

Your Very Nice Project Title

Gangmuk Lim

University of Illinois at Urbana-Champaign
{gangmuk2}@illinois.edu

Abstract

Many of modern large systems are composed of a group of individual components. Each component is doing their own dedicated duty based on what it observes from the environment. There are different programming models but we will focus on one where each component keeps checking on the status of the system and tries to make the system converge on a certain desirable status. There is no a single global controller who has the perfect knowledge of the entire system and guarantees that it will not fall into undesirable status. Not just because they are such a big and complex system but there are some particular reasons facilitating the system to be unintentionally faulty. First, multiple components often interact each other and control the same part of the system. Inherently it is possible that they behave contradictorily each other. Second, each component is often developed by multiple people, multiple teams and even multiple organizations. Each team cannot understand how other parts of the system functions in detail. It is almost infeasible and not practical for each component to perfectly take into consideration other parts of the system. This paper did case study to understand the failures involving multiple components. We pick Kubernetes container orchestration system as a representative example to dive into more specific failure cases. We analyzed 10 different failure cases which were reported in Kubernetes github issues, Kubecon (Kubernetes conference), and blog posts.

1 Introduction

The introduction of your awesome 523 project.

2 Background

Kubernetes

Controller Kubelet / Deployment / Scheduler / HPA / Descheduler / api-server / etcd

Reconciliation Monitoring system Distributed tracing e.g., Jaeger, Prometheus, ...

3 Case study

Patterns of multi-controller failure cases

How are these failures created? Large systems are developed by people in different teams and different organizations. One team or a certain group of people in case of open source is responsible for developing and maintaining one controller. It is impossible for them to know how all other controllers

work in detail. It is not feasible to make a controller not incurring any conflicted cases with all other controller for all possible cases. That is not even desirable development direction because then the process will take too long. How Kubernetes function is also contributing to this type of failure. What each controller does is periodically monitoring if the associated part in the cluster is currently in desirable status or not. If it is not, it will try to bring it back to the desirable status, running reconciliation process. For example, if the current number of pods is less than what it should be based on HPA configuration. HPA will detect it and increase the number of pods. The important part is a controller does not care what other controllers are doing and it just see the current status and run its own logic to put the cluster back to the right place.

Why is it hard to prevent these failures? There are different categories of failures existing in Kubernetes. Inherently, it is ambiguous to classify such failure cases as bug. Each controller is working as it is supposed to do and clearly there is no bug in its logic if you look them individually. Some of answers to reported github issues was that developers in Kubernetes community maintaining the controller are aware of or admit the issue but they ended up not fixing it while saying it is their design choice or there is no clear way to patch it. It is because often the clear solution does not exist not because Kubernetes community constantly procrastinate or neglect them. Most of problems we are presenting in this paper could be prevented if you know it will happen before you apply them to your cluster. For the example in D1 failure case, if you had known that Deployment configuration and Scheduler policy are contradictory each other, you would have not configured in such self-destroying way. However, there are several reasons that it is still not trivial to avoid this type of failures. The first reason is scalability. The number of pods could be easily over a thousand and the number of services deployed in the cluster could be easily over a hundred. To configure this scale of Kubernetes cluster having different applications running, you may need to manage a large number of configuration yaml files for each deployment, nodes, hpa, schedulers, etc. In this scale and complexity, it is challenging to make all controllers always in a coordinated manner in the cluster-wise level. The second reason is that in a large scale cluster some failures are not noticeable and slowly gnawing the cluster's resources. For example, in S1 failure case, node utilization on average could stay within

some range in the half way of scheduler config and descheduler config. It is possible that the cluster resource utilization looks stable even though what is happening behind is descheduler keeping evicting pods from high utilization node and scheduler placing pods in low utilization node making the utilization high again. The third reason is that semantically contradictory configurations do not mean it will always lead to failures. Many of these problems are triggered in a specific number of nodes and pods.

Detailed examples

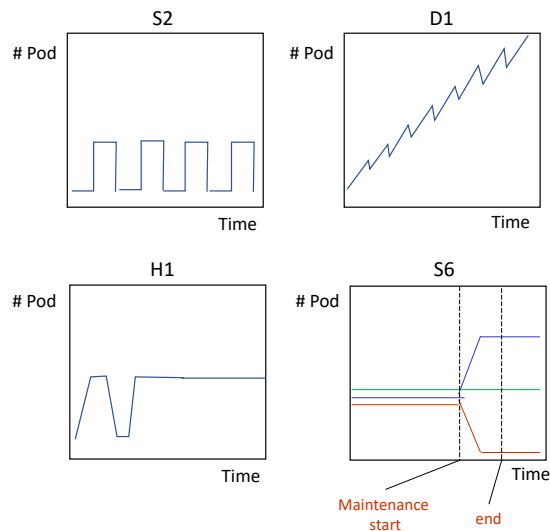


Figure 1. Number of pods over time for four failure cases.

Observation 1

4 Related Work

A single controller failure study

K8s operator failure

Testing

Chaos engineering

Model checking

Formal verification

Statistical model The related work of your project [1].

5 Discussion

discussion

modeling / verification /

6 Conclusion

This project is awesome.

7 Metadata

The presentation of the project can be found at:

<https://zoom/cloud/link/>

The code/data of the project can be found at:

<https://github.com/you/repo>

References

- [1] DIJKSTRA, E. W. The Structure of the "THE" Multiprogramming System. In *Proceedings of the 1st ACM Symposium on Operating System Principles (SOSP'67)* (Oct. 1967).

Case	Categories	Controllers	Properties	Behaviors
D1(R)	Conflicted config	Deployment + Kubelet	Liveness	Scheduling and evicting pods infinitely
H1(R)	Lack of context	HPA + App CPU changes	Safety	HPA is agnostic to app
H2(R)	Conflicted config	HPA + Deployment	Safety	Sub-optimal scaling behavior
H3(R)	Imperfect knowledge	HPA + Node reachability	Safety	Semantically wrong avg CPU util (reachability vs healthiness)
S1	Conflicted config	Scheduler + Descheduler	Liveness	High utilization(scheduler) <-> Low utilization(descheduler)
S2(R)	Conflicted config	Scheduler + Descheduler	Liveness	Deployment preference <-> Violation in maxSkew
S3(R)	Conflicted config	Scheduler	Liveness	Two pod spread constraints are conflicted each other
S5	Conflicted config	Scheduler	Safety	Pods are scheduled to one node, because of lopsided preference
S6(R)	Lack of feature	Scheduler	Safety	Scheduler is not able to adjust skewed placement
S7(R)	Lack of context	Scheduler + Kubelet	Liveness	Scheduler includes NotReady node for maxSkew calculation

Table 1