



Full length article

Improving scalability, energy efficiency, and cost-effectiveness in Kubernetes clusters using a nonlinear regression-based predictive replica model and ORLE algorithm



Indrani Vasireddy*, Rajeev Wankar, Raghavendra Rao Chillarige

School of Computer and Information Sciences, University of Hyderabad, Hyderabad, India

ARTICLE INFO

Keywords:

Kubernetes
Nonlinear regression
Predictive replica model
Optimal replica leader election (ORLE)
Load balancing
Leader election
Scalability
Native to the cloud
Resource utilization
Throughput
Latency
Stateful applications

ABSTRACT

Container-based deployments have transformed how modern applications are packaged, deployed, and scaled. They bring increasing benefits in terms of development agility, testing, and collaboration. Kubernetes is a famous container orchestration engine that manages the life cycle of containerized applications by automatically scaling the containers and load balancing among them. In this paper, we present an application-level leader election method known as ORLE(Optimal Replica Leader Election). After conducting a detailed performance analysis of ORLE, we also developed a nonlinear regression Predictive Replica Model that predicts the throughput in real time, which helps in the early identification of conditions that require replicas' scaling up or down. We integrated the proposed ORLE with this nonlinear regression Predictive Replica Model to improve the performance of Kubernetes clusters. Our model autoscales replicas concerning real-time traffic measurements to maximize resource utilization and system scalability. ORLE selects the most suitable replica as the leader to optimize the load distribution and reduce the overall latency of the request. Our experimental results show that our proposed solution outperforms the original leader election mechanism (Default RAFT) of Kubernetes and an existing state-of-the-art algorithm Balanced Leader Distribution (BLD) by up to 25% throughput improvement, 40% latency reduction, optimization of energy consumed, and cost efficiency per working replica under real-time conditions. The proposed model is more beneficial for stateful applications and microservices architectures since consistent performance and fast leader election are crucial in ensuring the Kubernetes cluster reliability and performance and are highly suitable for Kubernetes clusters deployed in a cloud computing environment, which demands high scalability, low latency, and efficient resource management.

1. Introduction

Kubernetes is an orchestration tool for containers that is used to deploy, schedule, and manage applications. Containers isolate applications and their dependencies from interacting with other modules, making them compatible with addressing various applications. They offer benefits such as effective resource utilization, faster development, and scaling [1]. Docker is a platform that allows users to develop, deploy, and run applications in isolated environments called containers [2]. Containers are one form of virtualization that bundles an application and all its dependent resources as one package to ensure its stable behavior in all its environments [3]. The file used to run the program within a Docker environment is referred to as a Docker image. Containers are created from docker images.

It is an open-source container orchestration tool to automate container deployment, management, scaling, and networking [4]. In Kubernetes, the smallest executable unit of an application running within a cluster is known as a pod [5,6]. A pod contains one or more tightly coupled containers and shares the same network name, space, storage, and resources [7,8]. In Kubernetes, replicas are regarded as an instance or copy of a Kubernetes pod. Replicas ensure high availability, load distribution, and scalability in our applications. Replicates and deployments are commonly used to manage replicas. A replica set is a Kubernetes component that ensures that there is always a stable set of running pods for a specific workload. Kubernetes architecture includes other resources like stateful sets and daemon sets for specific purposes involving stateful applications and running one pod per node. In Kubernetes, a replica controller allows the creation of multiple

* Corresponding author.

E-mail address: karuna.indu@gmail.com (I. Vasireddy).

pods easily and ensures that the desired number of pods is always maintained.

In Kubernetes, leader election occurs at both the system and the application levels. System components like kube-controller-manager and kube-scheduler elect a single active instance to perform operations. They use the Kubernetes API server with Lease or ConfigMap objects for coordination, which is backed by *etcd*. *etcd* uses the Raft consensus algorithm to maintain consistency and elect its own leader among cluster nodes. At the application level, custom controllers or operators use lease-based coordination mechanism or Endpoints to elect a leader among replicas. These applications may implement basic election logic, such as lease-based coordination or the Bully algorithm (a classic leader election algorithm in distributed systems but not directly used in Kubernetes). In all cases, the components interact only with the API server and not with *etcd* directly for leader coordination.

Raft is a leader-based consensus algorithm used internally by *etcd*, the backing store for the Kubernetes control plane, where a single node handles log replication and state changes, and if that leader fails, the remaining nodes elect a new leader based on timed heartbeats and election timeouts. The default Raft leader election algorithm in Kubernetes ensures that only one replica of a pod becomes the leader, while the other replicas act as followers [9].

Although application-based leader election ensures that only one replica holds the leader role at any given time, it does not prevent the pod from killing. If the leader pod crashes, a new leader must be elected, and the process of electing a new leader among the remaining replicas is started. Thus, the default algorithm maintains the continuity of leadership even when a pod is killed for any reason. The load balancing of pods in Kubernetes involves distributing network traffic among replicas of applications to achieve scalability and high availability [10]. This is done using different load-balancing strategies to ensure that user requests are handled by different replicas in a way that optimizes resource usage and prevents overloading of any single node because it is the responsibility of the leader pod to handle all data update requests, as the leader pod will have a heavier load compared to follower replicas [11]. Hence, in the Kubernetes cluster, the leader pod has the overhead of additional processing and communication tasks. In Kubernetes cluster, where stateful applications are hosted, efficient leader election algorithms play a crucial role in maintaining system reliability and efficient performance. Although the default algorithm maintains leadership continuity, it does not adapt well to changing workloads or resource demands. However, there are several crucial drawbacks in the current practice of leader election and auto-scaling of Kubernetes clusters. Current leader election mechanisms fail to estimate real-time resource usage and throughput and, thus, may select inadequate leaders who would not handle essential tasks efficiently. Moreover, standard autoscaling methods such as the HPA work in a reactive way, increasing resources only after utilization crosses the threshold, which results in slow scaling and even a decline of performance during periods of high traffic. Traditional approaches do not forecast traffic rates and can scale replicas without considering the election of leaders, making resource distribution uneven.

In this paper, we propose a method in which we employ the ORLE algorithm along with a Predictive Replica Model, based on non-linear regression. The ORLE algorithm chooses one replica where the read/write requests are processed depending on real time factors (since the collected network traffic values are supplied to the algorithm to select a proper leader, we use the term real time here) like throughput rate and resource utilization. This also allows the cluster scheduler to be sufficiently equipped to handle the current load with little delay. However, the nonlinear regression-based predictive replica model estimates the number of replicas required to satisfy the throughput rate needed, thus providing the system with dynamic scaling based on real-time performance needs. By predicting workload demands through non-linear regression and varying the number of replicas with respect to the processed throughput in current and future time, this model

optimizes energy consumption and costs for Kubernetes clusters. While incorporating the Predictive Replica Model with ORLE, we also enhance leader election and the utilization of available resources in line with future workload into increased availability, scalability and performance. Our experiments show that the proposed solution is efficient, proving that the optimized traffic flow, faster response time, less energy consumption, and overall operating expenses are optimized in the case of managing the Kubernetes cluster compared to the previous solutions. Through the estimation of the number of replicas by using the proposed nonlinear regression-based Predictive Replica Model, the system guarantees that replicas are scaled based on the throughput that is needed to serve the traffic while avoiding unnecessary replication and incorporation of additional resources, and thus cuts down unnecessary cost. This approach is most helpful for stateful applications or applications with microservices architecture since they require quick leader election and stable consistent performance to support the proper functioning of state-based services.

In this paper, we develop a nonlinear regression based Predictive Replica Model, which coordinates resource provisioning, energy consumption and leadership choices in Kubernetes clusters. Unlike previous approaches where the replica scale is fixed in certain time periods, our model predicts traffic density and scales replicas with the throughput needed to avoid misuse of resources and lack of resources. The ORLE algorithm finds an optimal replica for load control that also increases throughput and decreases latency and is cost-effective compared to prior approaches. This paper is organized as follows. Section 2 presents the related work, Section 3 presents the ORLE algorithm, Section 4 presents Nonlinear Regression-Based Predictive Replica Model from ORLE, Section 5 presents performance evaluation, and Section 6 presents conclusions.

2. Related research

The leader election algorithm plays an important role in load balancing in Kubernetes by ensuring that only one replica of a pod within a group of pods (replicas) is elected as a leader pod for serving requests at any given time, while other replicas remain as followers. This ensures consistency among pod replicas and prevents data duplication in distributed systems, such as Kubernetes. Although Kubernetes does not have an inbuilt leader-election mechanism, we can implement an external leader-election mechanism. While it is possible to implement leader election from scratch using consensus algorithms like raft, deploying additional software such as zookeeper, etc. or consul to manage the consensus process can be complex and may introduce operational overhead. The default leader election algorithm of Kubernetes uses mechanisms, such as the lease object and the leader elector utility, for leader election within its control plane components. The leader election concept is not part of Kubernetes itself, but is implemented by controllers or applications running within Kubernetes clusters. Kubernetes provides a platform and resources for implementing a leader election algorithm. Leader elector utility is a Kubernetes-local mechanism designed to simplify the implementation of leader election in distributed applications running on Kubernetes. These replicas are part of a Kubernetes service that exposes pods as endpoints. Every Kubernetes object has a unique resource version assigned to it by an API server. These resource versions are incremented by each update of the object. Each replica reads the resource version of the service endpoint object. Kubernetes objects can be annotated with key-value pairs using annotations. In the context of leader elections, annotations, such as the pod name, can be used to store information about the current leader. Each replica reads the current resource version of the service endpoint object and the current leader annotation value. If the current pod sees that the leader annotation is empty or matches its own pod name, it attempts to update the leader annotation with its own pod name using a compare-and-swap operation.

Recently, numerous discussions and studies have been conducted on pod's in Kubernetes. Many researchers have discussed different load-balancing strategies and pod migration techniques in Kubernetes. These studies can help ensure the balance of load between nodes and reduce cluster downtime. According to Kim et al. [12] they introduced an advanced Kube proxy that is named 'resource adaptive proxy', which aims to minimize traffic routing. The main concept of a resource-adaptive proxy is to efficiently direct a majority of client requests to pods situated on the local node by reducing the delay in the process of requests. According to Q. Nguyen et al. [13], in a KubeEdge-based edge computing environment, a local scheduling scheme was proposed to handle user requests directly at the local node; therefore, we do not need to forward traffic to remote nodes. Load balancers are required in a wide range of tasks, starting from the spanning edge for computing setups in data center clusters [14]. Load balancers ensure an even distribution of applications on an individual node. To enable auto-scaling capabilities, Kubernetes constantly monitors various components, including pods, applications, host machines, and metrics of a cluster collectively referred to as 'metrics' [15]. Auto-scalers are activated on the basis of the response of specific threshold levels within these metrics. Although Kubernetes essentially offers resource metrics, their current study is restricted to monitor the CPU and memory utilization of both pods and host machines.

In an effort to fill the gap between having more or less leaders in stateful applications in Kubernetes clusters, they proposed the BRD-BLE algorithm [16]. This new algorithm uses the Kubernetes API data on replica distribution and the number of leaders assigned for all applications on each worker node. This information is used in order to choose appropriate replicas as leaders and distribute leader replicas through the worker nodes. The algorithm also solves problems of load imbalance and power-law distributions, as the leader concentration or the lack of leader problem is eliminated. However, the focus of our proposed work is selecting the optimal replica as the leader, which is cost-effective. In their work Yongkang Ding et al. [17], proposed the Min-Leader optimal scheduling algorithm associated with the problem of improper scheduling of leaders for stateful applications among nodes by the Kubernetes scheduler. They proposed an algorithm to address the issue when the leaders are distributed unevenly among nodes and all the stateful application leaders will be in one place in the cluster. Lucas et al. [18] proposed a distributed coordination mechanism to control the movement of the replica, which is modeled in this work as a Kubernetes deployment. The ensemble of coordinators is chosen by an election algorithm acting with the help of Apache ZooKeeper, selecting a temporary coordinator for each movement of the target. Slightly more time was spent evaluating and refining the election algorithm, which is a customizable and tunable way to steer the probabilistic electoral component to a specified extent while guiding the election result according to certain heuristic values. By default, Kubernetes uses a basic method of leader election that enables only one replica to perform complex processes, while other replicas act in parallel as API followers or replicas. This process involves distributed consensus mechanisms that are present in the distributed setup like a raft. The Kubernetes lease object is used in the coordination of API to announce and maintain leadership. Pod leaders are chosen by means of a time-based election scheme when the first pod that updates the lease for the key becomes the leader. Although this is easy to implement and works fine in smaller clusters, it has no elasticity, that is, it does not scale with the work load or availability of resources.

Nguyen et al. proposed a BLD algorithm [19]. As the default leader election algorithm does not focus on the location of the leaders' election, it may result in a leader concentration issue on a particular node, which could cause the system's performance to significantly decline if one node has all of the replicas as leaders. Hence, the BLD algorithm improves upon previous methodologies by considering the distribution of leaders across nodes to achieve a more balanced leadership structure within the Kubernetes cluster. Although the Balanced Leader

Distribution (BLD) algorithm enhances the leader election process in Kubernetes by promoting a more equitable distribution of leadership across nodes, it does not address the efficiency of replicas. The BLD algorithm may lead to unnecessary leader transitions, as followers may repeatedly check and contest leadership based on a potentially outdated view of the leader distribution. This can introduce latency and instability within the system, particularly in environments with rapid changes in resource demands.

In this work, we proposed an Optimal Replica Leader Election algorithm for stateful applications in Kubernetes clusters by selecting the fastest replica based on throughput and resource utilization. This research follows these prior works and extends them by developing a Nonlinear Regression based Predictive Replica Model that predicts the number of replicas desired over time given network traffic. Unlike BLD, the Optimal Replica Leader Election (ORLE) algorithm selects the fastest replica depending on the throughput and resource utilization, thus providing balanced load distribution and relatively low time wasted in making leader-node determinations. Overall, the proposed model based on the combination of the predictive scaling and optimized leader election is faster than existing solutions. Our proposed algorithm collects real time traffic and predicts service demand to adjust the replica number. The leader election process is complex and the traditional Kubernetes ensures that only one replica is leader at a time, but it may reduce the performance. The ORLE algorithm uses the metric such as throughput, resource utilization to elect a leader. The main concept of our solution is the enhancement of the resource predicting approach with an optimized leader election protocol, which offers better performance to existing techniques. This enables us to control the resources optimally, as well as control the number of replicas based on the throughput actually needed via the integration of ORLE with the Predictive Replica Model based on the Nonlinear Regression. This way of predictive scaling guarantees that the system always has a right number of replicas while using as few resources as possible at any particular time. But our solution is not limited to superior scalability only, it even makes the system more energy efficient as well as cost effective in comparison to the existing system where resources are allocated dynamically based on the previous recorded demands. The integration of these make ORLE a very efficient solution for Kubernetes cluster that require high performance, scalability in addition to efficient resource management especially where the workload varies.

3. Optimal replica leader election algorithm

The procedure for the proposed Optimal Replica Leader Election (ORLE) algorithm is presented in Fig. 1, which has been designed to elect the leader in a Kubernetes cluster effectively. Unlike the Kubernetes default algorithm, ORLE selects the fastest replica in the leader election process and this process is integrated directly into our application code. This process utilizes Kubernetes client libraries to interact with the Kubernetes API for coordination and the election of a leader. This integration not only enhances the efficiency of the election process but also ensures that the selection of the leader is based on real-time performance metrics. ORLE's dynamic selection of the fastest replica minimizes the risk of overloading specific nodes and allows for better adaptability to changing resource demands. Moreover, the default Kubernetes leader election algorithm does not consider the operational context of replicas, potentially resulting in suboptimal leader placements. ORLE addresses this challenge by continuously evaluating the current state of replicas, ensuring that leadership is assigned to the most efficient replica available at any given moment. By adopting the ORLE algorithm, we enhance the robustness, scalability, and performance of our Kubernetes clusters, ultimately improving the overall reliability of distributed applications.

In the context of Kubernetes clusters that host stateful applications, an efficient leader election algorithm plays a crucial role in maintaining system reliability, high availability, and performance. Fig. 2

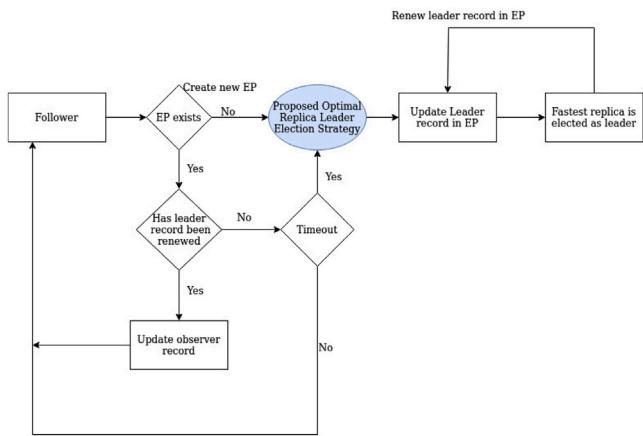


Fig. 1. Proposed modification in leader election process of default algorithm.

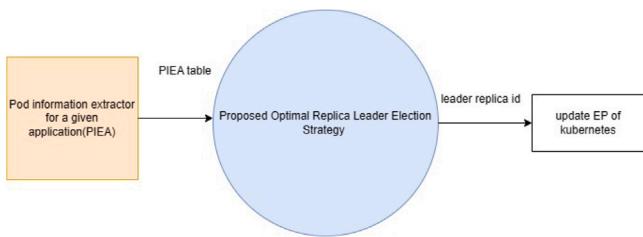


Fig. 2. Proposed procedure for optimal replica leader election algorithm.

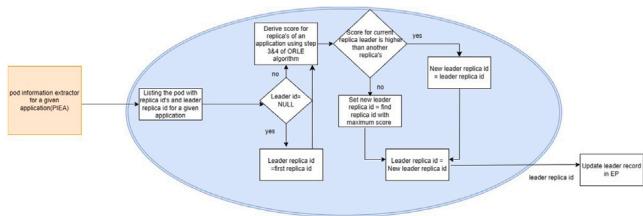


Fig. 3. Process for optimal replica leader election.

depicts an approach for an optimized replica leader election process patch where the fastest replica is selected based on throughput and resource utilization metrics. The proposed algorithm aims to optimize the leader distribution across replicas, thus improving load balance, performance, and availability. Through an organized design process, the algorithm dynamically selects the fastest replica as the leader to ensure efficient low latency, high throughput, energy efficient, and cost-effective. The ORLE algorithm selects the fastest replica based on a combination of two metrics, i.e., throughput and resource utilization. To identify the fastest replica, we measure each replica's throughput and record the time taken to process a particular application. However, the performance of the fastest replica is contingent upon the availability of the node's resources as well as its throughput. In addition, we collect resource utilization metrics, such as CPU and memory usage, for each replica. Taking into account the above two metrics, each replica is assigned a score. The replica with the highest score achieved through optimal throughput and minimum resource utilization is elected as the leader. This approach ensures that the fastest replica is elected as the leader and that the fastest replica handles the task efficiently within the cluster's resource limitations. With the integration of the ORLE algorithm with Kubernetes, we optimize the cluster's capability to dynamically manage leader election, efficiently respond to changing workloads, and optimize the utilization of resources.

```

1 def measure_throughput(replica):
2     start_time = time.time()
3     # Send workload to the replica and wait for completion
4     # For simplicity, let's assume the workload takes a certain amount of time
5     time.sleep(2)
6     end_time = time.time()
7     throughput = 1 / (end_time - start_time) # Requests per second
8     return throughput
9
10 replicas = ["replica-1", "replica-2", "replica-3"]
11
12 for replica in replicas:
13     throughput = measure_throughput(replica)
14     print(f"Throughput for {replica}: {throughput} requests/second")
  
```

Fig. 4. Throughput measurement of replicas.

In Fig. 3 the flow of the proposed algorithm for the leader election process is described. Initially, all replicas start in the follower state and the first replica is given as the leader replica. Then the replicas will calculate their score with the combination of two metrics as mentioned above, i.e., throughput and resource utilization of a particular replica. Periodically, each replica checks the scores of the other replicas. If any replica's score is higher than the others, it is designated as the candidate replica. The candidate replica initiates a leader election process and announces its leadership while other replicas acknowledge the leadership and compare their scores to the candidate's replica score. If a follower's replica score is lower than the candidate's replica score, it moves to a follower state, acknowledging the candidate as the leader. The elected leader sends "heartbeats" to maintain its leadership status. Replicas update their scores periodically and compare them with others. If a follower's replica score becomes higher than the leader's replica score, it initiates an election process to challenge the leader. The elected leader will periodically renew its leadership by sending "heartbeats" and updating the leader record in the EP. By combining throughput and resource utilization metrics, in selecting the fastest replica as the leader, we are selecting the leader based on both performance and efficiency.

To measure the performance of a replica in a Kubernetes cluster, we used Kubernetes monitoring solutions such as Prometheus and Grafana [20,21]. These tools help us gather and record various metrics related to the throughput of a replica in a Kubernetes environment (see Fig. 4).

To collect resource utilization metrics of a replica in a Kubernetes cluster, we used the Kubernetes API to get information about the pod's resource utilization, such as CPU usage and memory usage. In the proposed algorithm, in some cases, replicas may have equal processing capabilities and operate under the same conditions, leading to tie-breaking. However, due to the random delays introduced in the proposed algorithm, replicas will have slight variations in their execution times, leading to differences in throughput and latency values. The replica with the highest timestamp among all replicas at a given time will be declared the leader. If multiple replicas have the same highest timestamp, the first replica to update its timestamp will become the leader (see Fig. 5).

Using CPU and memory utilization as the focus, we obtain the performance characteristics of individual pods. The Kubernetes metrics server is a cluster-wide aggregator of resource usage data. It collects metrics such as CPU and memory usage from nodes and pods, which can be queried using kubectl top. The efficiency of the replica for candidate selection is based on metrics such as throughput and resource utilization. By reducing latency, ensuring low energy consumption, increasing throughput, and cost effectiveness, the proposed Optimal Replica Leader Election algorithm improves system performance as a

```

1     def get_resource_utilization(replica):
2         pod_name = replica
3         resource_utilization = v1.
4         read_namespaced_pod(pod_name, "default")
5             return resource_utilization
6
7         replicas = ["replica-1", "replica-2", "replica-3"]
8
9         for replica in replicas:
10            resource_utilization =
11            get_resource_utilization(replica)
12            print(f"Resource utilization for {replica}: {resource_utilization}")

```

Fig. 5. Resource utilization metrics of replicas.

whole. It reduces data transfer overhead and speeds up reaction times by carefully positioning replicas closer to consumers or computing units. In comparison, the default algorithm in Kubernetes implements a leader election based on a 'First-Come-First-Served' algorithm. The BLD algorithm proposed by Nguyen et al. ensures that the number of leaders at the candidate node does not exceed the specified threshold value [19]. BLD has limitations related to the frequency of leader updates and the precision of the Leader Management Election Process (LMEP). If leader information is not updated in a timely manner, it can lead to an imbalanced distribution, with some nodes potentially becoming overloaded while others remain underutilized. The Optimal Replica Leader Election (ORLE) algorithm helps to select the most optimal replica that will go ahead to act as a leader in Kubernetes and help to optimize resource utilization as well as energy consumption.

4. Nonlinear regression-based predictive replica model from ORLE

Most prior work aims at auto-scaling purely based on basic CPU and memory utilization threshold with the inability to incorporate a leader selection process or to forecast resource requirements based on performance parameters. In our work, we present a nonlinear regression-based predictive replica model based on the outcome of the ORLE and that gives a precise prediction of the number of replicas required to obtain the greatest performance in a cluster. This model is different from the other models and is self-adjustable in the number of replicas depending on the throughput and utilization data of the resources derived from ORLE. Based on these outputs, the model enables auto-scaling in Kubernetes clusters, ensuring that performance availability will never be an issue on the increasing workloads.

4.1. Nonlinear regression based predictive replica model

In this section, using ORLE performance data, we designed a nonlinear regression-based predictive replica model that determines the number of replicas needed according to metrics such as throughput, number of applications, or considerations of replicas. Analyzing the throughput and resource usage of the system after ORLE, we found that the traffic generated from the system during the leader election process could be used to accurately estimate traffic in the future and thus decide the number of replicas. This model predicts performance based on replica count and optimizes the resources before the overloading of nodes degrades the performance of the Kubernetes cluster.

$$Y = A \cdot P^B \cdot Q^C \cdot \exp(-(D \cdot P + E \cdot Q)) \quad (1)$$

Here, $(.)$ defined the simple multiplication operation, and the remaining parameters are defined as follows:

- **NA** It is the Number of applications
- **NR** It is the Number of Replica's
- $P = NA$

Table 1

Model parametric values.

Dep_Var	log(A)	log(B)	log(C)	D	E	AdjRSq	PValue
TP	4.640	0.078	0.063	0.033	-0.000	0.987	4.349E-10
KTP	4.617	-0.206	0.034	-0.010	-0.002	0.985	7.346E-10
LT	1.296	-1.861	0.561	-0.340	-0.002	0.945	5.469E-07
KLT	3.914	-0.250	-0.267	0.019	-0.009	0.422	0.047

- $Q = NR$

- **A**: A scaling factor that sets the overall magnitude of throughput or latency.
- **B**: A coefficient that shows how throughput or latency scales with the number of applications (NA).
- **C**: A coefficient that shows how throughput or latency scales with the number of replicas (NR).
- **D**: A coefficient that represents the sensitivity of the exponential decay with respect to the number of applications (NA).
- **E**: A coefficient that represents the sensitivity of the exponential decay with respect to the number of replicas (NR).

The Table 1 shows the model parameters, where the model parameter values are in logarithmic form of Eq. (1) above for throughput and latency of both the proposed ORLE algorithm and the default Kubernetes algorithm. Here, **AdjRSq** defines the adjusted square value of R. At the system level, given a specific snapshot of available nodes, this model allows us to predict throughput accurately. In addition, our goal is to use this model to predict the number of replicas.

4.2. Proposed algorithm

The Optimal Replica Leader Election (ORLE) with the nonlinear regression predictive model algorithm works to automatically scale in or scale out replicas in Kubernetes cluster to select the optimal leader and adjust the number of replicas in real time based on throughput and resource utilization. The system gathers information on CPU usage and throughput of all replicas in the cluster, which allows us to make necessary decisions in the context of leaders and scaling. According to the metrics provided, the replica with the least CPU utilization and highest throughput is selected as the new leader, so that the cluster leader can manage the current load efficiently. The performance of the leader after the election is equally observed as a constant check for the system requirements of throughput and CPU. This is checked against a target value of throughput to adjust the amount of replicas required in a cluster, thereby autoscaling is done dynamically.

4.2.1. Steps for predicting the number of replicas

1. **Determine the Desired Throughput**: The first step is to define the desired performance (Y) based on the requirements of the system or the expected workload. This could be driven by service level agreements (SLAs) or performance benchmarks that the system must achieve.
2. **Specify the Number of Applications**: Next, the current number of running applications (P) in the cluster is entered into the model. This value represents the active workloads that the system is managing, which directly impacts resource needs.
3. **Predict the Number of Replicas (Q)**: With the desired throughput and the number of applications known, the model is used to solve for the optimal number of replicas (Q) needed to achieve the target throughput. Rearranging the equation and isolating Q , the model predicts how many replicas should be provisioned to achieve the desired performance.

Optimal Replica Leader Election Integrated with Nonlinear Regression Predictive Replica Model

Initialization: All replicas start in the follower state; Each replica maintains its own score, initialized to 0; Use the Predictive Replica Model to forecast the number of replicas required based on incoming traffic data;

```

while true do
    foreach replica do
        Collect throughput and resource metrics;
        Calculate the replica's score based metrics;
    end
    foreach replica do
        if replica's score is significantly higher than other replicas' scores then
            Transition replica to the candidate state;
            Initiate leader election process;
            Announce candidacy;
            Broadcast candidacy to other replicas;
        end
    end
    foreach candidate replica do
        Other replicas receive candidacy announcement;
        foreach replica do
            Acknowledge candidacy;
            Compare scores;
            if follower's score is lower than candidate's score then
                Transition follower to follower state;
                Acknowledge candidate as leader;
                Start following candidate's leadership;
            end
        end
    end
    foreach elected leader do
        Send periodic "heartbeats" to maintain leadership status;
        Periodically update leader record in EP to renew leadership;
        if heartbeat failures exceed threshold then
            Trigger leader election process;
        end
    end
    foreach replica do
        Periodically update scores based on latest metrics;
        Compare updated scores with other replicas;
        if follower's score becomes higher than leader's score then
            Transition follower to candidate state;
            Announce candidacy to challenge current leader;
        end
    end
    if traffic forecast changes significantly then
        Adjust the number of replicas based on throughput;
    end
end

```

Algorithm 1: Proposed ORLE with Predictive Replica Model

4. Adjust Based on Real-Time Conditions: As workloads fluctuate, this model allows for dynamic adjustments to the number of replicas. For example, if the number of applications increases, the model can predict how many additional replicas are required to maintain the same throughput. In contrast, if the throughput target changes, the number of replicas can be adjusted accordingly (see Fig. 6).

```

1 def orle_simulation():
2     current_replicas = nr
3     target_throughput = Th
4
5     for i in range(5):
6         print(f"\nCycle {i+1}:")
7         replicas = collect_metrics()
8         leader, stats = elect_leader(replicas)
9         print(f"Leader elected: {leader} with CPU
10 ={stats['cpu_usage']}% and Throughput={stats
11 ['throughput']}")
12         total_throughput = sum(replica['
13 throughput'] for replica in replicas.values())
14         print(f"Total Throughput: {total_
15 throughput} with {current_replicas}
16 replicas.")
17         current_replicas = adjust_replicas(
18         total_throughput, current_replicas,
19         target_throughput)
20         print(f"Adjusted Replicas: {current_
21 replicas}")
22         monitor_leader(leader, replicas)
23         forecasted_throughput =
24         forecast_throughput(total_throughput)
25         print(f"Forecasted Throughput: {forecasted_
26 throughput}")

```

Fig. 6. Using ORLE to predict replica's based on throughput.

Table 2
Throughput levels vs. Number of replicas.

Throughput level	Throughput range (Units)	Action taken	Final number of replicas
Very low	0–199	Scale up	11
Low	200–349	Scale up	12
Medium	350–449	Maintain	10
High	450–599	Scale down	9
Very high	600+	Scale down	8

Table 2 shows how our proposed autoscaling predictive model varies replicas based on the throughput to avoid both under-provisioning and over-provisioning. The model is trained to grow or shrink, depending on system throughput, to optimize resource usage. If throughput is within the range of goals, the number of replicas remains the same; if not, changes occur in the number of replicas. This adaptive scaling mechanism improves the performance of the stability application by counteracting fluctuations in throughput while minimizing unnecessary resource consumption. By implementing this model, Kubernetes clusters can efficiently scale resources based on predictive insights rather than reactive responses. This ensures that the system remains responsive to changes in workload while avoiding over-provisioning, leading to better resource utilization and overall system performance. The integration of this Predictive Replica model with the ORLE algorithm further enhances the scalability and adaptability of the system by making informed decisions about replica management in real time.

5. Performance evaluation

5.1. Experimental setup

In this section, we discuss the experimental setup and the process of comparing the evaluation metrics between the default leader election algorithm, the BLD (balanced leader distribution algorithm) and the proposed Optimal Replica Leader Election (ORLE) algorithm. Evaluation metrics are used to compare the throughput, latency, and energy of the leader election algorithm. The experiment was repeated for 3, 5, 7 applications each with 10, 20, 30, 40, 50 replica's For evaluating the performance of the proposed algorithm, we established a Kubernetes

cluster comprising a single master node and 10 worker nodes, utilizing Kubernetes version 1.29.1 and Docker version 1.6.0. The master node has four CPU cores and 8 GB of RAM, while the ten worker nodes each possess two CPU cores and 8 GB of RAM. Within this cluster, multiple stateful applications were deployed. This paper presents the results of the evaluation of the comparison between the two prior algorithms available for leader election and the proposed ORLE algorithm based on the evaluation metrics for 3, 5 and 7 applications each with 10, 20, 30, 40, 50 replicas. To evaluate the leader election latency, we employ varying quantities of stateful applications on each node to ensure synchronization among the application's replicas. The default leader election algorithm's performance remains moderate, While the Balanced Leader Distribution (BLD) algorithm improves the leader election process in Kubernetes by promoting a more equitable distribution of leadership across nodes, it has its own limitations compared with the proposed Optimal Replica Leader Election algorithm. ORLE's theoretical optimality comes with trade-offs in practicality and adaptability, particularly as the number of applications increases. The Locust tool is a load testing framework that is used to create and send requests to applications [22].

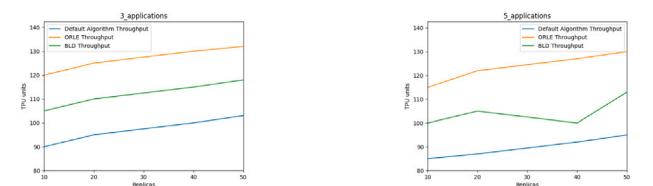
Throughput serves as a key metric reflecting the efficiency of each algorithm in processing application transactions. We compared the performance of the proposed algorithm for leader election with BLD and the default algorithm for leader election according to the role of the replica. For read operations, nodes with both the leader replica and the follower replica exhibit comparable throughput when the number of concurrent requests is increased. In other words, as the workload on the system intensifies with more concurrent read requests, the performance of the nodes with both the leader and follower replicas remains relatively similar in terms of processing these requests, because the read operation can be handled by nodes with the leader replica or follower replica. The read operations are consistent in time, since no coordination or synchronization is needed. However, for write operations, only the leader is allowed to perform the write, as it is responsible for maintaining consistency across the cluster. This introduces additional overhead for coordination and replication. We have experimentally verified that read operations indeed exhibit consistent latencies across the leader and follower pods, while write operations exhibit higher and variable latency because of the leader's exclusive responsibility for handling writes and synchronizing them with followers. Hence, here, we evaluated only the write operation of the leader in both algorithms, as the write operation is performed only by the leader. If a written request comes from a follower, it is forwarded to the leader. The leader election algorithm, with the Optimal Replica Leader Election approach, gave high throughput and low latency in the write operation. In this section, we present the performance evaluation of three leader election algorithms, the default leader election algorithm, the BLD algorithm, and the proposed Optimal Replica Leader Election algorithm. Throughput is measured in transactions per second (TPS). Latency is measured in milliseconds, and energy is measured in Joules.

5.2. Results

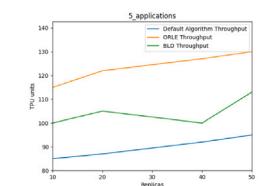
5.2.1. Throughput comparison

Fig. 7 illustrates the comparison between the default leader election algorithm, BLD and the proposed ORLE algorithm in terms of throughput for different numbers of applications.

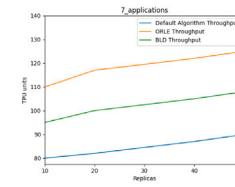
When considering three applications, the default leader election algorithm exhibits a mean throughput of 94.5 TPS, the BLD algorithm exhibits a mean throughput of 111 TPS while the proposed ORLE algorithm exhibits a mean throughput of 125 TPS. The proposed ORLE algorithm displays high throughput, indicating that the proposed ORLE algorithm is more efficient.



(a) Throughput Comparison for 3 Applications

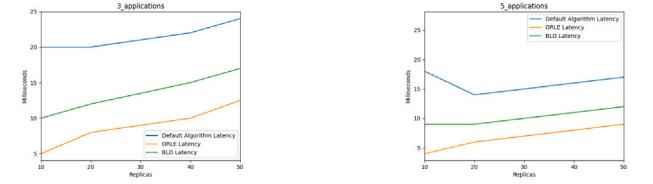


(b) Throughput Comparison for 5 Applications

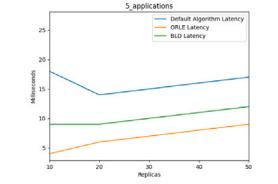


(c) Throughput Comparison for 7 Applications

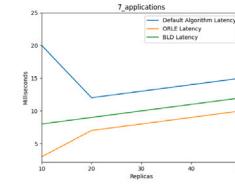
Fig. 7. Throughput comparison for 3, 5, and 7 applications between proposed ORLE and default leader election algorithm.



(a) Latency Comparison for 3 Applications



(b) Latency Comparison for 5 Applications



(c) Latency Comparison for 7 Applications

Fig. 8. Latency comparison for 3, 5, and 7 applications between proposed ORLE and default leader election algorithm.

5.2.2. Latency comparison

Fig. 8 shows the comparison between the default leader election algorithm, the BLD algorithm, and the proposed ORLE algorithm in terms of latency. With three applications, the default leader election algorithm exhibits a mean latency of 19 ms, while the BLD algorithm exhibits a mean latency of 13.5 ms. The proposed ORLE algorithm has the lowest latency, with a mean latency of as low as 9 ms, indicating that the proposed ORLE algorithm is more efficient.

Table 3 represents the mean, standard deviation for the values obtained for the leader election process as statistical analysis of the default leader election algorithm, BLD algorithm and the proposed ORLE algorithm for 3, 5 and 7 applications. The high throughput and lowest latency values of the Optimal Replica Leader Election algorithm compared with other two algorithms show that the proposed ORLE algorithm is an efficient algorithm for stateful applications in Kubernetes using the fastest replica approach to demonstrate high availability, scalability, and efficiency.

5.3. Energy comparison

RAFT and BLD consume more energy than ORLE for the smaller number of replicas because of the differences in the leader selection

Table 3
Performance metrics for 3, 5, and 7 applications.

Metric	App 3-Std. Dev.	App 3-Mean	App 5-Std. Dev.	App 5-Mean	App 7-Std. Dev.	App 7-Mean
Default Kubernetes Throughput	3.8	94.5	4.2	88.5	5.3	84.5
BLD Throughput	4	111	3.5	108	4.5	106
Proposed ORLE Throughput	5	125	5.5	122	6.5	117.5
Default Kubernetes Latency	3	19	2.5	16.5	2	14
BLD Latency	2	13.5	2	12	1.8	11
Proposed ORLE Latency	2.5	9	1.8	7.5	1.5	6.5

Table 4
Energy consumption comparison for ORLE, RAFT, and BLD algorithms.

Replica count	ORLE energy (J)	RAFT energy (J)	BLD energy (J)
10	94.85	130.76	101.80
20	215.13	228.53	244.20
30	336.90	313.99	329.78
40	485.97	441.60	431.65
50	553.50	531.39	593.19

strategy used in the scheme. When the number of replicas increases, BLD becomes the most energy-consuming algorithm, which will be suitable for ORLE in some simple and medium-sized Kubernetes clusters. The energy efficiency of ORLE is based on the ability to choose the least resource-consuming replica of the used service, thus avoiding the utilization of additional CPU and memory resources, leading to increased power consumption and operational costs. With self-adjusting replica model of replica, this means that there is always the right number of replicas that are created according to the current throughput which prevents over-provisioning and under-provisioning. This proactive scaling lowers the overhead costs of the infrastructure by allowing the provision of resources only when required. Since ORLE reduces both the energy consumption and resources needed in a cloud environment, it achieves better cost efficiencies without compromising performance or scalability. Our proposed ORLE algorithm achieves higher throughput due to its integration of resource utilization and throughput metrics for leader selection, ensuring minimal overhead and optimal resource allocation. Lower latency results are attributed to dynamic leader election that minimizes inter-node communication delays (see Table 4).

5.4. t-test analysis for throughput and latency

5.4.1. t-test analysis between ORLE and default kubernetes algorithm

While performing a paired two-sample t-Test for means of throughput, between proposed ORLE throughput (TP) and default Kubernetes algorithm throughput (KTP), the estimated difference is:

$$\text{Difference}(TP - KTP) = 32.13333333 \quad (2)$$

While performing a paired two-sample t-Test for means of latency, between Proposed ORLE latency (LT) and default kubernetes algorithm latency (KLT), the estimated difference is:

$$\text{Difference}(LT - KLT) = -8.96666667 \quad (3)$$

The t-values are very high for both comparisons ($t(14) = 39.11$ for TP vs. KTP and $t(14) = 7.05$ for LT vs. KLT), indicating a large difference between the sample means relative to the variation in the data. The p-values are extremely low ($p = 5.30 \times 10^{-16}$ for TP vs. KTP and $p = 2.9 \times 10^{-6}$ for LT vs. KLT), much lower than the typical significance level of 0.05.

5.4.2. t-test analysis between ORLE and BLD algorithm

While performing a paired two-sample t-Test for means of throughput, between proposed ORLE throughput (TP) and BLD algorithm throughput (BLDTP), the estimated difference is:

$$\text{Difference}(TP - BLDTP) = 16.33 \quad (4)$$

While performing a paired two-sample t-Test for means of latency, between Proposed ORLE latency (LT) and BLD algorithm latency (BLDLT), the estimated difference is:

$$\text{Difference}(LT - BLDLT) = -3.30 \quad (5)$$

The t-values are very high for both comparisons ($t(14) = 19.37$ for TP vs. BLDTP and $t(14) = -7.77$ for LT vs. BLDLT), indicating a large difference between the sample means relative to the variation in the data. The p-values are extremely low ($p = 1.66 \times 10^{-11}$ for TP vs. BLDTP and $p = 1.93 \times 10^{-6}$ for LT vs. BLDLT), much lower than the typical significance level of 0.05. This implies that there is a statistically significant difference in the means that have been observed, meaning we can reject the null hypothesis that there is no difference in means. For both tests, the results are statistically significant. The low p-values indicate that the differences observed are unlikely to be due to random chance. This establishes that the throughput of the proposed algorithm is higher than the default Kubernetes algorithm and BLD algorithm and the latency of the proposed algorithm is lower than that of the default Kubernetes algorithm and BLD algorithm. Thus, the proposed algorithm is much superior compared to the default algorithm and BLD algorithm in Kubernetes.

6. Conclusions and future research

This paper presents a new strategy to estimate the number of replicas that would require to give a desired throughput and is more optimal than the normal Kubernetes strategy. As mentioned we have chosen the fastest replica in the network and by combining the ORLE algorithm with the Predictive Replica Model we can accurately forecast the number of replicas needed according to the throughput. This paper provides a new solution to improve the resource utilization, and availability of Kubernetes stateful applications. To achieve this, our proposed algorithm monitors throughput and resource utilization of each replica continuously so as to make informed selections of which replica becomes a leader and so improves the general performance of the system. Analysis of the results shows that adjusting the leader election process in Kubernetes clusters can bring considerable performance gains for stateful applications. In future work, a study could be done on tuning the algorithm with different thresholds of selecting candidates and a deeper analysis of the algorithm can be done in larger and more complex Kubernetes environment. We also aim to develop a comprehensive cost computation model to better calculate throughput for multiple applications and replicas, further enhancing resource management and optimizing performance within Kubernetes clusters. This approach provides a robust framework for managing resources efficiently and optimizing performance in Kubernetes environments.

CRediT authorship contribution statement

Indrani Vasireddy: Writing – original draft, Formal analysis, Data curation, Conceptualization. **Rajeev Wankar:** Writing – review & editing, Supervision, Conceptualization. **Raghavendra Rao Chillarige:** Writing – review & editing, Supervision, Methodology.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

- [1] Chang C, Yang S, Yeh E, Lin P, Jeng J. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In: GLOBECOM 2017-2017 IEEE global communications conference. 2017, p. 1–6.
- [2] Dockerio. Docker documentation. 2023, <https://docs.docker.com/>. (Accessing since 2020).
- [3] Junior P, Miorandi D, Pierre G. Stateful container migration in geo-distributed environments. In: 2020 IEEE international conference on cloud computing technology and science (cloudCom). 2020, p. 49–56.
- [4] Mirampalli S, Wankar R, Srirama SN. Evaluating NiFi and MQTT based serverless data pipelines in fog computing environments. Future Gener Comput Syst 2024;150:341–53. <http://dx.doi.org/10.1016/j.future.2023.09.014>.
- [5] Loo H, Yeo A, Yip K, Liu T. Live pod migration in kubernetes. 2018, <https://www.ubc.ca/>.
- [6] Junior P, Miorandi D, Pierre G. Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes. In: 2022 IEEE 6th international conference on fog and edge computing. ICFEC, 2022, p. 26–33.
- [7] Vasireddy I, Wankar R, Chellarige R. Recreation of a sub-pod for a killed pod with optimized containers in kubernetes. In: 3rd international conference on expert clouds and applications. ICOECA, 2023.
- [8] Indrani, Wankar R, Chellarige R. Pod Migration with Optimized Containers Using Persistent Volumes in Kubernetes. Springer; 2024.
- [9] The Kubernetes Authors. Kubernetes documentation. 2020, <https://kubernetes.io/docs/>. (Accessing since 2020).
- [10] Nguyen TT, Yeom YJ, Kim T, Park DH, Kim S. Horizontal pod autoscaling in kubernetes for elastic container orchestration. Sensors 2020;20(16):4621. <http://dx.doi.org/10.3390/s20164621>.
- [11] Kim E, Lee K, Yoo C. On the resource management of kubernetes. In: 2021 international conference on information networking. ICOIN, 2021, p. 154–8.
- [12] Kim SH, Kim T. Local scheduling in KubeEdge-based edge computing environment. Sensors 2023;23(3):1522. <http://dx.doi.org/10.3390/s23031522>.
- [13] Nguyen Q, Phan L, Kim T. Load-balancing of kubernetes-based edge computing infrastructure using resource adaptive proxy. Sensors 2022;22(2869).
- [14] Jain N, Mohan V, Singhai A, Chatterjee D, Daly D. Kubernetes load-balancing and related network functions using P4. In: Proceedings of the symposium on architectures for networking and communications systems. 2021, p. 133–5.
- [15] Zhao Anqi, Huang Qiang, Huang Yiting, Zou Lin, Chen Zhengxi, Song Jianghang. Research on resource prediction model based on kubernetes container auto-scaling technology. In: IOP conference series: materials science and engineering. IOP Publishing; 2019, 052092.
- [16] Liu Junnan, Ding Yongkang, Liu Yifan. A balanced leader election algorithm based on replica distribution in Kubernetes cluster. Clust Comput 2024;1–10.
- [17] Ding Yongkang, Liu Junnan, He Xin. Min-leader optimal scheduling algorithm in kubernetes clusters. In: Advances in artificial intelligence, big data and algorithms. IOS Press; 2023, p. 977–82.
- [18] Galery Käser Lucas. Semi-random leader election for distributed moving target defense coordination in kubernetes. 2023.
- [19] Nguyen ND, Kim T. Balanced leader distribution algorithm in kubernetes clusters. Sensors 2021;21(3):869. <http://dx.doi.org/10.3390/s21030869>.
- [20] The Prometheus Authors. Prometheus documentation. 2024, <https://prometheus.io/docs/>. (Accessed January 2024).
- [21] Grafana Labs. Grafana documentation. 2024, <https://grafana.com/docs/>. (Accessed January 2024).
- [22] Locustio. Locust documentation. 2024, <https://docs.locust.io/>. (Accessed January 2024).