Cluster Computing (2024) 27:7241–7250
https://doi.org/10.1007/s10586-024-04333-6

A balanced leader election algorithm based on replica distribution in Kubernetes cluster

Junnan Liu[1] · Yongkang Ding[1] · Yifan Liu[1]

Received: 1 December 2023 / Revised: 9 January 2024 / Accepted: 1 February 2024 / Published online: 17 March 2024

Abstract
Kubernetes is a well-known open source project that provides a powerful orchestration platform for containerized applications. To ensure high scalability and availability of services, redundant deployment is usually adopted in Kubernetes clusters, creating multiple replicas for each application. Each replica of a stateful application needs to persistently store data and use a leader-based consistency maintenance mechanism to ensure strong consistency among replicas. In this mechanism, the elected leader is responsible for updating data and synchronizing it to the followers, which results in a higher workload for the leader. When there are multiple stateful applications, the Kubernetes leader election algorithm does not consider the distribution of multiple leaders among nodes, which may lead to the phenomenon of too many leaders on some nodes. This can reduce system performance due to the high workload of the leaders themselves. To address this problem, we propose a balanced leader election algorithm based on replica distribution, which enables multiple stateful application leaders to be evenly distributed among the cluster's worker nodes. The algorithm effectively solves the problem of system performance degradation caused by leader concentration and achieves load balancing among nodes. We verify the effectiveness of the algorithm through experiments.

Keywords Kubernetes · Stateful · Consistency · Balanced leader · Load balancing

1 Introduction

Recently, container-based virtualization technology has emerged as the predominant method for deploying applications in cloud computing [1]. Containers, a lightweight virtualization technology [2], run on the host operating system and encapsulate only the application and its dependencies. This approach significantly reduces system resource consumption and enhances application deployment efficiency. Compared to traditional virtualization, container technology enables faster application startup, shutdown, and migration, while enhancing isolation and facilitating more convenient management methods [3]. Large-scale systems require container orchestration tools to effectively manage numerous containers. Kubernetes stands out as one of the extensively employed container cluster management tools [4]. It enables swift deployment, automatic scaling, and fault recovery, delivering substantial convenience for developing and deploying cloud-native applications [5]. For ensuring high service availability and scalability, Kubernetes commonly employs a redundant deployment strategy, configuring multiple replicas for each application. In the event of a replica failure or increased load, the cluster can automatically execute failover and load balancing [6], thereby guaranteeing uninterrupted service provision. StatefulSet is a specialized API object in Kubernetes designed for managing stateful applications. Key features involve assigning stable network identifiers, providing persistent storage (PV), and orchestrating the deployment and scaling of Pods in a predetermined sequence. However, StatefulSet exhibits significant differences from other Kubernetes objects like Deployment and ReplicaSet. While these objects manage Pods, Deployment

✉ Junnan Liu
ljn@henu.edu.cn

Yongkang Ding
18637874722@163.com

Yifan Liu
3303824807@qq.com

[1] School of Software, Henan University, Kaifeng 475001, China

and ReplicaSet are better suited for stateless applications as they emphasize Pod quantity over individual Pod identity. In contrast, StatefulSet ensures the uniqueness of each Pod and provides persistent storage.

In Kubernetes clusters, stateful applications require data state preservation and persistent data storage to ensure data recovery during failures or scaling operations [7]. To achieve data consistency among replicas, these applications commonly utilize a leader-based consistency maintenance mechanism [8]. As depicted in Fig. 1, each Pod in this mechanism comprises two containers: a leader election container and a main container. The leader election container is responsible for electing leaders among replicas. Kubernetes offers a straightforward leader election algorithm [9], which this paper calls KUBE-LE, to facilitate a leader-based consistency maintenance mechanism. It employs the Lease resource object of the Kubernetes API for distributed resource locking, where multiple replicas contend for control of the Lease object to assume the role of the leader. The leader periodically renews the timestamp and identity information in the Lease object to inform other replicas of its presence. This ensures that at any given time, only one Pod can become the leader. The leader election container also provides a simple web server that returns the current leader name. The main container is responsible for handling client requests. In the case of a write request, the main container will access the leader election container to determine whether it is the leader. If yes, it processes the request, otherwise, it redirects the request to the leader Pod. This ensures that all writes are handled by the leader, maintaining data consistency. For a read request, the receiving Pod directly handles it. In a strong consistency system, all replicas maintain the latest data state, enabling any replica to handle a read request.

This mechanism ensures data consistency and fault tolerance but places an additional load on the leader replica. Due to the design characteristics, the leader replica bears a heavier load than other replicas. In a cluster with multiple stateful applications, the KUBE-LE algorithm does not fully consider the distribution of leaders among nodes. This situation may lead to a concentration of multiple leaders on specific nodes, leaving others with fewer leaders. In such a scenario, the node hosting the leader faces a higher load, handling more write requests and data synchronization tasks, while other nodes may remain idle or experience low load [8]. Therefore, when implementing a leader-based consistency maintenance mechanism using the KUBE-LE algorithm, two issues may arise: an uneven distribution of leaders among nodes for multiple stateful applications, resulting in an excessive or insufficient number of leaders on certain nodes; and an uneven distribution of leaders causing imbalanced resource utilization among nodes, with some nodes experiencing excessive or insufficient usage of CPU, memory, network, etc. Both of these issues can impact the performance and stability of the Kubernetes cluster, consequently diminishing the service quality of stateful applications.

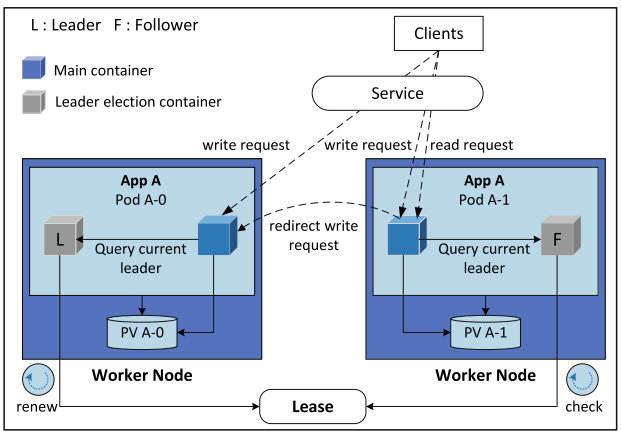

Fig. 1 Leader-based consistency maintenance mechanism

This paper presents a balanced leader election algorithm, named BRD-BLE, which is based on replica distribution, aiming to resolve the problem of centralized leadership in Kubernetes clusters hosting multiple stateful applications. This algorithm selects a leader replica from the node that possesses the fewest leaders, taking into account the distribution of replicas across stateful applications. By adopting this approach, a balanced distribution of leaders across active nodes in the cluster for multiple applications is ensured, thus preventing scenarios in which certain nodes become overwhelmed while others remain idle due to leader concentration. Experimental evaluations were carried out in a Kubernetes cluster, and the outcomes substantiate that the proposed algorithm yields a significant enhancement in node resource utilization and system throughput when compared to the KUBE-LE algorithm.

The subsequent sections of this paper are structured as follows: Sect. 2 provides an overview of related work; Sect. 3 offers a comprehensive depiction of the balanced leader election algorithm based on replica distribution; Sect. 4 showcases the experimental results and analysis; and finally, Sect. 5 presents the concluding remarks of this paper.

2 Related work

There have been numerous research papers investigating effective load balancing in Kubernetes cluster systems. Takahashi et al. [10] introduces a portable load balancer that can be used in any environment, facilitating the migration of web services. It implements a containerized software load balancer that operates within a Kubernetes

container cluster, utilizing the Internet Protocol Virtual Server (IPVS) provided by the Linux kernel. Liu et al. [11] proposes a multi-metric load balancer designed for Kubernetes. In addition to static load balancing strategies, this load balancer dynamically allocates requests based on the runtime status of servers and applications, enabling the configuration of more sophisticated dynamic load balancing strategies that can be tailored to specific business requirements. Dua et al. [12] presents an alternative algorithm for task scheduling. It configures the cluster for a specific type of task (real-time, data-intensive, etc.) and incorporates load balancing techniques through task migration. Lee et al. [13] implements a high-performance load balancer in a containerized environment, utilizing eBPF/XDP in the Linux kernel for traffic distribution, and providing easy management through Kubernetes. Experimental results show that the proposed load balancer achieves significantly better throughput performance compared to iptables DNAT, and the performance difference increases as the packet size decreases. Wang et al. [14] introduces a load balancing algorithm known as Dynamic Weighted Random Routing (DWRR). It effectively addresses load balancing among containers in a Kubernetes cluster that has heterogeneous CPUs. Botez et al. [15] proposes an innovative solution for implementing LBaaS (Load Balancer as a Service) in CLOUDUT, an academic private cloud infrastructure. It is based on Kubernetes and provides high availability and enhanced security by utilizing secrets during the Docker image build phase, allowing for real-time route updates with a maximum downtime of one second. Baresi et al. [16] introduces a novel autoscaling solution called KOSMOS for Kubernetes. It utilizes control theory to vertically scale containers and employs heuristic methods to address resource contention and horizontally scale containers while appropriately allocating resources. Pramesti et al. [17] introduces an autoscaler based on response time prediction for managing microservice applications in a Kubernetes environment. It utilizes a machine learning model to predict response time and calculates the number of pods required to meet the target response time for the application based on the prediction. Kim et al. [18] proposes a fragment leader distribution algorithm that maximizes the throughput of distributed data storage by evenly distributing fragment leaders among cluster members. This algorithm improves data storage throughput significantly by limiting the maximum number of leaders a cluster member can have based on monitoring the state of fragments within the cluster member.

The consensus problem is a fundamental challenge in distributed systems, where the goal is for replicas to achieve data agreement despite faults and network delays. Researchers have designed and applied several algorithms to tackle this problem in various Kubernetes scenarios. Paxos [19] is a widely recognized distributed consensus algorithm that achieves consensus when a majority of replicas agree on a proposed value. While the algorithm offers benefits like strong fault tolerance and availability, it presents challenges including complexity, implementation difficulties, low efficiency, and limited support for dynamic configuration. Raft [20] is a consensus algorithm employed to implement replicated state machines. It improves the understandability and implementability of Paxos by separating the logic and simplifying the design. The Raft algorithm decomposes the consensus problem into subproblems like leader election, log replication, and safety. It ensures leader liveness and detects failures using heartbeat mechanisms. ZooKeeper [21] is an open-source distributed coordination service that ensures coordination, correctness, consistency, reliability, and atomic operations in distributed applications. It can be utilized to implement functionalities like distributed locks, queues, elections, and publish/subscribe, thereby serving as a foundational service for distributed applications. Oliveira et al. [22] evaluates the performance of the Raft algorithm on physical machines and Docker containers managed by Kubernetes. The results demonstrate similar performance in both environments. Netto et al. [23] explores the integration of the Raft consensus protocol into containers managed by Kubernetes for achieving state machine replication. The paper compares the performance and resource consumption of KRaft and Raft on physical machines. It finds that KRaft exhibits similar performance to Raft but necessitates more network transmission, whereas Raft on physical machines demands more processing power and memory. Netto et al. [24] presents a solution for integrating coordination services in Kubernetes by leveraging etc as a shared memory to accomplish state machine replication. The solution introduces a lightweight protocol named DORADO that enables state machine replication within containers, thereby enhancing container fault tolerance and availability. Netto et al. [25] introduces Koordinator, a method for providing coordination services in Kubernetes that ensures consistency and availability of container replication using state machine replication technology. Koordinator functions as a lightweight service layer that can integrate with various consensus algorithms and applications, thereby enhancing flexibility and modularity in container management. Abdollahi et al. [26] presents a solution that enhances the availability of stateful microservices by implementing high availability management for Kubernetes. The solution is to integrate a high availability (HA) state controller with the Deployment and StatefulSet controllers in Kubernetes. It achieves state replication and automatic service redirection to healthy microservice instances through the management of secondary labels.

Simplifying the leader election process, Kubernetes offers a straightforward leader election algorithm [9]. It utilizes the Lease resource object of the Kubernetes API for achieving distributed resource locking. Multiple replicas compete to gain control of the Lease object and become the leader. The leader election algorithm in Kubernetes has lower overhead and a simpler implementation compared to the Paxos and Raft algorithms. Nguyen et al. [8] presents a consistency maintenance mechanism based on leader election is proposed for stateful services in Kubernetes clusters. They examines the leader election algorithm in Kubernetes and highlights the potential issue of multiple leaders being concentrated on the same node, leading to load imbalance and performance degradation. Through experiments, the paper showcases how leader distribution affects load balancing and throughput, emphasizing the need to optimize leader distribution. Therefore, this paper presents a balanced leader election algorithm that relies on replica distribution. The algorithm optimizes the Kubernetes leader election process to achieve load balancing by evenly distributing leaders across multiple nodes.

3 Balanced leader election algorithm based on replica distribution

In this section, a detailed introduction will be provided for the BRD-BLE Algorithm. The algorithm's main objective is to choose a suitable replica, among all replicas of an application, as the new leader in case the current leader is non-existent or fails. A suitable replica is determined by having fewer leaders on its node compared to the other replicas on their respective nodes. If multiple suitable replicas exist, one of them is randomly chosen. The algorithm utilizes the Lease resource object of the Kubernetes API to implement a distributed resource lock, ensuring the correctness and consistency of the leader election process. Subsequently, a detailed explanation of the data structures and execution flow involved in this algorithm will be presented.

3.1 Data structures involved in the BRD-BLE algorithm

To implement the BRD-BLE algorithm, two crucial factors must be considered: replica distribution and leader distribution. Replica distribution pertains to the worker nodes hosting each replica of the stateful application, while leader distribution indicates the count of replicas functioning as leaders on each worker node. These two factors determine the process of selecting a suitable leader from multiple candidate replicas of the stateful application. In order to obtain replica distribution and leader distribution, the utilization of Kubernetes' scheduler and API is necessary.

The Kubernetes scheduler assumes the responsibility of orchestrating the assignment of Pods to suitable nodes for execution. This process relies on a series of predicate and priority strategies. Notably, the SelectorSpreadPriority [27] priority strategy seeks to distribute multiple copies of Pods, which are under the management of the same service or controller, across different nodes as uniformly as possible. This allocation strategy aims to enhance the system's resilience to disasters and improve load balancing performance. By facilitating an equitable distribution of replicas among nodes, the strategy establishes fundamental support for a balanced leader election process. Consequently, this approach effectively acquires and manages replica distribution information, offering crucial data support for subsequent decisions related to leader election and load balancing.

Kubernetes API is the core component of Kubernetes cluster. It adopts RESTful style and provides a way to interact with the cluster. It can be used to create, update, delete and monitor various resource objects in the cluster, such as Pod, Service, StatefulSet and Lease. Utilize the Kubernetes API to obtain and update information related to replica distribution and leader distribution to implement the BRD-BLE algorithm. Through the List method of the Kubernetes API, obtain the information of all Pods managed under the StatefulSet controller corresponding to the application, and extract the name of the Pod and the name of the node where it is located. In this way, the data of the replica distribution can be obtained and identified as ReplicaDistribution (RD). RD is a piece of data, with the Pod name as the key and the node name as the corresponding value. This data is embedded in the code implementation of the BRD-BLE algorithm in the form of variables. Through the Create method of the Kubernetes API, a piece of data is stored on the Annotation of the master node to record the number of leaders on each worker node. We name this piece of data LeaderDistribution (LD). LD is a data structure with the node name as the key and the number of leaders as the corresponding value. Its update operation is the responsibility of the leader of each stateful application to ensure the correctness and consistency of LD. This design makes it possible to quickly obtain the number of leaders on any node through the Kubernetes API's List method.

Utilizing the RD and LD data, we can select an appropriate replica as the leader based on the leader count on the replica's respective node. A suitable replica is defined as the one with a lower leader count on its node compared to other replicas on their respective nodes. If multiple replicas meet the criteria, one of them is selected at random. Our objective is to achieve load balancing by minimizing the

disparity in leader counts among the worker nodes in the cluster.

3.2 The execution process of the BRD-BLE algorithm

The procedural flow of the BRD-BLE algorithm is illustrated in Fig. 2. In a stateful application, each replica assumes one of two states: follower or leader. During the initialization phase, all replicas are initially set to the follower state and endeavor to attain the leader status by competing for control of the Lease object. The entire execution process comprises the following key Steps:

Step 1: As a follower, the replica periodically checks for the existence of the Lease object. If the Lease object does not exist, indicating the absence of a leader, the replica proceeds to Step 2. Conversely, if the Lease object exists, signifying the presence of a leader, the replica advances to Step 3.

Step 2: The replica evaluates whether all replicas are operational to acquire comprehensive RD data. Given the sequential initiation of stateful applications, it is imperative to await the full deployment of all replicas before initiating the leader election process. This ensures the election of a suitable leader among replicas. If all replicas are running, the competing leader replica generates a new Lease object, and the initial replica to create it progresses to Step 4. In contrast, as long as one replica is not yet running, the running replicas persist in the follower state and return to Step 1.

Step 3: The replica retrieves the leader record from the Lease object and compares it with the observer record it maintains. Each replica maintains an observer record, which saves the most recently copied leader record from the Lease object and the timestamp when the observer record was updated. Discrepancies between two records indicate that the existing leader has renewed the Lease object. In such cases, the replica updates its observer record and the timestamp of the update, continues in the follower state, and returns to Step 1. When the records are identical, the replica examines whether the Lease object has expired by calculating the elapsed time from the timestamp of the last observer record update to the current time. If the Lease has not expired, signifying the validity of the current leader, the replica continues in the follower state, returns to Step 1. If the Lease object has expired, surpassing the LeaseDurationSeconds value, indicating the leader's failure to renew the Lease object, the replica proceeds to Step 4.

Step 4: The replica retrieves the current RD and LD data, calculate the minimum number of leaders on the node where each replica is running, and verify whether the number of leaders on the node where it is located matches the minimum value. A mismatch indicates the unsuitability of the replica to assume the leader role, prompting it to persist in the follower state and return to Step 1. A match, however, indicates the suitability of the replica to become the leader, leading it to Step 5.

Step 5: The replica assuming the leader role updates the leader record in the Lease object and the number of leaders in the corresponding node in the LD data. Additionally, the replica regularly renews the leader record in the Lease object to sustain its leader status. Ultimately, the replica acting as the leader assumes responsibility for managing all write requests from clients and ensuring synchronization of data updates among other replicas in the follower state.

4 Experimental results and analysis

To assess the benefits of the BRD-BLE algorithm compared to the KUBE-LE algorithm regarding leader distribution balance, node resource utilization, and system

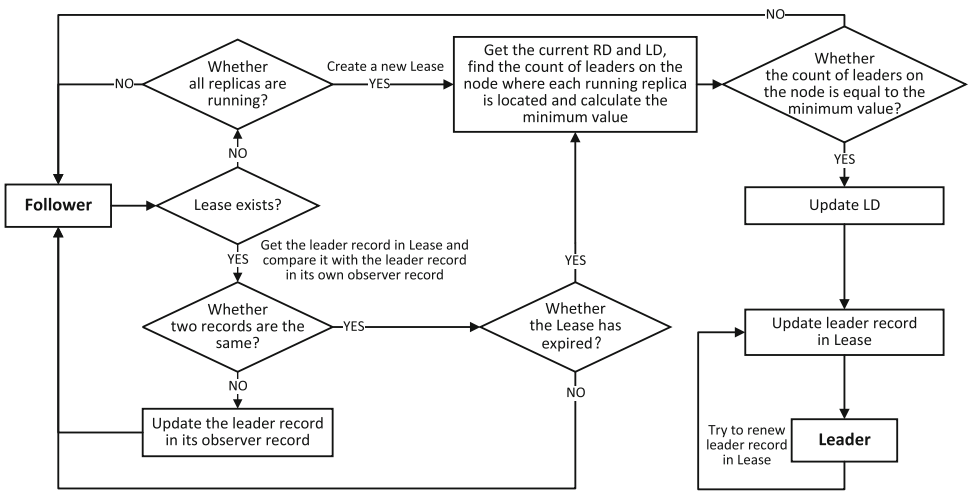

Fig. 2 Flowchart of the BRD-BLE algorithm

throughput, we established an experimental Kubernetes cluster on an ARM-based Kunpeng server [28]. The cluster comprised one master node and five worker nodes, with the master node equipped with 16 CPU cores and 16 GB of RAM, while each worker node had 8 CPU cores and 8 GB of RAM. The versions of Kubernetes and Docker employed were 1.23.4 and 20.10.9, respectively. Multiple stateful applications [29] were deployed in the cluster, enabling external access through NodePort services. The Hey benchmarking tool [30] was utilized to simulate diverse request volumes from clients to each application, facilitating the evaluation of the cluster's performance.

4.1 Distribution of leaders in multiple applications

To assess the leader distribution balance of the two algorithms in scenarios involving multiple stateful applications, we deployed varying numbers of stateful applications in the aforementioned cluster environment, ranging from 5 to 30. Each application was equipped with 3 replicas to improve reliability and scalability. These applications implemented a leader-based consistency maintenance mechanism to ensure data consistency among the replicas. In every stateful application, two containers are established within each replica. The main container operates a straightforward web server, tasked with processing user requests and generating responses. The secondary container, known as the leader election container, incorporates either the KUBE-LE or BRD-BLE algorithm. This container's role is to conduct leader elections among the replicas of stateful services. Furthermore, we allocate Persistent Volume (PV) storage volumes to each replica for data state preservation. This configuration emulates a microservices architecture scenario where multiple services collaboratively deliver comprehensive functionality. We conducted 100 deployment experiments for different application quantities, recording the leader count on each worker node after each deployment, and derived the average as the final result.

To visually observe the effect of leader distribution for the two algorithms with different numbers of applications, we categorized the number of leaders on each worker node into five levels, ranging from low to high. Figure 3 illustrates the leader distribution for both algorithms in the range of 5–30 applications. It is evident that when using the KUBE-LE algorithm, significant disparities exist in the number of leaders among the worker nodes. Some nodes have few leaders, while others have a large concentration. This imbalance worsens as more applications are added. In contrast, when using the BRD-BLE algorithm, the number of leaders on each worker node is relatively similar, and there is no apparent skewness. Furthermore, this balance remains consistent as more applications are added.

To quantitatively assess the disparity in leader distribution balance between the two algorithms, we employed the standard deviation as a metric to gauge the spread of the number of leaders among worker nodes. A smaller standard deviation implies a more equitable distribution of leaders, while a larger standard deviation signifies a more concentrated allocation of leaders. We calculated the standard deviation of Leader Distribution (LD) for each deployment and derived the average value as the outcome. Figure 4 depicts the average standard deviation of LD for both algorithms across 5–30 applications. The graph reveals that when employing the BRD-BLE algorithm, the average standard deviation of LD consistently remains below 0.25, indicating a relatively uniform distribution. In contrast, when utilizing the KUBE-LE algorithm, the average standard deviation of LD is notably larger, surpassing 0.76. With the deployment of 30 applications, the average standard deviation of LD escalates to 1.91. This suggests that the KUBE-LE algorithm tends to concentrate leaders on specific nodes, whereas the BRD-BLE algorithm results in a more balanced distribution of leaders across nodes.

To further illustrate the distribution of leaders in extreme scenarios for both algorithms, we selected the least balanced leader distribution from the 100 deployment experiments and presented it in Table 1. The table reveals significant discrepancies in the number of leaders among the worker nodes when employing the KUBE-LE algorithm. Certain nodes exhibit an excessive or insufficient number of leaders, with one node concentrating nearly half of the leaders. This imbalance not only subjects the node to excessive load pressure but also heightens the risk of failure, thereby compromising the system's performance and reliability. Conversely, the utilization of the BRD-BLE algorithm results in a more equitable distribution of leaders across nodes. Regardless of the number of deployed applications, the number of leaders on each node fluctuates around the mean, with minimal disparity between the maximum and minimum values. This observation underscores the effectiveness of the BRD-BLE algorithm in mitigating an excessive concentration of leaders on specific nodes, facilitating a balanced distribution of leaders among worker nodes in scenarios involving multiple stateful applications.

4.2 The influence of leader distribution on node resources

To assess the influence of leader distribution on node resource utilization, we deployed 15 stateful applications using both the KUBE-LE algorithm and the BRD-BLE

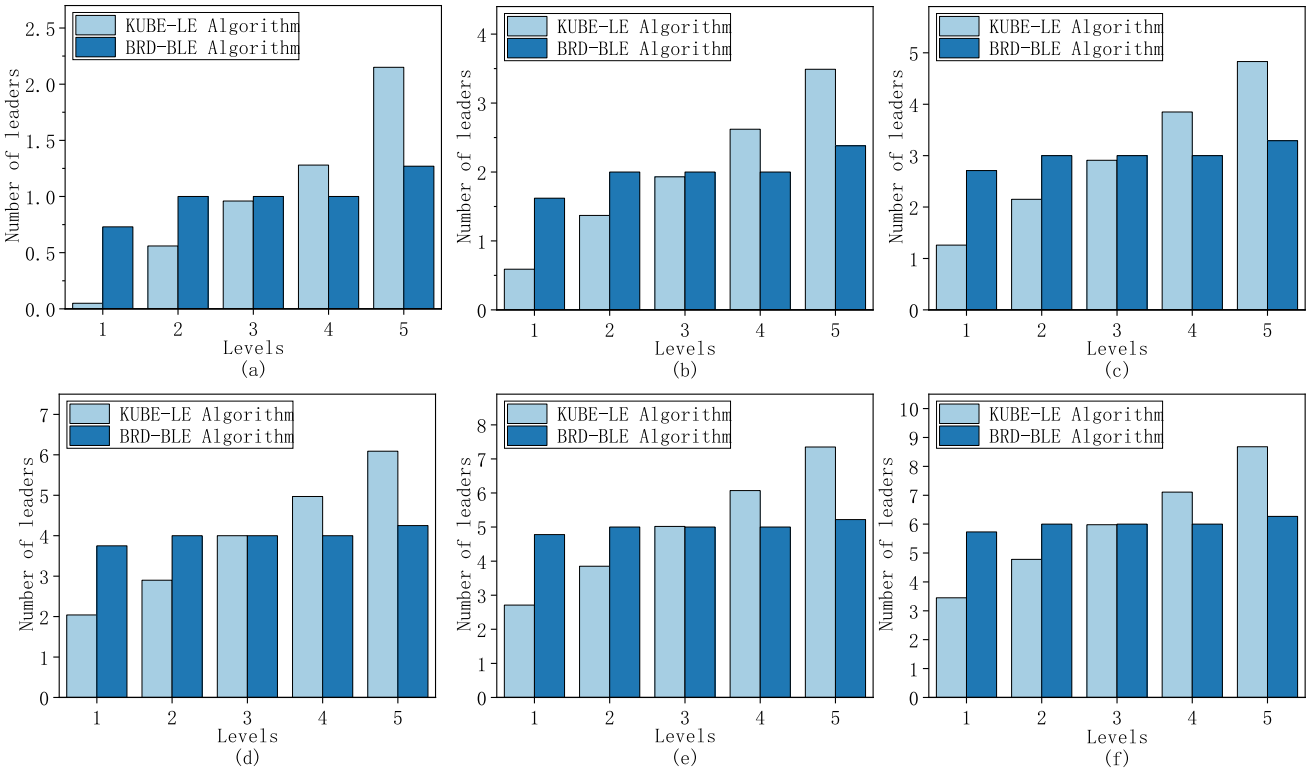

Fig. 3 Distribution of different numbers of leaders among worker nodes: **a** 5 applications, **b** 10 applications, **c** 15 applications, **d** 20 applications, **e** 25 applications, **f** 30 applications

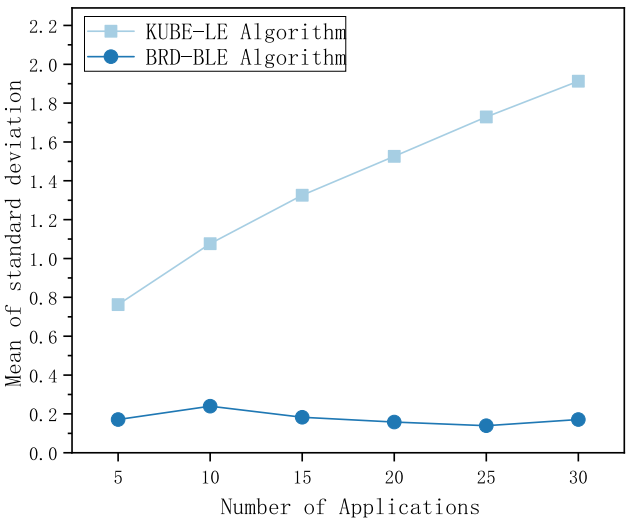

Fig. 4 Standard deviation of leader distribution

algorithm in the previously mentioned cluster environment. Among the 100 deployment experiments, we selected the outcome displaying the least balanced leader distribution, which is presented in Table 2. Subsequently, we conducted 10 rounds of load testing in which a single client continuously sent write requests to each application for 60 s. The requests were evenly distributed across the replicas of each application using the IPVS proxy mode [31]. Following each experiment, we recorded the CPU utilization of every worker node and computed the average value as the final outcome.

To compare the differences in node resource utilization between the two algorithms, we utilized Fig. 5 to depict the average CPU utilization of each worker node during the simultaneous write requests from a single client to all 15 applications. The graph reveals a substantial discrepancy in CPU utilization among the worker nodes when employing the KUBE-LE algorithm. Node1, lacking any leaders, exhibits a meager average CPU utilization of 6.52%, indicating its idle state. Conversely, Node5, responsible for seven leader roles, experiences a significantly high average CPU utilization of 81.71%, signifying the node's heavy load pressure, potentially impeding the performance of other services on that node. These findings suggest that the KUBE-LE algorithm's uneven distribution of leaders leads to excessive or insufficient utilization of certain nodes, resulting in an imbalance in the cluster's workload. In contrast, when utilizing the BRD-BLE algorithm, CPU utilization among the worker nodes demonstrates better balance. Node2, Node3, and Node4 each handle three leader roles, maintaining an average CPU utilization of approximately 52%. This indicates that these nodes effectively shoulder a considerable portion of the workload without experiencing overload or idleness. Consequently,

Table 1 The least balanced leader distribution result for the different numbers of applications

Number of applications	KUBE-LE algorithm						BRD-BLE algorithm					
	5	10	15	20	25	30	5	10	15	20	25	30
Node1	0	0	0	1	2	2	0	1	2	3	4	5
Node2	0	0	1	1	4	4	1	2	3	4	5	6
Node3	1	1	3	4	4	5	1	2	3	4	5	6
Node4	1	4	4	5	4	5	1	2	3	4	5	6
Node5	3	5	7	9	11	14	2	3	4	5	6	7

Table 2 The least balanced leader distribution result for the 15 applications

Worker node	Node1	Node2	Node3	Node4	Node5
KUBE-LE algorithm	0	1	3	4	7
BRD-BLE algorithm	2	3	3	3	4

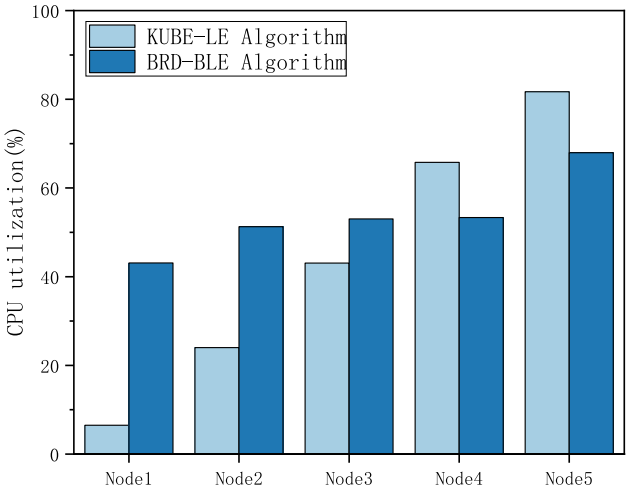

Fig. 5 Average CPU utilization of each worker node when the client sends write requests

the BRD-BLE algorithm's equitable distribution of leaders ensures a more reasonable resource utilization across the cluster, enhancing its load balancing capability.

4.3 The influence of leader distribution on throughput

To assess the influence of leader distribution on system throughput, we conducted experiments using the deployment result that exhibited the least balanced leader distribution among the 15 applications when both algorithms were employed. Following this, we dispatched 20,000 write requests to each application, employing varying

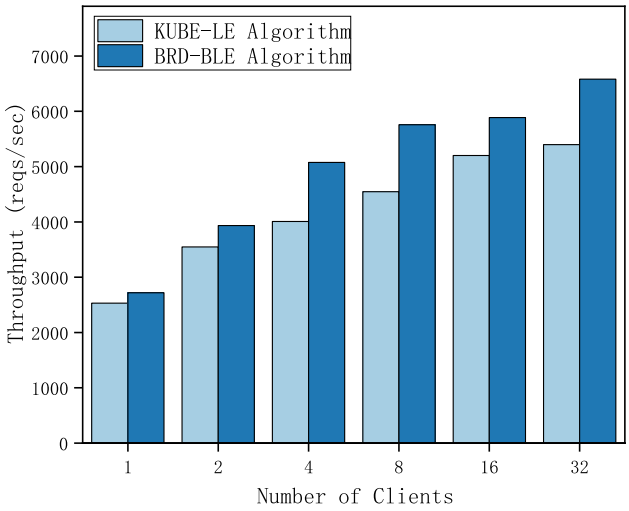

Fig. 6 Cumulative throughput of the system as different numbers of clients send write requests

numbers of clients (ranging from 1 to 32), and recorded the system's cumulative throughput.

Figure 6 illustrates the cumulative throughput of the system as different numbers of clients simultaneously sent write requests to the 15 applications. The graph reveals that the employment of the KUBE-LE algorithm results in a relatively low cumulative throughput. This can be attributed to one node, which concentrates nearly half of the leaders, experiencing an elevated burden of write requests and data synchronization tasks. Consequently, this node becomes overloaded, leading to increased response time and diminished processing capacity. Conversely, when utilizing the BRD-BLE algorithm, the cumulative throughput of the system exhibits significant improvement. For instance, with a single client, the cumulative throughput increases by approximately 7.41%, and with 32 clients, it increases by approximately 21.94%. These findings indicate that the more evenly distributed leaders among the nodes, as facilitated by the BRD-BLE algorithm, enable each node to effectively exploit its resources for request processing, thereby bolstering the system's processing capacity and response speed.

5 Conclusion

This paper addresses the issue of uneven leader distribution among multiple stateful applications in Kubernetes clusters and introduces the BRD-BLE algorithm. The proposed algorithm utilizes the Kubernetes API to gather replica distribution information and the count of leaders on each worker node for every application, facilitating the selection of an appropriate replica as the leader. By effectively balancing the number of leader replicas across worker nodes, this algorithm mitigates single-point failures and load imbalance that may result from leader concentration or scarcity. To evaluate the algorithm, we deployed a Kubernetes cluster on an ARM-based Kunpeng server and conducted experiments using varying numbers of stateful applications. The experimental results demonstrate the algorithm's accurate execution of leader election and its significant improvement in resource utilization and system throughput compared to the KUBE-LE algorithm. Overall, this algorithm offers essential assurance for maintaining high service quality in stateful applications within Kubernetes clusters.

Acknowledgements This work is funded by 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.

Author contributions J.L. carries out the conception of experimental ideas and the formulation or evolution of overall research goals and objectives Method design and create models. Y.D. conducted experiments and wrote the main manuscript texts, images and table productions. Y.L. is involved in the analysis of the data and the production of the images. All the authors reviewed the manuscript.

Funding This work was financially supported by Research on Key technologies of resource scheduling and service High Availability based on ARM architecture, Project No. 232102210199.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

1. Wu, Z.: Development and trends of virtualization technology in cloud computing. J. Comput. Appl. **37**(4), 915–923 (2017). https://doi.org/10.11772/j.issn.1001-9081.2017.04.0915
2. Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A.F.: Performance evaluation of container-based virtualization for high performance computing environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 233–240 (2013). https://doi.org/10.1109/PDP.2013.41
3. Bernstein, D.: Containers and cloud: from LXC to docker to Kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014). https://doi.org/10.1109/MCC.2014.51
4. Kubernetes Documentation. [Online]. https://kubernetes.io/docs/home/. Accessed 19 Jan 2023
5. Jiao, Q., Xu, B., Fan, Y.: Design of cloud native application architecture based on Kubernetes. In: 2021 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), pp. 494–499 (2021). https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech52372.2021.00088
6. Hu, T., Wang, Y.: A Kubernetes autoscaler based on pod replicas prediction. In: 2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS), pp. 238–241 (2021). https://doi.org/10.1109/ACCTCS52002.2021.00053
7. Vayghan, L.A., Saied, M.A., Toeroe, M., Khendek, F.: A Kubernetes controller for managing the availability of elastic microservice based stateful applications. J. Syst. Softw. **175**, 110924 (2021). https://doi.org/10.1016/j.jss.2021.110924
8. Nguyen, N., Kim, T.: Toward highly scalable load balancing in Kubernetes clusters. IEEE Commun. Mag. **58**(7), 78–83 (2020). https://doi.org/10.1109/MCOM.001.1900660
9. Simple Leader Election with Kubernetes and Docker. [Online]. https://kubernetes.io/blog/2016/01/simple-leader-election-with-kubernetes/. Accessed 23 Jan 2023
10. Takahashi, K., Aida, K., Tanjo, T., Sun, I.: A portable load balancer for Kubernetes cluster. In: HPC Asia 2018: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, pp. 222–231 (2018). https://doi.org/10.1145/3149457.3149473
11. Liu, Q., Haihong, E., Song, M.: The design of multi-metric load balancer for Kubernetes. In: 2020 International Conference on Inventive Computation Technologies (ICICT), pp. 1114–1117 (2020). https://doi.org/10.1109/ICICT48043.2020.9112373
12. Dua, A., Randive, S., Agarwal, A., Kumar, N.: Efficient load balancing to serve heterogeneous requests in clustered systems using Kubernetes. In: 2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC), pp. 1–2 (2020). https://doi.org/10.1109/CCNC46108.2020.9045136
13. Lee, J.-B., Yoo, T.-H., Lee, E.-H., Hwang, B.-H., Ahn, S.-W., Cho, C.-H.: High-performance software load balancer for cloud-native architecture. IEEE Access **9**, 123704–123716 (2021). https://doi.org/10.1109/ACCESS.2021.3108801
14. Wang, Q., Ren, Y., Yang, S., Guan, J., Li, B., Zhang, J., Tan, Y.: ProxyDWRR: a dynamic load balancing approach for heterogeneous-CPU Kubernetes Clusters. In: 2022 IEEE International Conference on Joint Cloud Computing (JCC), pp. 65–72 (2022). https://doi.org/10.1109/JCC56315.2022.00017
15. Botez, R., Petruti, C.-M., Ivanciu, I.-A., Dobrota, V.: Kubernetes-based load balancer as a service for private cloud infrastructures. In: 2022 14th International Conference on Communications (COMM), pp. 1–6 (2022). https://doi.org/10.1109/COMM54429.2022.9817323
16. Baresi, L., Hu, D.Y.X., Quattrocchi, G., Terracciano, L.: KOSMOS: vertical and horizontal resource autoscaling for Kubernetes. Serv. Orient. Comput. ICSOC **2021**, 821–829 (2021). https://doi.org/10.1007/978-3-030-91431-8_59
17. Pramesti, A.A., Kistijantoro, A.I.: Autoscaling based on response time prediction for microservice application in Kubernetes. In: 2022 9th International Conference on Advanced Informatics:

Concepts, Theory and Applications (ICAICTA), pp. 1–6 (2022). https://doi.org/10.1109/ICAICTA56449.2022.9932943
18. Kim, T., Choi, S.-G., Myung, J., Lim, C.-G.: Load balancing on distributed datastore in opendaylight SDN controller cluster. In: 2017 IEEE Conference on Network Softwarization (NetSoft), pp. 1–3 (2017). https://doi.org/10.1109/NETSOFT.2017.8004238
19. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/3335772.3335939
20. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, pp. 305–320 (2014). https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
21. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.C.: ZooKeeper: wait-free coordination for Internet-scale systems. In: USENIX Annual Technical Conference (2010). https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf
22. Oliveira, C., Lung, L.C., Netto, H., Rech, L.: Evaluating Raft in Docker on Kubernetes. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-48944-5_12
23. Netto, H., Oliveira, C.P., Oliveira Rech, L., Alchieri, E.: Incorporating the raft consensus protocol in containers managed by Kubernetes: an evaluation. Int. J. Parallel Emerg. Distrib. Syst. **35**(4), 433–453 (2020). https://doi.org/10.1080/17445760.2019.1608989
24. Netto, H., Oliveira, C.P., Oliveira Rech, L., Alchieri, E.: State machine replication in containers managed by Kubernetes. J. Syst. Archit. **73**, 53–59 (2017). https://doi.org/10.1016/j.sysarc.2016.12.007
25. Netto, H.V., Luiz, A.F., Correia, M., Oliveira Rech, L., Oliveira, C.P.: Koordinator: a service approach for replicating docker containers in Kubernetes. In: 2018 IEEE Symposium on Computers and Communications (ISCC), pp. 58–63 (2018). https://doi.org/10.1109/ISCC.2018.8538452
26. Abdollahi Vayghan, L., Saied, M.A., Toeroe, M., Khendek, F.: Microservice based architecture: towards high-availability for stateful applications with Kubernetes. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), pp. 176–185 (2019). https://doi.org/10.1109/QRS.2019.00034
27. SelectorSpreadPriority. [Online]. https://kubernetes.io/docs/reference/scheduling/policies/#priorities. Accessed 23 Jan 2023
28. Kunpeng server based on arm architecture. [Online]. https://e.huawei.com/cn/products/computing/kunpeng. Accessed 19 Oct 2022
29. Stateful Applications. [Online]. https://kubernetes.io/docs/tutorials/stateful-application. Accessed 19 Feb 2023
30. Hey. A tiny program sends some load to a web application. [Online]. https://github.com/rakyll/hey. Accessed 10 Apr 2023
31. IPVS-based intra-cluster load balancing. [Online]. https://kubernetes.io/zh-cn/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive. Accessed 26 Feb 2023

Junnan Liu (1980—), male, Master of Engineering, associate professor, currently works at the School of Software, Henan University. The main research directions are cloud computing, wireless sensor networks, graphics and image processing, and data analysis.

Yongkang Ding (1995—), male, is currently studying for a master's degree in software engineering at the School of Software, Henan University. The main research directions include load balancing strategies for Kubernetes clusters and cloud computing.

Yifan Liu (1997—), male, is currently studying for a master's degree in software engineering at the School of Software, Henan University. The main research directions include task scheduling strategies for heterogeneous distributed systems and high-performance computing.