Advances in Artificial Intelligence, Big Data and Algorithms
G. Grigoras and P. Lorenz (Eds.)

doi:10.3233/FAIA230909

Min-Leader Optimal Scheduling Algorithm in Kubernetes Clusters

Yongkang DING, Junnan LIU[1], Xin HE
School of Software, Henan University, Kaifeng, 475001, China

Abstract: The development of containerization and virtualization technologies has made Kubernetes a popular container orchestration platform that allows multiple replicas for scalability and availability. For stateful applications, each replica requires persistent storage, and a leader must be elected among the replicas to handle client requests and ensure data consistency. However, leaders for multiple stateful applications can become unbalanced across nodes, leading to inefficient resource utilization and load imbalance. This paper proposes the Min-Leader optimal scheduling algorithm that balances leader distribution across nodes during the scheduling phase of Pods. The algorithm considers the number of leaders on each node and prioritizes nodes with the least number of leaders with the goal of improving cluster availability and stability. The algorithm is integrated into the default Kubernetes scheduler as a plugin and validated using experiments. The algorithm effectively balances workload between nodes, improving cluster availability and stability.

Keywords: Kubernetes, stateful, Min-Leader optimal scheduling algorithm, load balancing

1. Introduction

The adoption of container technology for application deployment has grown in popularity, in part due to the emergence of cloud computing and microservice architecture [1]. Containers are lightweight, provide strong isolation, and high portability, enabling fast creation, startup, and destruction. These aspects make them ideal for constructing distributed applications on a large scale. However, managing a large number of containers, guaranteeing their communication and collaboration, and enhancing their security and reliability has been a challenging task for container technology. Large-scale applications require a container orchestration system to implement standardized deployment, management, and monitoring of containers. One of the most commonly used and stable container orchestration systems in industry is Kubernetes [2]. Originally created and led by Google, it is an open-source framework. Kubernetes operates using a master-slave architecture, consisting of one or more master nodes and several worker nodes. The master node is responsible for managing the state and configuration of the cluster, while the worker node runs the application containers, reports node status, and communicates resource information to the master node.

Kubernetes clusters categorize applications as either stateful or stateless. Stateful applications require stored and persistent data states, unlike stateless applications that

[1]Corresponding author: Junnan LIU, School of Software, Henan University;
e-mail: 18637874722@163.com

do not require any data storage and can be created or destroyed without impacting quality of service. Stateful applications usually rely on a leader-based consistency maintenance mechanism [3] to maintain consistency between replicas. To implement a leader-based consistency maintenance mechanism using the Kubernetes leader election algorithm [4], stateful applications elect a leader among the replicas to manage client requests. The system controls the transfer of leader power through the Lease resource lock.

StatefulSet serves as an API object used to manage stateful applications as a workload effectively. With StatefulSet, the deployment and scaling of a group of Pods can be managed, with the advantage of ensuring their order and uniqueness. Pods get deployed sequentially and compete for control over the Lease resource lock, with the first replica that gains control of the lock becoming the leader. The Scheduler uses a scheduling algorithm to determine the node on which these replicas run, introducing randomness into the selection process of the node. There is a possibility of workload imbalances, underutilized cluster resources, and other related issues caused by the uneven distribution of multiple stateful applications leaders among nodes. Resource scarcity and quality reduction of other services on that node are possible when multiple leaders converge on that specific node.

This article presents the Min-Leader optimal scheduling algorithm as a solution to balance the distribution of leaders among nodes during the Pod scheduling phase. The algorithm introduces novel evaluation indicators to assess the number of leaders on nodes during the Optimal Scoring Phase of Pod scheduling. Nodes with lower leadership density receive a higher score, leading to the pod being scheduled on those nodes to achieve a relatively equitable distribution of leaders in the cluster. The Min-Leader scheduling algorithm requires no modifications to the properties or labels of Pod or StatefulSet objects, Lease resource lock, or leader election algorithm. Instead, it necessitates adding a new optimal scheduling plugin to the Scheduler and incorporating it into the Optimal Scoring Phase.

In this paper, the contents will be organized as follows: Section 2 will provide an introduction to the research work related to this paper. Section 3 will provide a detailed discussion of the design and implementation of the Min-Leader optimal scheduling algorithm. Section 4 will present the experimental results and analysis. Finally, Section 5 will summarize the findings of the paper and provide suggestions for future research.

2. Related Work

Kubernetes is a popular and mature container orchestration system that faces challenges in optimizing cluster load balancing. Numerous research works currently explore and improve on this issue. Some studies specifically focus on optimizing the design and implementation of the Kubernetes scheduler to improve cluster resource utilization and application performance. For example, Li et al. [5] discuss the design and implementation of the scheduler while summarizing dynamic resource management in Kubernetes, and suggest relevant improvements. Yang et al. [6] consider resource usage and propose a strategy based on a weighted graph model to schedule containers on nodes. Zhang et al. [7] propose a Kubernetes-based container coordination strategy for edge computing systems, which adaptively allocates computing and storage resources. Qiu et al. [8] suggest a distributed Kubernetes architecture that supports collaborative work between cloud and edge environments. In

addition to optimizing the scheduler for load balancing, Vayghan et al. [9] proposed a solution that is able to automatically direct client requests to healthy pods. This solution integrates a state controller with Kubernetes, enabling it to determine the status of the pods and assign "active" or "standby" labels to them. Client requests are redirected to the pods with the "active" label, and messages containing state data are copied to the standby pods, to which messages containing state data are copied. Nguyen et al. [3] propose a leader-based consistency maintenance mechanism for stateful applications. The mechanism selects a leader among all replicas to process write requests and synchronize data changes. However, the high workload of the leader and the limitations of Kubernetes' leader election algorithm can result in an unbalanced resource load in the cluster. This paper validates the importance of evenly distributing leaders in the cluster for improving load balancing and scalability.

The paper proposes a novel method to achieve this at the scheduling layer by evenly distributing multiple application leaders, with the aim of researching and demonstrating the domestically produced operating system used in the Kunpeng platform.

3. Design and Implementation of Min-Leader Optimal Scheduling Algorithm

This study addresses the issue of uneven leader scheduling of multiple stateful applications among nodes by the Kubernetes scheduler by introducing Min-Leader, an optimal scheduling algorithm, and details its design and implementation.

3.1. Algorithm Design

This paper proposes the Min-Leader optimal scheduling algorithm to alleviate the node performance degradation issue caused by a disproportionate number of leaders in some nodes. This algorithm aims at achieving a balanced scheduling of stateful application leaders among nodes. The algorithm scores the nodes based on the number of leaders, where nodes with fewer leaders receive higher scores. This prioritizes scheduling of the first copy of an application to the lowest scoring node, thereby ensuring equitable distribution of leaders in the cluster.

The algorithm comprises two steps: firstly, acquiring the leader count on each node, and secondly, scoring the nodes based on such count. Eq. (1) guides the scoring method wherein i represents the node number, $L(i)$ is the number of leaders on the i-th node, and $L_{Cluster}$ represents the total number of leaders in the cluster.

$$Score(i) = 10 * (1 - \frac{L(i)}{L_{Cluster}}) \tag{1}$$

3.2. Algorithm Implementation

The Min-Leader optimal scheduling algorithm consists of two steps.

Step 1 involves obtaining the number of leaders on each node. To do this, we access the Leases API in Kubernetes API, which grants us access to all the Lease objects created by stateful applications within the namespace. We then gather the

names of the leader replicas from the HolderIdentity property of each Lease object. Finally, we use the Kubernetes API to query the nodes that comprise each leader and count how many distinct leaders are on each node, as well as the total number of leaders in the whole cluster.

Step 2 is where we score the nodes. We only score the first replica of a stateful application. This is due to the fact that it is the first replica that initializes the acquisition of the Lease resource lock, ultimately becoming and staying as the leader. Other replicas are not taken into account during the scoring phase. When the first replica begins the scheduling process, we obtain the number of leaders within the cluster and the node that needs to be scored. We then use Formula 1 to calculate the score of that node within the Min-Leader optimization phase. We repeat the process for all nodes and select the node with the highest score for binding.

To implement our custom solution, we leveraged the pluggable scheduling framework of Kubernetes. Our custom scheduler integrated the Min-Leader optimal scheduling algorithm as a plugin, which would be used for the Score extension point. We left the other extension points to use Kubernetes' default policies.

4. Experimental Results and Analysis

In order to validate the effectiveness of the Min-Leader scheduling algorithm, a Kubernetes cluster was constructed on the Kunpeng server for experimental purposes. The server runs on the ARM architecture with the Kubernetes and Docker versions of 1.23.4 and 20.10.9 respectively. The cluster has a single master node and five worker nodes. Each worker node has 8 CPU cores and 8GB RAM, whereas the master node has 16 CPU cores and 16GB of RAM. Various stateful applications were deployed in the cluster. To test these applications, requests were sent through the Hey stress testing program [10] using NodePort services.

4.1. Leader Distribution

To evaluate the distribution of leaders, we deployed ten stateful applications using both the Kubernetes scheduler and the Min-Leader scheduler. Following deployment, we compared the resulting distribution of leaders between the two schedulers, as presented in Table 1. Our findings revealed that, after being deployed by the Kubernetes scheduler, the distribution of leaders was uneven, with five leaders present on node4 but none on node5. This imbalance could heavily overload node4, ultimately compromising its performance due to the inherent high workload of leaders. Contrastingly, when the Min-Leader scheduler was used, leaders were evenly distributed across all five nodes, leading us to the conclusion that the algorithm can successfully balance the distribution of leaders. As a result of this, individual nodes are not overloaded, and overall load balancing between nodes is maintained.

Table 1. The distribution of leaders among nodes.

Worker Node	node1	node2	node3	node4	node5
Kubernetes scheduler	2	2	1	5	0
Min-Leader scheduler	2	2	2	2	2

4.2. Leader Distribution's Impact on Node Resources

Write requests were sent to each application for 100 seconds using a single client, while measuring the average CPU utilization of each worker node (refer to Figure 1). The results indicated that uneven leader distribution through the Kubernetes scheduler led to some nodes with high CPU utilization rates, specifically, the utilization rate of node4 reached 74.26%, while others were very low, with node5 at only 3.4%. This resulted in cluster load imbalance. The results indicated that uneven leader distribution through the Kubernetes scheduler led to some nodes with high CPU utilization rates, specifically, the utilization rate of node4 reached 74.26%, while others were very low, with node5 at only 3.4%. This resulted in cluster load imbalance. In contrast, the Min-Leader scheduler retained each node's CPU utilization rate at approximately 40%, while maintaining load balance.

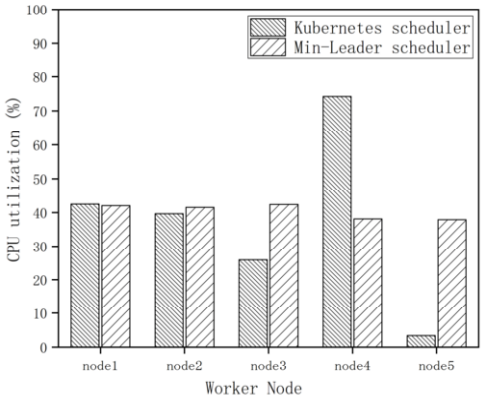

Figure 1. CPU use on each node when clients send write requests.

As a result, the Min-Leader scheduler enables evenly distributed leaders, enhanced cluster load balancing performance and mitigates risks associated with single-node failures.

5. Conclusions and Future Directions

This article proposes the Min-Leader optimal scheduling algorithm to improve the Kubernetes scheduler's load balancing capability for the problem of unevenly distributed leaders in multiple stateful applications. The algorithm introduces a new evaluation index, i.e., the number of leaders each node holds, to score the nodes accordingly. It prioritizes the selection of nodes with the least number of leaders. Its effectiveness and feasibility have been experimentally proven. However, the newly elected leader may disrupt the balance between nodes if a previous leader fails. In the future, we plan to improve the Kubernetes leader election algorithm to ensure that the load balancing between nodes remains balanced.

Acknowledgements

This work was supported by a grant from Postgraduate Education Reform and Quality Improvement Project of Henan Province Under Grant NO.YJS2022JD26, Postgraduate Education Reform and Quality Improvement Project of Henan University Under Grant NO.SYLJD2022008 and NO.SYLKC2022028, the Key Technology Research and Development Project of Henan Province under Grant 222102210055.Major Science and Technology Special Project of Henan Province, Research and Demonstration of Kunpeng Platform-based Domestic Operating System under Grant 201300210400.Supported by Research on Key technologies of resource scheduling and service High Availability based on ARM architecture, Project No. 232102210199.

References

[1] Wu, Zhixue, "Development and Trends of Virtualization Technology in Cloud Computing." Journal of Computer Applications, vol. 37, no. 4, pp. 915-923 (2017).
[2] Kubernetes, Kubernetes Documentation. https://kubernetes.io/docs/home/ (accessed Apr. 6, 2023).
[3] N. Nguyen and T. Kim, "Toward Highly Scalable Load Balancing in Kubernetes Clusters." IEEE Commun. Mag., vol. 58, pp. 78–83 (2020).
[4] Kubernetes leader election algorithm, "Simple Leader Election with Kubernetes and Docker." Available online: https://kubernetes.io/blog/2016/01/simple-leader-election-with-kubernetes/ (accessed Apr. 10, 2023).
[5] X. Li et al., "Dynamic Resource Management for Kubernetes: A Survey." ACM Computing Surveys, vol. 53, no. 3, pp. 1-40 (2020).
[6] H. Yang et al., "Resource-Aware Container Scheduling in Kubernetes Cluster." IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 4, pp. 866-877 (2020).
[7] S. Zhang et al., "Container Coordination with Kubernetes for Edge Computing Systems." IEEE Internet of Things Journal, vol. 6, no. 2, pp. 2013-2023 (2019).
[8] W. Qiu et al., "Distributed Kubernetes: Building Hybrid and Edge Clouds." IEEE Network, vol. 33, no. 1, pp. 64-71 (2019).
[9] Vayghan, L.A. et al., "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes." 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), Sofia, Bulgaria, July 22-26, 2019.
[10] Hey, "A Tiny Program Sends Some Load to a Web Application." https://github.com/rakyll/hey (accessed May. 10, 2023).