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. Seocnd, 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 awsome 523 project.

2 Background

Kubernetes

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

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

3 Case study

Patterns of multi-controller failure cases

How are these failures created at the first place? 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 responsble for developing and maintaining a certain part out of the entire system. It is impossible for them to know how all other parts of the system function in detail. It is not feasible to write a controller not incuring any conflicted cases with all other controller for all possible cases. Even if it is technically possible, it is not even desirable because then the development process will take exhastively long time. The programming model of the large system including Kubernetes is another contributer to this type of failure. We will use Kubernetes to describe more detailed example. What each controller does in Kubernetes 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 built-in logic of the controller. For example, Horizontal Pod Autoscaler(HPA) scales up the number of pods if CPU utilization becomes higher than 50% (it is configurable). This specific model is called reconciliation. During reconciliation The important part is a controller does not take into account what other controllers are doing and how they can react to its action. It simply sees the current status of the cluster and run its own logic to put the cluster back to the right place. Even within one controller, there could be multiple configurations which are functionally not exclusive each other. For exapmle, Deployment could be configured with multiple *Pod Topology Spread Constraint* policies and currently Kubernetes does not have safety net guaranteeing that the configurations are not contradictorial or any kind of alerting system warning it is contradictorial. It is completely at your own risk.

Why is it hard to prevent these failures? If it is evident that the system could fall into already known precarious status when the cluster is deployed with self-destroying configuration, why were they not fixed already? It is not because it is such a huge system and the community is large enough or active to resolve these problems. It is because they are not trivial problems. Note that each part of the system is not doing anything wrong if you look at them individually and there is no bug in the code. It is inherently 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 against reported github issues was that developers in Kubernetes community maintaining the controller are aware of or admit that is the issue but they ended up not fixing or not being able to fix it while saying it is their design choice or there is no clear way to patch it. The clear solution does not exist since multiple parties in the system are involved and tangled each other. It is not because Kubernetes community constantly procrastinates or neglects them. Most of these 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 contradictorial 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 nubmer 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 failrues 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 deschduler config. It is possible that the cluster resource utilization looks stable even though what is happening beind is deschduler keeping evicting pods from high utilization node and scheduler placing pods in low utilization node makig the utilization high again. The third reasons is that semantically contradictorial configurations does not mean it will always lead to failrues. Many of these problems are triggered in a specific number of nodes and pods.

Common patterns in multi-controller failure cases Table.1 summarizes the failure cases we analyzed and reproduced in this paper. 10 failure cases are analyzed and 8 out of 10 are reproduced in Kubernetes cluster. KinD cluster was used for failure reproduction.

Reproduced in a small scale The first common pattern is all of failure cases presented in this paper does not need large scale cluster to reproduce them. Maximum number of nodes and pods used for reproduction are 3 and 6 respectively. It implies that they are substances of large scale cluster. Testing tool or verification program may want to be acknowledged by this fact when it comes to making search algorithm efficient and fast.

Observation 1

4 Related Work

A single controller failure study

K8s operator failure

Testing

Chaos engineering

Model checking

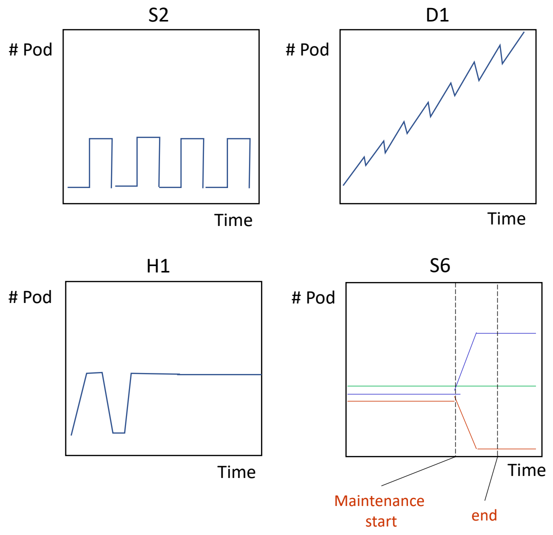

Figure 1. Number of pods over time for four failure cases.

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. Summary of multi-controller failure cases. Reproduced cases are marked with (R).