B. The Experiments

We used the architecture in Fig. 1. Our cluster is composed of three VMs running on an OpenStack cloud. Ubuntu 16.04 is the OS on all VMs and Docker 17.09 is running as the container engine. Kubernetes 1.8.2 is running on all nodes. Network Time Protocol (NTP) [14] is used for time synchronization between nodes in the cluster. The microservice we use in these experiments is VLC video streaming. The pod template provided to the deployment controller contains the container image of the streaming server, which once deployed will stream from a file. In our experiments, the desired number of pods that the deployment controller needs to maintain is one. This can help to understand the achievable availability through the Kubernetes' repair action by itself. Our video streaming microservice is stateless and in case of failure, the video will restart from the beginning of the file.

Shit黄色那段没看懂

We identified two sets of failure scenarios. In the first set, an application failure is due to a pod failure whereas in the second set it is due to a node failure. In each set, we distinguish between two failure scenarios. Scenario I designates a failure simulated

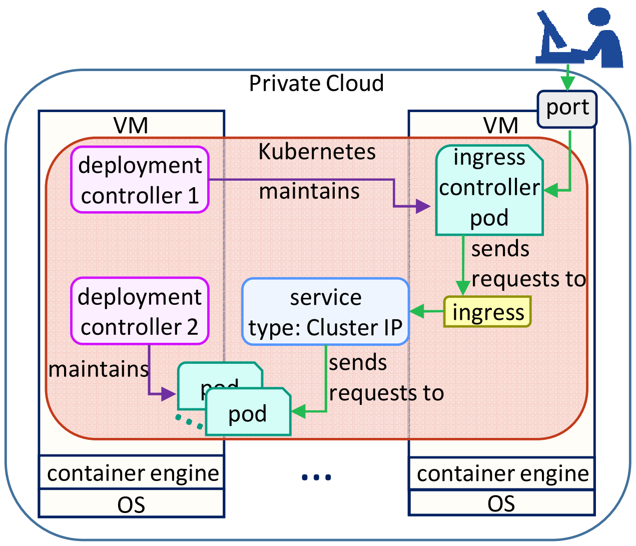

Fig. 1. Private cloud - exposing services via ingress.

by an administrative operation internal to Kubernetes while Scenario II is simulated by a trigger external to Kubernetes. Below, we explain each failure scenario in details. The results of our experiments for these failure scenarios are presented in Table I and Table II. Each scenario in the experiments has been repeated 10 times. The results presented in the tables are the averages of the 10 measurements. All measurements throughout the paper are reported in seconds.

1) *Pod Failure Scenarios*

Scenario I - Application failure due to administrative pod termination: The common way of showing Kubernetes' reaction to pod failure is to delete the pod using the administrative commands in the CLI [13]. In this scenario, the pod is ordered to terminate by a command, it will consequently be removed from the endpoints list of the service and this is when we consider that the pod has failed. By default, pods have 30 seconds of graceful termination period. During this time, the pod will not receive new requests, but will keep serving the requests previously assigned to it. This gives ample time to 充足的 Kubernetes to schedule a new pod and deal with incoming requests. Since it is the responsibility of the deployment controller to always maintain one replica of this pod, it will bring up a new one and this event marks the reaction time. The repair time is when the new pod is started. Although at this time, the pod has started and is streaming the video, it will not be available to users unless it is added to the endpoints list of the service. Therefore, we consider the streaming service as recovered when the new pod is added to the endpoints list of the service. The measurements for this scenario are presented in Table I.

Scenario II - Application failure due to pod process failure: When a pod consisting of some containers (application containers) is deployed, along with those containers included in its specification, one more container is created which is the pod's container itself. Since pods themselves are processes running in the OS, they may crash resulting in the failure of the entire pod. Hence, we consider this case as the externally triggered pod failure scenario. To simulate a pod process failure, we kill the process of the pod container from the OS. The crash of the pod process is detected by the Kubelet and this time is marked as the reaction time. Consequently, the Kube-proxy removes the pod from the endpoints list of the service. When the pod process crashes, a graceful termination signal is sent to the application container and Docker will wait 30 seconds before forcefully killing it. As it was observed in the experiments, although the pod has crashed and the streaming service is no longer available, Kubernetes waits for the application container to terminate before starting a new pod and since the application container is given a graceful termination signal, this may take up to 30 seconds. After it is guaranteed that the application container is terminated, the deployment controller will restart the pod. We mark this time as the repair time. The recovery time is when the pod is added to the endpoints list of the service again. The measurements for this scenario are presented in Table I.

TABLE I. APPLICATION FAILURE DUE TO POD FAILURE

Failure scenario (unit: seconds)	**Reaction time**	**Repair time**	**Recovery time**	**Outage time**
Scenario I	0.041	0.982	1.547	1.588
Scenario II	0.496	32.570	31.523	32.019

2) *Node Failure Scenarios*

Scenario I - Application failure due to an administrative deletion of the node: A method used to simulate node failure is the administrative deletion of a node without draining it [13]. Although this is not what we would consider similar to a spontaneous node failure since it is done via Kubernetes CLI, we examine it to understand the differences in Kubernetes' reactions. In this scenario, a node hosting a pod is deleted using a Kubernetes' CLI command. As a result, the cleanup of all containers and processes related to Kubernetes on this node is initiated. Any pod running on the node that is to be deleted enters a state where it does not receive new requests. Hence, this is what we consider the moment of pod failure. However, the behavior in this scenario is different from that of pod failure Scenario I. Here, the pod will serve the previously assigned requests for only around one second (not the default 30 seconds of graceful termination period). Shortly after, the pod is completely deleted and this time marks the reaction time. When the pod is completely deleted, the deployment controller will attempt to add a new pod on another node. Repair time is when the new pod is started. Recovery time is marked later when the new pod is added to the endpoints list of the service. The measurements for this scenario are presented in Table II.

Scenario II - Application failure due to externally triggered node failure: As mentioned before, one of the responsibilities of the Kubelet is to report the status of the node to the master. In the default configuration of Kubernetes, the Kubelet posts the node's status to the master every 10 seconds. It is allowed for a node to be unresponsive for 4 consecutive status updates before it is marked as failed. After marking the node as failed, the master waits another 4 minutes and 20 seconds before considering the pods running on the failed node as terminated and only then those pods are rescheduled on another node. This means that with the default configuration of Kubernetes, it takes around 5 minutes to recover from a node failure. To simulate this scenario, the VM hosting the pod is shutdown by the Linux's reboot command. The master considers the node as not ready after the fourth missed status update and this time is marked as the reaction time. After around 5 minutes, once a new pod is brought up by the deployment controller, we consider the pod as repaired. When the new pod's IP is added to the endpoints list, the service is recovered. The measurements for this scenario are presented in Table II.

IV. ANALYSIS

A. *Analysis of the Pod Failure Scenarios*

In the pod failure Scenario I, the reaction time is 0.041 seconds which is significantly better than the 0.496 seconds of the pod process failure (Scenario II). The reason is that in the former, the termination is triggered from inside of Kubernetes, which then reacts according to the termination procedure, while in the latter it is up to the Kubelet's health check to detect that the pod is no longer present and this depends on how close to the next health check the failure happens.

TABLE II. APPLICATION FAILURE DUE TO NODE FAILURE

Failure scenario (unit: seconds)	**Reaction time**	**Repair time**	**Recovery time**	**Outage time**
Scenario I	0.031	1.009	1.500	1.531
Scenario II	38.187	262.542	262.665	300.852

使成为孤儿；孤儿的

An important observation on the experiments is shown in Fig. 2. Although the pod process is failed forcefully in case of Scenario II (Fig. 2 (b)), the orphaned application container of the pod receives a graceful termination signal. Thus, the pod process failure is detected by the Kubelet, which waits for Docker and will not start the repair process before it makes sure that the application container of the pod is terminated as well. This means graceful termination of the (application container whose duration depends on Docker's configuration), impacts and delays the service recovery time. However, this may also allow for fault propagation when the pod process fails due to real fault or bug. Fault isolation principles would require immediate forceful cleanup of the application containers once their pod's process failure is detected. This grace period is the reason why the repair time for Scenario II is 32.57 seconds. This is significantly longer than that of the Scenario I which is 0.982 seconds. In the latter (Fig. 2. (a)), Kubernetes performs the graceful termination and repair procedures in parallel. The ordering is guaranteed for certain actions of these procedures. For example, the removal of the terminating pod from the endpoints list precedes the start of the repair procedure, and the completion of pod termination follows the addition of the new pod. This parallelization is possible due to the assumption that there is no fault in the system. This emphasizes the point made earlier about the correct simulation of failures with respect to availability metrics. As it is observed, Scenario I reports an outage time of 1.588 seconds while for Scenario II it is 32.019 seconds.

繁殖

B. Analysis of the Node Failure Scenarios

For the node failure scenarios, we observed similar differences in all measured availability metrics. For Scenario I, since the failure is triggered from inside of Kubernetes, the reaction time is 0.031 and it is significantly faster than the reaction time of Scenario II which is 38.187 seconds. As explained before, it depends on the period of Kubelet's status update (default 10 seconds) and the allowed number of misses (default 4 seconds).

Another important observation for Scenario I is that although the failure is triggered from inside of Kubernetes, the new pod is started after the old one is terminated. It was expected to behave similarly to the administrative pod termination (Scenario I) where the new pod is started before the old one is terminated.

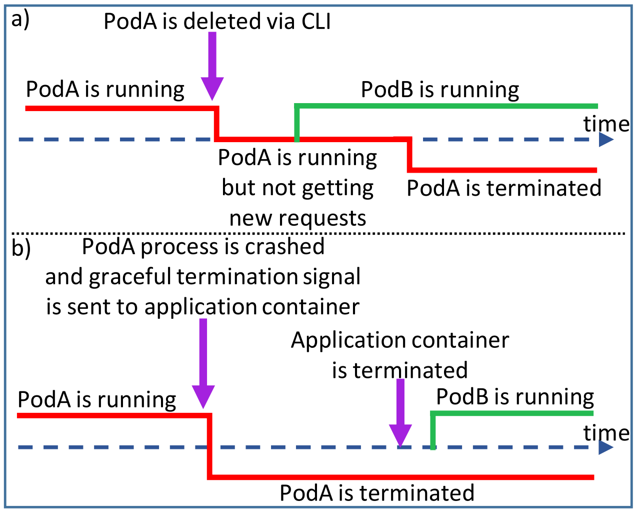
a)
PodA is deleted via CLI
PodA is running
PodB is running
time
PodA is running but not getting new requests
PodA is terminated
b)
PodA process is crashed and graceful termination signal is sent to application container
Application container is terminated
PodA is running
PodB is running
time
PodA is terminated

Fig. 2. Analysis of pod failure scenarios. (a) Scenario I. (b) Scenario II.

V. Conclusion

Kubernetes enables healing through its failure recovery actions and they are often evaluated through internal operations. For these types of operations, Kubernetes reacts reasonably well in comparison with its reaction to failures resulting from external triggers. According to our experiments, in the latter, the downtime is significantly higher. It is important to note that the default configuration of Kubernetes results in a significant service outage in case of externally triggered node failure. As our measurements show for these types of failures, the outage time is about 5 minutes, which is equivalent to the amount of downtime allowed in a one-year period for a highly available system. These differences indicate that high availability requirements are not satisfied automatically by deploying an application or a microservice with Kubernetes.

Although the default configuration can be changed, figuring out how to reconfigure Kubernetes' reaction to node failures while avoiding network overhead and false positive reports can be complicated and requires a great effort.

Acknowledgment

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) and Ericsson.

References

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[2] "Microservices," *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Acc.: 06-Feb-18].

[3] N. Dragoni *et al.*, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, Springer, Cham, 2017, pp. 195–216.

[4] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology," in *SoutheastCon 2016*, 2016, pp. 1–5.

[5] A. Kanso, H. Huang, I. T. J. Watson, and A. Gherbi, "Can Linux Containers Clustering Solutions offer High Availability?," p. 6.

[6] "Docker - Build, Ship, and Run Any App, Anywhere." [Online]. Available: https://www.docker.com/. [Acc.: 02-Jan-18].

[7] "Kubernetes," *Kubernetes*. [Online]. Available: https://kubernetes.io/. [Acc.: 24-Jan-18].

[8] "Kubernetes Documentation," *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/home/. [Acc.: 23-Jan-18].

[9] *ingress-nginx: Ingress controller for nginx*. Kubernetes, 2017.

[10] M. Toeroe and F. Tam, *Service Availability: Principles and Practice*. John Wiley & Sons, 2012.

[11] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *J. Netw. Comput. Appl.*, vol. 60, pp. 54–67, Jan. 2016.

[12] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How To Make Your Application Scale," *ArXiv170207149 Cs*, Feb. 2017.

[13] "Run a Replicated Stateful Application," *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/. [Acc.: 25-Jan-18].

[14] "ntp.org: Home of the Network Time Protocol." [Online]. Available: http://www.ntp.org/. [Acc.: 03-Jan-18].