A Failure Study On Multi-Controller Failures in Cluster Management System

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

Abstract

[1] Modern cluster management system is composed of a collection of loosely coupled control components. They are independent and do their job without being directed commanded by other control component on the surface level. However, once you go one level deeper, they are directly or indirectly interacting with each other. In this paper, we will deeply understand how **multiple controllers** in modern cluster manager can cause pathological or non-optimal behavior. We reproduced 10 failure cases involving multiple controllers and each failure is caused by different combination of controllers. The detailed failure analysis raises the red flag to users and explain how and why they occur to help them to get sense of what multi-controller failures are. Furthermore, based on what is observed in our failure reproduction, we suggest some promising solution to claim that we need additional layer to predict and prevent potential multi-controller failures before it cracks the cluster.

1 Introduction

Modern cluster management system is composed of a collection of loosely coupled control components. Each controller operates asynchronously in distributed fashion. They fulfill their own dedicated duties independently, hoping the entire cluster to converges on desirable status eventually. There is no single global controller who has the perfect knowledge of the entire system. By nature of distributed controller design, however, there is no guarantee that the entire cluster will not fall into undesirable status such as oscillating phenomenon.

This is not just because it is such a huge and complex system but there are two concrete reasons facilitating the system to be unintentionally unstable. One is the design of the system and the other is the development environment. First, control realms of multiple controllers in the cluster sometimes are not evidently exclusvie but could be overlapped with each other. For example, scheduler and descheduler both are involved in pod scheduling. Inherently it is possible that they behave in contradictory way that configurations in multiple controllers diverge and cluster status never converges or go over sub-optimal path to stable status. Seocnd, each controller 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. Covering all conditions that could be possibly induced by multiple controllers' interaction requires inhibitively large search space which is not practical and should not be pursued.

In this paper, we will deeply understand how multiple controllers in modern cluster management system can cause pathological or non-optimal behavior. Further, we will provide some insight and promising solution for this problem. To dive into more specific failure cases, we pick Kubernetes container orchestration system as a representative example which is the most widely used cluster management system. We analyzed and reproduced 10 different failure cases which were reported in Kubernetes github issues, Kubecon (Kubernetes conference), and blog posts.

2 Background

There have been many cluster management system such as Borg [22], Omega [18], Kubernetes [9] from Google and Twine [21] from Meta. These systems consist of multiple controllers which play a role in administering a part of the cluster management. Each controller directly or indirectly interacts with each other by reading from and writing to consistent shared key-value store to keep cluster in a certain desirable status [4, 8]. The objects in key-value store represents controllers as well as all other components comprising the cluster. A controller has its own defined desirable status regarding the dedicated role. The controller periodically checks the cluster status whether it derails from the desirable state. If it does, then it takes appropriate action. This execution model is called *reconciliation*. Each controller has its own reconciliation logic, basically control loop and the desirable state is defined by the configuration given by Kubernetes user. For example, *Horizontal Pod Autoscaler (HPA)* [10] is autoscaling pods based on the target resource utilization along with the minimum and maximum number of pods. It periodically checks the number of pods and if the average resource utilization diverges from the target value, it will increase or decrease the number of pods within the min-max boundary. More details will be covered in each failure case study in section 3

[1] This work is done with collaboration with Bingzhe Liu in UIUC and Brighten Godfrey in UIUC.

3 Case study

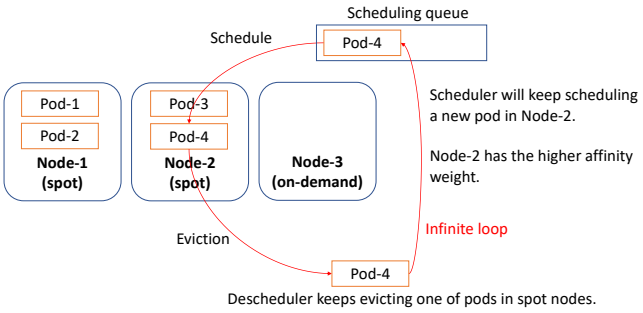


Figure 1. Failure case S2.

Failure cases

In this section, we will walk through 8 failure cases to provide detailed picture of how multi-controller can fall apart. All failure cases are elephant in the room. The problem exists clearly right there in the cluster, nibbling away at cluster resources but none of controllers can recognize it or stop it. *S3* case will not be described in this paper. It is explained in Kubernetes official document [7].

D1, Deployment + Kubelet (Taint) Deployment controller [12] manages updates for pods. It has *nodename* configuration which is one way of restricting pods to run in a specific node. *Kubelet* [11] is node managment agent. Node can be marked by *Taint* configuration to restrict pods that can be scheduled in the node. Pod is required to have corresponding *Toleration* in order to be schedulable in the tainted node. The first pathological behavior is triggered if *nodename* in deployment and *Taint* in Kubelet try to do exactly the opposite scheduling. For example, deployment specifies *node-1* for explicit pod scheduling. However, the node is tainted and pod does not have the corresponding toleration. In this case, two conflicted controllers will create infinite loop of Deployment controller scheduling pods to node-1 and Kubelet controller evicting pods from node-1. Apparently, nothing will stop to form a vicious scheduling cycle. Furthermore, *TerminationGracePeriodSeconds* configuration in deployment lets you handle termination gracefully (default: 30s). It can result in a large number of pods in *terminating* status, still taking up cluster resources.

H1, HPA + App CPU utilization change *Horizontal Pod Autoscaler (HPA)* [10] is in charge of automatically scaling up and down based on defined scaling rule (default: 50% CPU utilization). If average pod CPU utilization of a deployment exceeds 50%, HPA will create new pods to satisfy the target metric value. HPA does not take into consideration why CPU utilization increases. One common source of increase in CPU utilization could be surge in the number of incoming requests. In this case, scaling up is valid reaction. However, there are other sources of CPU usage. For example, garbage collection, initilization phase of application or any other forms of computation happening inside application are going to increase CPU utilization as well. For these cases, even if scaling up will not help to reduce the CPU utilization, it will scale up since HPA differentiate where CPU increase comes from. Unintended scale up is critical issue, leading to increase in cost. Again, note that none of application and HPA controller is doing anything wrong individually.

H2, Deployment + HPA You can specify the number of replicas in deployment configuration (Default: 1). A deployment is applied with empty replica field and it creates 1 replica. HPA is applied later with configuration of 6 minimum replica and now the number of replicas becomes 6. At some point, you applied a new configuration for the deployment and at this time, the replica field is defined to 3. One of two controllers' replica configuration should be ignored. However, what happens is the number of replicas becomes 3 temporarily for a few seconds by deployment and 6 again by HPA. If the intented number of replica is 6, the change is sub-optimal. If the intent is 3, it fails due to duplicate configuration.

S1, Scheduler + Descheduler Scheduler and Descheduler could be configured to prefer low utiliation node or high utilization node. Default behavior of scheduler is spreading pods as much as it can to make it more fault-tolerant. The purpose of preferring high utilization node is to enable bin packing, retains less number of nodes required to run the cluster and save the overall cost. Unnecessary scheduling and descheduling cycle could be created by confliction between the two controllers. Scheduler is configured to place pods in high utilization node and descheduler is condfigured to evict pods from high utilization node.

S2, Daemonset + Descheduler *Daemonset* will manage a part of scheduling rule. One of them is *NodeAffinity*. It schedules pods based on affinity weight. The problematic situation is daemonset checks the node affinity and scheduler keeps placing pods in node-1. At the same time, descheduler is running with *RemoveDuplicate* configuration which forces only one pod to run in a node and evicts the rest of them. It is per deployment policy. NodeAffinity and RemoveDuplicate are conflicted each other and will create endless cycle of termination, eviction and creation of pod. *S5* arises in a similar fashion by image locality config in deployment and descheduler.

S6, Node maintenance + Scheduler From time to time, nodes could be put in maintenance, for example to add a new library, update security feature. Onec the node is shut down, pods in the node will be terminated and scheduled to another node. For instance, there are three nodes and three

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)	Lack of context	HPA + Node reachability	Safety	Semantically wrong avg CPU util (reachability vs healthiness)
S1	Conflicted config	Descheduler + Daemonset	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 skewed 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).

replica, one replica running in each node. The node-1 is going through the maintenance and automatically the pod running there is moved to the node-2. The overall placement becomes skewed but it is the best possible movement at the moment. Later, the maintenance finishes and node-1 begins operation again. The expectation is rebalancing the pod spread by moving one pod from node-2 to node-1. Apparently, it will not happen unless descheduler is installed with right plugin (Descheduler is not default controller). This is not the optimal placement and reduces availability by failing appropriate pod spread.

S7, Scheduler + Kubelet In this failure case, scheduler is configured with maxSkew of 1. Maximum skewness allowed is 1. The deployment creates 3 pods and scheduler will place one pod per a node, pod-*{1,2,3}* in node-*{1,2,3}* respectively in this example. However, node-3 becomes *NotReady* since kubelet in node-3 is unreponsive for some reason. Pod-3 is not able to be scheduled or will be evicted if it was running. It is expected that node-3 should not be counted when censoring the maxSkew and pod-3 is still scheduled in one node among healty nodes, node-1 and node-2. It does not violate maxSkew:1 rule (two pods in node-1 and one pod in node-2 or the other way around). What actually happens is pod-3 will not be scheduled and remains pending indefinitely since node-3 is still included for maxSkew calculation. It needs to be investigaed further to diagnose the root cause. Regardless of why it occurs, it is counter-intuitive to human that a pod cannot be scheduled due to unresponsive node. It means it is hard to diagnose it when it happens and Kuberentes again does not alert any warning. It is another potential black hole that will suck human resources, wasting their time to look into the issue.

4 Common Patterns and Insights From the Failure Cases

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.

Most of failure cases involves scheduling behavior 7 out 10 failure cases are directly or indirectly associated with scheduler. Some failure cases does not even require multiple controllers, e.g., S3, S5, S7. Even two conflicted configurations within a scheduler controller can result in failure. Kubernetes is composed of numerous controllers and other system components. Notably the schduler is one of the most complicated controllers. In addition how complicated it is, there are two fundamental design choices of kube-scheduler causing these failures. First, kube-scheduler does not revisit its previous scheduling decision. Once it schedules a pod in a node, the pod will never be re-scheduled to another node unless it is forced by external events like node failure or node taint. Kube-scheduler tries best effort at the time of scheduling and hands off. Even if first placement met all scheduling requirements, it can be violated later by future events. In S6 case, when pods were scheduled, they were spread by default scheduler policy. However, placement becomes skewed when the third node goes into maintenance and all pods in the node are evicted and scheduled in other nodes. Even after the third node becomes alive and able to host pods, the scheduler will not fix the skewed placement.

Needs for middle layer tools As confirmed in the failure case study, Kubernetes currently does not have any type of system layer preventing or warning configurations violating safety or liveness property. It is intrinsically challenging for Kubernetes to support additionaly prevention layer of safety and liveness property. There are three reasons that positions it in fundamentally difficult problem. First, there is no way to express user's intent in the current Kubernetes version. Someone might say just implementing a new layer doing that particular job will solve the problem. However, it is not trivial in which way users should express their intents. Which format should new commands look like, which commands should be added, etc. On top of that, it is putting extra burden to users who needs to learn additional commands.

Kubernetes configurations are already sufficiently complicated. Second, to avoid multi-controller problems at the first place, one possible rather blind solution is removing features that contributes to potentially contradictorial configurations existing in mutliple controllers. For example, MostAllocated config in scheduler and high node utilization plugin in descheduler are pursuing the opposite purpose. One is trying to do bin packing (placing pods in high utilization node) and the other is trying to evict pods from high utilization nodes, hoping them to be scheduled in low utilization node. We can circumvent this probable multi-controller confliction by deprecating high node utilization plugin in descheduler. The same goal can be fulfilled by appropriate pod spreading configuration in scheduler. Although they might represent the same goal in high level, it can never be exactly same. It is simply because the mechanisms they achieve it are different - one is schedule and the other is eviction. They are in charge of exclusive roles. Third, the confliction does not manifest always by turning on those plugins. It also depends on the exact configurations like threshold for each setting. Deleting conflicted features is overkill since the presence of potential confliction does not always lead to problem as well as it is not appropriate solution since their functionalities are fundamentally different. Our conclusion is a need for middle layer verification tool. It takes input of Kubernetes yaml config files as they are and outputs whether that certain combination of configurations contains potential multi-controller problems. This new middle layer tool cannot fix the problem automatically since it still is not able to figure out user's intent and it should not be designed to do that because it would be too complicated. The new layer execution should happen offline or at least should be run in background. In other words, ti should not be in the critical path that can hurt the performance of cluster management task. We do not know what the most suitable verification technique is for this problem. We leave it for future work.

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. When building testing tool or verification program, this observation can be leveraged to help their algorithms to be more efficient by reducing search space it needs to explore.

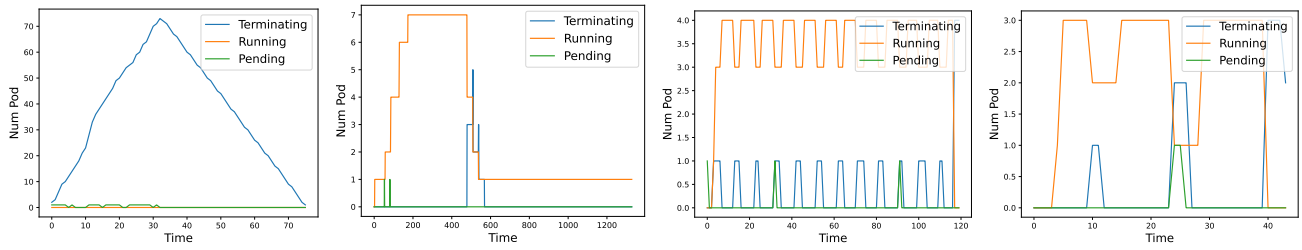
5 Why does it happen at the first place?

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



(a) D1 - Deployment + Kubelet **(b)** H1 - HPA + App CPU changes **(c)** S2 - Descheduler + Daemonset **(d)** S6 - Scheduler + Node maintenance

Figure 2. Number of pods over time for four failure cases.

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.

6 Related Work

Chaos engineering Chaos engineering [1–3, 5, 6, 13] builds the more confidence in enduring tumultuous situation by running experiments in a way that could be a source of crashing the system. One example pattern is randomly terminating arbitrary instances such as pod in kubernetes. However, multi-controller failures are not triggered by such faults in the cluster. They cannot be revealed by chaos engineering.

Testing There exists a long history in testing distributed system [15–17, 19, 23]. The most related testing tool is Sieve [20] utilizing precise fault injection technique. However, there are two key difference. First, it is Sieve only focused on a single controller failure. It is not able to test cases that multi-controllers are involved. Second, the primary goal of Sieve is to find actual bugs in the system with precise fault injection technique. In contrast, the multi-controller failure cases presented in this paper are not strictly defined as bugs even if it exhibits pathological behavior. Hence, these failures will not be detected and cannot be resolved by bug fix patch.

Model checking Model checking [14] is technique that automatically verifies if the model satisfies all the required properties. The system can be modeled and the tools are going to cover all the possible combination of execution path. It could be a promising solution to judge if given configuration could put the system in undesirable status. This approach has several challenges. One is modeling the system correctly so that it can capture how the program works. It is also not one time effort. When a newer version is released, the model should be updated accordingly to keep high fidelity. Second is defining properties. It depends on which properties it wants to verify. Infinitely oscillating number of pods could be one example of liveness property for Kubernetes. The model checking can be the promising way to find the multi-controller failures in a specific set of configuration and furthur can explain the source of the failure.

7 Conclusion

For the best of our knowledge, this is the first failure case study on multi-controller in cluster management system. We found that systems which is managed in distributed manner is highly vulneralble to the multi-controller failures. The 10 cases shown in this paper describe how multi-controller can craft the self-destroying situation which never converges on any good state and sub-optimal status updates which is very likely to confuse the users. We alert users to the fact that multi-controller issues are prevalent in cluster management system and suggest a new layer of verifier to prevent them before it happens.

8 Metadata

The presentation of the project can be found at:

https://drive.google.com/drive/folders/1QEGSYBYxbBwJxcIOLNg2Q76X

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

https://github.com/gangmuk/k8s-failure-reproduction/tree/main

References

[1] Chaos kube, periodically kills random pods in your kubernetes cluster. https://github.com/linki/chaoskube, 2020.

[2] Chaos mesh, a solution for system resiliency on kubernetes. https://github.com/chaos-mesh/chaos-mesh, 2020.

[3] Chaos monkey. https://github.com/Netflix/chaosmonkey, 2020.

[4] Etcd, a distributed reliable key-value store for the most critical data of a distributed system. https://github.com/etcd-io/etcd, 2020.

[5] Kube monkey, periodically kills random pods in your kubernetes cluster. https://github.com/asobti/kube-monkey, 2020.

[6] Pumba, chaos testing tool for docker. https://github.com/alexei-led/pumba, 2020.

[7] S3 - Conflicting topology spread constraints. https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/#example-conflicting-topologyspreadconstraints, 2020.

[8] Zookeeper, a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. https://zookeeper.apache.org, 2020.

[9] Kubernetes. https://kubernetes.io/docs/concepts/overview/, 2023.

[10] Kubernetes Horizontal Pod Autoscaler controller. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2023.

[11] Kubernetes kubelet controller. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/, 2023.

[12] Kubernets Deployment controller. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, 2023.

[13] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. Chaos engineering. *IEEE Software 33*, 3 (2016), 35–41.

[14] Holzmann, G. The model checker spin. *IEEE Transactions on Software Engineering 23*, 5 (1997), 279–295.

[15] Lukman, J. F., Ke, H., Stuardo, C. A., Suminto, R. O., Kurniawan, D. H., Simon, D., Priambada, S., Tian, C., Ye, F., Leesatapornwongsa, T., Gupta, A., Lu, S., and Gunawi, H. S. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[16] Ozkan, B. K., Majumdar, R., Niksic, F., Befrouei, M. T., and Weissenbacher, G. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang. 2*, OOPSLA (oct 2018).

[17] Ozkan, B. K., Majumdar, R., and Oraee, S. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang. 3*, OOPSLA (oct 2019).

[18] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13.

[19] Sun, X., Cheng, R., Chen, J., Ang, E., Legunsen, O., and Xu, T. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 735–751.

[20] Sun, X., Luo, W., Gu, J. T., Ganesan, A., Alagappan, R., Gasch, M., Suresh, L., and Xu, T. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 143–159.

[21] Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., and Zhang, P. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 787–803.

[22] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[23] Yuan, X., and Yang, J. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 1141–1156.