ACCEPTED FROM OPEN CALL

Toward Highly Scalable Load Balancing in Kubernetes Clusters

Nguyen Dinh Nguyen and Taehong Kim

The authors analyze the Kubernetes architecture by focusing on load balancing, a leader election algorithm, and a leader-based consistency maintenance mechanism. They address the following challenges: the leader has heavy loads due to its inherent design, and the Kubernetes leader election algorithm cannot evenly distribute the leader throughout nodes.

Abstract

Container-based virtualization has recently been flourishing due to its ease of deployment and flexibility of resource provisioning. Kubernetes, a well-known open source project, is an orchestration platform for containerized applications. Herein, applications can have multiple replicas to provide high scalability and availability, but some applications for stateful service require a leader to be elected to maintain consistency and coordinate tasks among the replicas. In this article, the Kubernetes architecture is first analyzed by focusing on load balancing, a leader election algorithm, and a leader-based consistency maintenance mechanism. Then we address the following challenges: the leader has heavy loads due to its inherent design, and the Kubernetes leader election algorithm cannot evenly distribute the leader throughout nodes. Finally, experimental results are provided to prove the importance of leader distribution throughout nodes in the cluster for highly scalable load balancing.

Introduction

Recently, virtualization has become a key technology for cloud computing [1]. In particular, container-based virtualization is a lightweight method to create a virtual environment, which runs at the software level within the host machine [2]. It has been growing rapidly with alternating virtual machines (VMs) due to low resource usage and high portability [3]. Moreover, container-based micro-services are revolutionizing the methods of application design [4]. Instead of employing traditional single monolithic architecture wherein all components are combined into a single unit, the container-based micro-services allow for an application to be composed of multiple lightweight containers across a large number of nodes. In large-scale systems, several containers are dispersed throughout the network, which requires an orchestration tool to deploy the containers and manage resources and services.

Kubernetes (K8S) is one of the most popular and powerful open source orchestration tools for containerized applications [5]. It provides diverse functions for container orchestration, such as deployment, resource management, auto-scaling, and load balancing [6]. Kubernetes manages resources in the unit of pods, which are the smallest logical units; further, it dynamically adjusts pod resources according to requirements of the application and the available resources in the cluster. There are three types of autoscalers, namely, the cluster autoscaler (CA), vertical pod autoscaler (VPA), and horizontal pod autoscaler (HPA). Among these, the HPA provides a seamless and flexible service by monitoring resource utilization, such as CPU/RAM utilization or other custom metrics, and automatically scaling the number of pods in an application up or down. For instance, when the CPU usage limit of a pod exceeds the configured usage threshold, the HPA creates additional replicas of the pod to mitigate the overload on the current pods of the application in the cluster. Conversely, if the resource usage is lower than expected, the HPA conserves the available resources by removing the pods.

In Kubernetes, load balancing involves the distribution of network traffic among the replicas of applications in accordance with load balancing strategies to maximize the scalability and availability of the service. In other words, user requests are handled by one of the replicas, and users need not be aware of the internal process. Each replica in Kubernetes can have its own data store; thus, it becomes imperative to maintain consistency across these distributed data stores in a stateful application. To handle this consistency problem, several distributed systems [7–9] exploit a leader-based consistency maintenance scheme, where an elected leader is solely responsible for updating data and then replicating it to the followers. As all the data update requests are forwarded and handled only by the leader, the leader has a heavier load compared to that of the followers due to the inherent design of such a scheme. Particularly, when leaders of different applications are concentrated in a specific node, it causes the user requests to also be concentrated on that node, which may result in a bottleneck within the system.

Despite the role-based disparity in workloads, a leader-based consistency maintenance mechanism for distributed data stores is indispensable in a stateful application. Kubernetes also provides a simple leader election algorithm in the form of a container image. Here, we propose a leader-based consistency maintenance mechanism for stateful services. In addition, we discuss the importance of leader distribution throughout the nodes in a Kubernetes cluster for highly scalable load balancing. The rest of this article is organized as follows. First, we review the Kubernetes architecture and its services. We then analyze the leader

Digital Object Identifier:
10.1109/MCOM.001.1900660

The authors are with Chungbuk National University. The corresponding author is Taehong Kim.

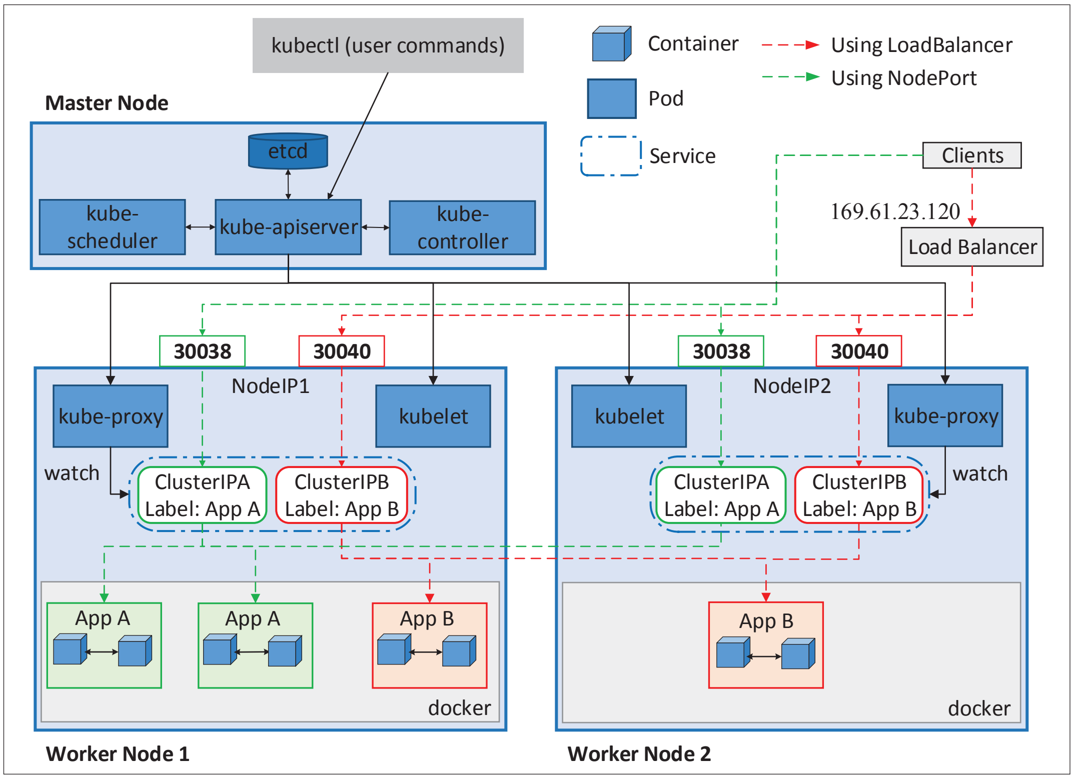


Figure 1. Kubernetes architecture.

Kubernetes is an open-source based container orchestration platform for automating deployment, scaling, and management of containerized applications. In the Kubernetes platform, pods, each of which consists of a group of one or more containers, are the smallest execution unit. They are tightly coupled, use a unique cluster IP, and share storage. The containers inside a pod can easily communicate with each other on the localhost.

election algorithm and a leader-based consistency maintenance mechanism in Kubernetes. Next, we evaluate the imbalance of the resource usage and the throughput with respect to different leader distribution scenarios in the cluster. Finally, the conclusions are presented.

Overview of Kubernetes

Kubernetes Architecture

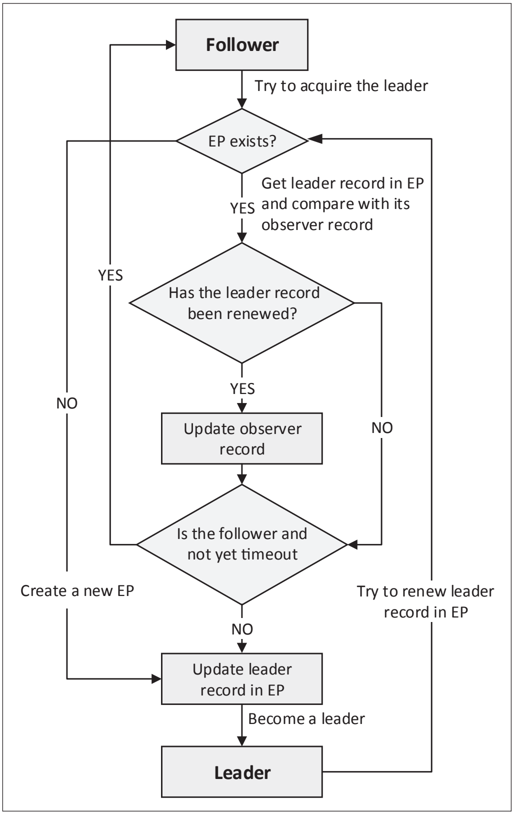
Kubernetes is an open source container orchestration platform for automating deployment, scaling, and management of containerized applications. In the Kubernetes platform, a pod, which consists of a group of one or more containers, is the smallest execution unit. Pods are tightly coupled, use a unique cluster IP, and share storage. The containers inside a pod can easily communicate with each other on the *localhost*.

The Kubernetes cluster consists of master and worker nodes, as shown in Fig. 1. Each node can be operated in either a physical machine or a VM. By default, there is a single master node responsible for controlling the cluster, but multiple master nodes can be utilized to provide high availability. The master node is made up of different components including the *kube-apiserver*, *kube-controller*, *kube-scheduler*, and *ectd* [5]. The *kube-apiserver* provides an entry point for the Kubernetes control plane to control the entire Kubernetes cluster. It receives all requests from the client and all other components in the cluster, authenticates them, and updates the corresponding objects in the Kubernetes's database. The *kube-controller* continuously watches the shared state of the cluster using *apiserver* and tries to alter the current state to the desired state. For example, it is responsible for noticing and responding when nodes go down or ensuring the correct number of running replicas for the application in the cluster. The *kube-scheduler* looks out for unscheduled pods and deploys these pods to an appropriate node in the cluster. The scheduling decision is based on some factors such as resource requirements, hardware/software/policy constraints, affinity, and anti-affinity specifications. The *etcd* is a distributed, consistent key-value store and is used to store all cluster data, including configuration data and the state of the cluster.

The application is deployed in the worker nodes. Each worker node is managed by the master node and consists of three main components that include the *kubelet*, *container runtime* (like Docker [10] and rkt [11]), and *kube-proxy*. *Kubelet* is responsible for managing the containers running on the machine. It communicates with the master node to report current states of the worker node and obtain decisions from the master. *Container runtime* is used to run containers in pods. It is responsible for pulling the container image from a registry, unpacking the container, and running the container. *Kube-proxy* runs on each node that implements the Kubernetes service. It maintains the network rules that allow communication to pods from inside or outside the cluster.

In order to manage the replicas of the application (e.g., creating or deleting replicas, maintaining a stable set of running replica pods at any given time), Kubernetes provides some objects such as *Deployment* and *StatefulSet*. Specifically, *Deployment* is usually used to manage stateless applications, while *StatefulSet* is used for stateful applications. In addition, Kubernetes also provides services that define a logical set of pods in the cluster and a policy for accessing them. A *Label*

In order to implement the leader election, a well-known algorithm such as Raft or Zookeeper can be used. However, to avoid high implementation cost to the user as well as to achieve easy deployment of the leader election, Kubernetes itself provides a leader election container image.



Follower
Try to acquire the leader
EP exists?
Get leader record in EP and compare with its observer record
YES
Has the leader record been renewed?
YES
Update observer record
Is the follower and not yet timeout
YES
NO
NO
Create a new EP
Try to renew leader record in EP
NO
Update leader record in EP
Become a leader
Leader

Figure 2. Leader election algorithm in Kubernetes.

is used to select the set of pods targeted for the service.

Service in Kubernetes

Because the IP address of the pods changes every time they are restarted, it is difficult to access a pod directly using its IP address. A service is an abstraction for a group of pods. It is bound to a *ClusterIP*, which is a virtual IP address that never changes. Therefore, the client can always connect to a group of pods via the *ClusterIP* of the service. When clients connect to the *ClusterIP*, their traffic is automatically transferred to an appropriate backend pod. This configuration is implemented using *kube-proxy*. It constantly watches for the creation and deletion of service objects. *Kube-proxy* supports three proxy modes: *userspace*, *iptables*, and *IPVS*, which stands for IP virtual server. These modes operate slightly differently. For example, the *userspace* proxy selects a backend pod in a round-robin manner, and the *iptables* proxy selects a backend pod in a random manner. Furthermore, *IPVS* provides more options, such as round-robin, destination hashing, and source hashing, to select the backend pod. The default for the *kube-proxy* mode in Kubernetes is the i*ptables* proxy.

However, *ClusterIP* is only reachable from within the cluster. To access the cluster from outside, there are two typical types of service in Kubernetes: *NodePort* and *LoadBalancer*. With regard to the *NodePort* service, Kubernetes opens a static port on each node (called "NodePort"). If we do not specify a value in the NodePort, it selects a random port. We can access the service from outside the cluster using the IP address of the node and the NodePort, like <*NodeIP*>:<*NodePort*>. For example, in Fig. 1, we create a *NodePort* service with NodePort 30038 for a set of pods that has the label *App A*. In order to access the pods, clients can use the address *NodeIP1:30038*. *Kube-proxy* also creates corresponding iptables rules to capture the traffic coming to the 30038 port and passes it to *ClusterIPA*. Eventually, it redirects the traffic to the back-end pods, and the selection of back-end pods depends on the proxy configuration.

The *LoadBalancer* service works when we are using a cloud provider for the Kubernetes cluster. Clients can access the service via a public IP address that is provided by the cloud provider. The cloud provider configures the load balancer in its network to proxy the NodePort on multiple nodes, and the load balancing algorithm depends on the cloud provider implementation. In addition, the *NodePort* and *ClusterIP* services are also created automatically together with *LoadBalancer*, and they are used to redirect the external traffic to an appropriate pod in the cluster. For example, as shown in Fig. 1, the label *App B* is used to select the set of pods targeted by the *LoadBalancer* service. An external IP, 169.61.23.120, is provided by the cloud provider. Clients send requests to the *LoadBalancer* service through this address, and the requests are routed to NodePort 30040 of worker nodes 1 or 2 according to the cloud provider implementation and redirected to one of the backend pods through *ClusterIPB*.

Leader-Based Consistency Maintenance Mechanism in Kubernetes

In distributed systems, consensus protocols play a crucial role in ensuring the consistency of all replicas in the presence of node faults. Paxos [7], Raft [8], and Zookeeper [9] are representative consensus protocols utilized in real systems, and these require agreements on transactions to ensure the consistency of the replicated data store. In Paxos, the leader sends a proposal to all replicas, which is applied throughout all replicas if a majority of the replicas have accepted the proposal. Raft shows equivalent performance and fault tolerance to those of Paxos as it is derived from Paxos; however, it reconstructs the consensus process into subproblems such as leader election and log replication. Zookeeper provides reliable coordination in the distributed system, and it can be used to maintain in-memory databases and log transactions. These consensus protocols commonly employ the leader election process to preserve the consistency of replicas through the leader. This indicates that the leader-based consistency maintenance mechanism, despite its inherent issues such as a concentrated workload on the leader [12], is essential. Kubernetes also provides a simple leader election algorithm that can be used to maintain consistency among replicas in a stateful application.

Leader Election Algorithm in Kubernetes

Typically, in a leader election algorithm, a set of candidates compete to declare themselves the leader. If one of the candidates wins, it becomes the leader. Once the leader election process is complete, the leader continuously sends "heartbeats" to retain its leading position. If the current leader fails for some reason, the other candidates attempt to become the leader. In order to implement the leader election, a well-known algorithm such as Raft or Zookeeper can be used. However, to avoid high implementation cost to the user as well as to achieve easy deployment of the leader election, Kubernetes itself provides a leader election container image.

This algorithm is implemented based on Kubernetes endpoint objects (EPs), which hold an annotation about the information for the leader election. There are important parameters regarding the leader record in EPs: *holderIdentity* is the name of the leader, *leaseDurationSeconds* is the timeout duration that a non-leader candidate waits to forcibly acquire leadership, and *renewTime* is the time when the leader renews the leader record in an EP. The leader periodically renews the leader record in the EP to retain its leadership. If the leader fails to renew the record after a timeout duration, the other candidates are free to take over the record.

The procedure of the leader election algorithm is described in Fig. 2. Once the pods initialize as followers, they race to become a leader. First, these candidates check whether the EP exists or not (the name of the EP is defined in the leader election configuration file). If the EP does not exist, it creates a new EP, updates the leader record in the EP, and becomes a leader. After the leadership is achieved, it periodically renews the leader election record in the EP. Otherwise, the candidate gets the current leader record in the EP. An observer record is maintained by each replica to hold the leader election record of the leader in the last acknowledgment. After obtaining the information about the leader from the EP, it compares its observer record with the current leader election record to know whether the leader record has been renewed or not. If the leader record has been renewed, it updates the observer record. Subsequently, if it is the leader, it updates the leader record in EP to keep its leading position. If it is a follower, it checks the timeout by calculating the total time elapsed between the last observation to the current time. If this value is over *leaseDurationSeconds*, it updates the leader election record in the EP and becomes a new leader. Otherwise, it remains in the follower state and periodically tries to acquire leadership by repeating the above procedure.

Stateful Service Model for Consistency Maintenance

Stateful applications are services that require saving data to persistent data storage such as a database or key-value store for use by the server, clients, and other applications [13]. In Kubernetes, each pod in the stateful application can create its own persistent data store using Persistent Volume Claim (PVC). For example, a stateful application with two replicas will create two pods; thus, there are a total of two PVCs, and each pod has its own persistent data store.

As discussed in the introduction, an application in the Kubernetes cluster can run multiple replicas to increase application availability and improve performance. For example, throughput and latency can be improved due to the distribution of incoming requests among replicas by load balancing service. However, because each replica in the stateful application has its own data store, handling the inconsistency among these distributed data stores is a challenge that needs to be addressed.

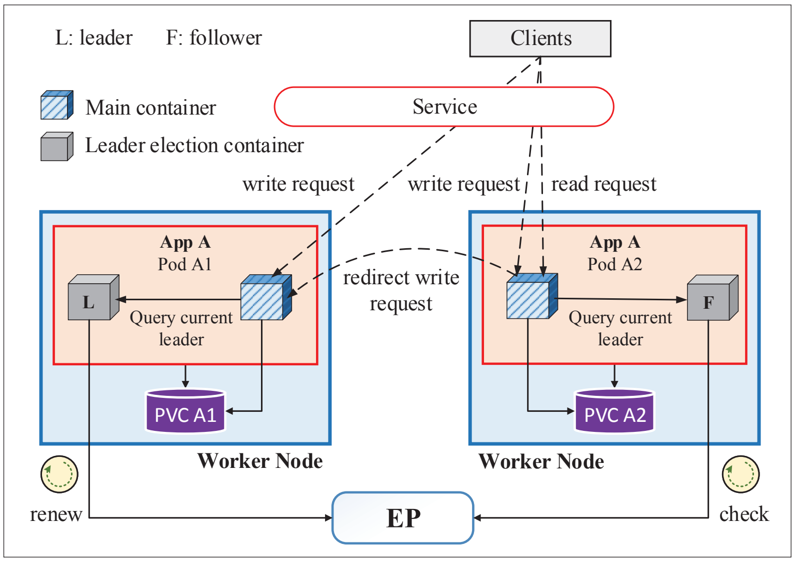


Figure 3. Stateful service model for consistency maintenance.

To overcome the inconsistency problem of stateful applications, we design a consistency maintenance mechanism based on the Kubernetes leader election algorithm, as shown in Fig. 3. The stateful application can be deployed using StatefulSet with multiple replicas, and each replica has its own data store to save and retrieve data for each write and read operation. For the leader-based model, the application needs to designate one replica among the set of replicas as the leader and the other replicas as followers. To achieve this, a pod of the application is composed of a main container and a leader election container beside the main container. The main container is responsible for handling requests from clients, and the leader election container carries out the leader election process among replicas of the application. Since the leader election container provides a simple web server that returns a simple JSON object containing the current leader's name, the main container in the same pod can simply search for the leader over a specified address (e.g., http://127.0.0.1:4040). Eventually, the main container can easily become aware of its role (leader or follower) and handle the request from the client based on its role. Once the clients send requests to the set of replicas of an application from outside through the Kubernetes services, it handles the requests based on the read or write operation. The read requests are served by the received pod regardless of its leader/follower role. On the contrary, the write requests are handled only by the leader. Thus, if the received pod is a follower, it redirects the write requests to the leader.

It is obvious that the leader has a higher workload than the follower as the leader handles redirected requests in the leader-based consistency mechanism. Moreover, it has been proven by [12], using a Raft-based software-defined networking (SDN) controller cluster, that if many leaders are elected on a specific node, the workload imbalance can be aggravated. This hinders the full

Our experimental results proved that the Kubernetes leader election algorithm does not consider leader distribution when multiple applications are involved, which may result in performance degradation due to the inherently high workload on leaders.

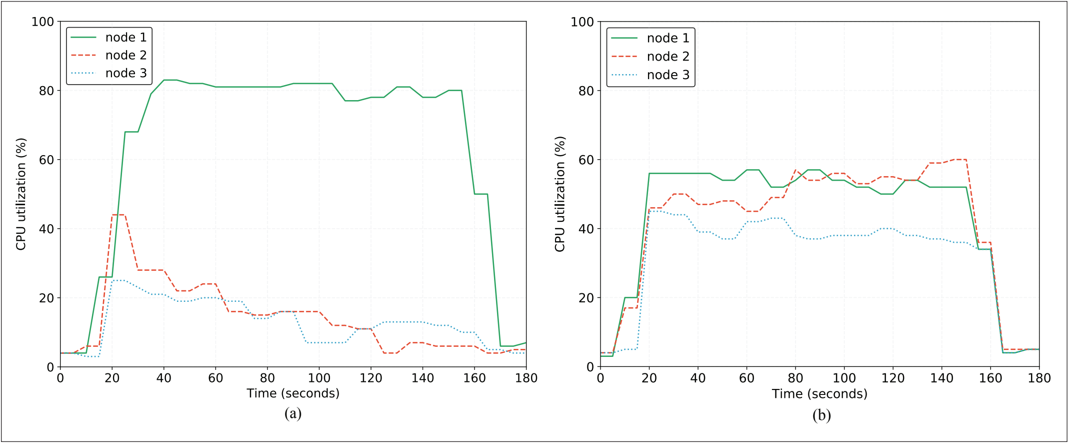


Figure 4. CPU utilization in nodes: a) concentrated leaders; b) balanced leaders.

exploitation of the computational resources of the distributed system. The same problem applies to Kubernetes, which can execute multiple applications in a single cluster comprising several worker nodes. We consider a scenario in which different stateful applications are deployed in the cluster. Each application has multiple replicas and one leader among these replicas. The current leader election algorithm in Kubernetes does not consider the node where the leader is deployed. Consequently, it is highly possible that most leaders are concentrated on a specific worker node. It leads to a workload imbalance problem among worker nodes — one node with many leaders may have a heavier workload than other nodes. Therefore, it can cause a bottleneck in the cluster, which will result in a significant decrease of the system performance.

Evaluation and Discussion

Experimental Settings

To evaluate the system performance in terms of CPU consumption and throughput, we set up a Kubernetes cluster comprising a set of four nodes with Kubernetes version 1.14.2 and Docker version 18.09.6. One master node runs with 4 CPU cores and 4 GB RAM, and three worker nodes run with 4 CPU cores and 3 GB RAM. We deploy five applications in the cluster. Each application has five replicas and one leader among these replicas. In our evaluation, we use the *NodePort* service to expose the application to the outside, and *Hey* [14] is used to create and send requests to each application.

Leader Distribution with Multiple Applications

We first evaluate the leader distribution performed using the Kubernetes leader election algorithm. The experiment is repeated 100 times, and the results indicate that 5 leaders are distributed in three nodes with a ratio of 2.78:1.52:0.7, implying that a substantial portion of the leaders are assigned to a specific node, as the current leader election algorithm in Kubernetes does not optimize leader distribution.

As our goal is to analyze the effect of leader distribution among nodes in a cluster, the performance between “concentrated leaders” and “balanced leaders” scenarios are compared in the following subsections. For concentrated leaders, all the leaders of the applications are assumed to be contained in one specific node. For balanced leaders, the number of leaders in each node is balanced. In our experiment, the leader distribution is 2, 2, and 1, which corresponds with the three worker nodes.

Biased Workload on the Leader

Figure 4 illustrates the CPU utilization of each worker node for the write operation according to the leader distribution. Here, four clients simultaneously send write requests to each application for 150 s, where the requests are distributed to each of the replicas of the applications using the iptables proxy mode. In Fig. 4a, five leaders of five applications are contained in node 1. The CPU usage of node 1 is more than 80 percent while clients are sending write requests to the applications, whereas that of nodes 2 and 3 is just roundly 20 percent. This is because the write requests are handled only by the leader. Therefore, the node that contains all the leaders can be overloaded with CPU resources, thereby becoming a bottleneck in the system. In contrast, in Fig. 4b (two leaders in nodes 1 and 2, one leader in node 3), balanced CPU utilization is observed among the nodes, which is approximately 60 percent for nodes 1 and 2. It indicates that if the leader distribution is balanced among the nodes, the workload can be effectively shared between these nodes.

Effect of Leader Distribution on Throughput

Next, we analyze the effect of the leader distribution on the throughput by allowing a number of clients to simultaneously send write requests directly to the leader of each application. The number of clients accessing each application is increased from 1 to 32, and the total number of requests per application is set at 20,000.

As can be inferred from Fig. 5, the cumulative throughput for five applications demonstrates that the system performance is substantially affected by the distribution of leaders among the nodes. The throughput obtained with 1 client in balanced leaders is approximately 48.7 percent higher than that obtained in concentrated leaders. Notably, the throughput in the “concentrated leaders” scenario becomes relatively constant due to a bottleneck when the number of clients is 4 or higher, although it tends to increase with an increase in the number of clients in the case of balanced leaders. This is attributed to the fact that all write requests are handled by the leader alone. In the "concentrated leaders" scenario, only one node is allocated to handle all the client requests; hence, the performance of the cluster relies completely on the capacity of this node. Furthermore, in the case of balanced leaders, the client requests are effectively handled by distributing the loads among the nodes in the cluster. Consequently, the overall throughput of the applications in the cluster is improved substantially in the case of balanced leaders. This proves the importance of leader distribution for balancing the workload to effectively handle client requests.

Finally, it is necessary to discuss the optimization of leader distribution to achieve highly scalable load balancing in Kubernetes. This can simply be achieved by updating the leader election algorithm provided in the form of a container image to balance the leaders throughout the cluster nodes. As the leader election algorithm is provided independent from the Kubernetes platform, developers and system administrators can fully exploit the performance improvements achieved via the updated algorithm.

Conclusion

In this article, we investigate Kubernetes in terms of its architecture, services, and core functions such as the HPA and load balancing. We also design a leader-based consistency maintenance mechanism targeted at stateful services. Our experimental results prove that the Kubernetes leader election algorithm does not consider leader distribution when multiple applications are involved, which may result in performance degradation due to the inherently high workload on leaders. There have been various implementations of leader election algorithms for maintaining consistency in the Kubernetes cluster, and it is important to consider balancing the number of leaders throughout the nodes in the cluster to maximize the performance of the system.

Acknowledgments

This work was partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (no. NRF-2019R1F1A1059408) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (no. 2018-0-00387, Development of ICT based Intelligent Smart Welfare Housing System for the Prevention and Control of Livestock Disease).

References

[1] M. G. Xavier *et al.*, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," *Proc. 2013 21st Euromicro Int'l. Conf. Parallel, Distrib. Network-Based Processing*, 2013, pp. 233–40.
[2] S. Soltesz *et al.*, "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," *Oper. Sys. Rev.*, 2007, pp. 275–87.
[3] J. Santos *et al.*, "Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications," *IEEE Conf. Net. Softwarization*, 2019.
[4] M. Fazio *et al.*, "Open Issues in Scheduling Microservices in the Cloud," *IEEE Cloud Comp.*, vol. 3, no. 5, 2016, pp. 81–88.
[5] Kubernetes, "Production-Grade Container Orchestration"; https://kubernetes.io/, accessed Sept. 10, 2019.

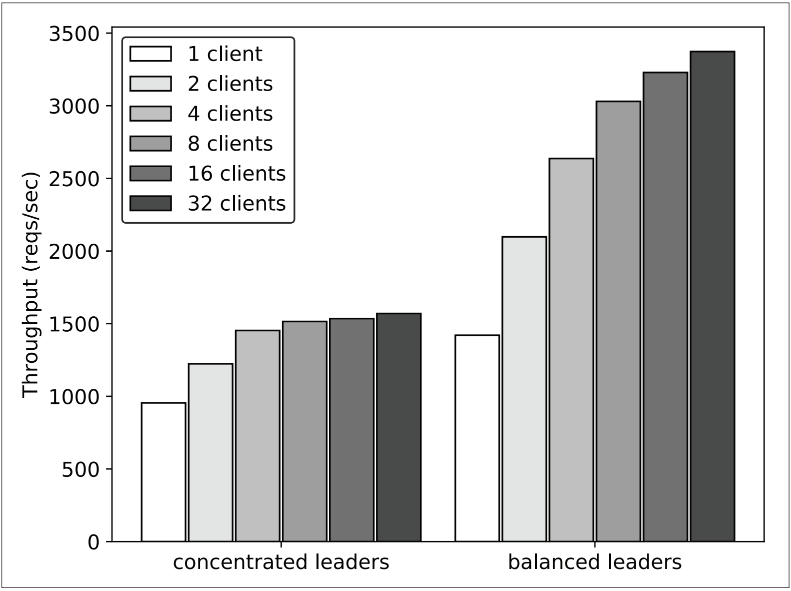


Figure 5. Cumulative throughput on write operation according to leader distribution.

[6] B. Burns, K. Hightower, and J. Beda, *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*, O'Reilly Media, 2017.
[7] L. Lamport, "The Part-Time Parliament," *ACM Trans. Comp. Systems*, vol. 16, no. 2, 1998, pp. 133–69.
[8] J. Ousterhout and D. Ongaro, "In Search of an Understandable Consensus Algorithm," *Proc. USENIX Annual Tech. Conf. 2014*, pp. 305–19.
[9] P. Hunt *et al.*, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," *Proc. USENIX Annual Tech. Conf. 2010*, pp. 145–58.
[10] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Comp.*, vol. 1, no. 3, 2014, pp. 81–84.
[11] rkt, "A Security-Minded, Standards-Based Container Engine"; https://coreos.com/rkt/, accessed Sept. 10, 2019.
[12] T. Kim, J. Myung, and S. E. Yoo, "Load Balancing of Distributed Datastore in OpenDaylight Controller Cluster," *IEEE Trans. Net. Serv. Manag.*, vol. 16, no. 1, 2019, pp. 72–83.
[13] Google Cloud Documentation, "Deploying a Stateful Application"; https://cloud.google.com, accessed Sept. 10, 2019.
[14] Hey, "A Tiny Program Sends Some Load to a Web Application"; https://github.com/rakyll/hey, accessed Sept. 10, 2019.

Biographies

Nguyen Dinh Nguyen (nguyennd@cbnu.ac.kr) received his B.S. degree in electronics and telecommunications from Vietnam National University, Hanoi in 2017. Now he is pursuing his Master's degree in the School of Information and Communication Engineering, Chungbuk National University. His research interests cover cloud computing, edge computing, and SDN/NFV.

Taehong Kim (taehongkim@cbnu.ac.kr) received his B.S. degree in computer science from Ajou University, Korea, in 2005, and his M.S. degree in information and communication engineering from Korea Advanced Institute of Science and Technology (KAIST) in 2007. He received his Ph.D. degree in computer science from KAIST in 2012. He worked as a research staff member at Samsung Advanced Institute of Technology (SAIT) and Samsung DMC R&D Center from 2012 to 2014. He also worked as a senior researcher at the Electronics and Telecommunications Research Institute (ETRI), Korea, from 2014 to 2016. Since 2016, he has been an associate professor with the School of Information and Communication Engineering, Chungbuk National University, Korea. He has been an Associate Editor of *IEEE Access* since 2020. His research interests include edge computing, SDN/NFV, the Internet of Things, and wireless sensor networks.