Distributed Dynamic Scheduling Optimization among Containers

2310175 Yao Yudie

In recent years, the container orchestration system Kubernetes is widely used to implement services in microservice architectures, but important control functions, including the scheduler, are concentrated on the master node. While this design simplifies operation and management, it also brings challenges to the scalability and fault tolerance of the system. If the scheduler stops due to failure, it can affect the scheduling of new pods, cluster monitoring, and other critical operations. The Kube-scheduler performs scheduling by mainly considering CPU and memory resources, which may cause load imbalance between nodes. In addition, because the Kube-scheduler operates as a centralized component in the cluster, its failure may have a significant impact on the scheduling function of the entire cluster.

The purpose of this research is to develop a distributed scheduling system based on Deep Reinforcement(DRS), which optimizes the utilization rate and load balancing of the environment. By distributing the scheduling components, the entire cluster scheduling function is not affected even if one component fails.

In this research, I propose a distributed scheduling method using MARL, where the scheduler learns the resource usage of the nodes, the resource requirements of pods, and the network delay between the nodes, and learns how to make scheduling decisions based on the needs and priorities of different pods. This allows the scheduler to learn from its own experience and autonomously adapt to changes in the environment. This reduces the risk of single points of failure and minimizes the impact of node or component failures on the entire system, while enabling real-time decision-making by leveraging local information on each node, achieving optimal resource utilization and even load distribution.

The MARL model design in this proposed method is defined as follows: The model uses schedulers located on each node as agents. The state space is defined in three types: local state, global state, and waiting pod information. Local state includes directly observable information by each agent (CPU utilization, memory utilization, disk I/O information, node health status and number of currently running pods), while global state represents cluster information (CPU utilization, memory utilization and disk I/O information). Waiting pod information contains the CPU and memory resources requirements of pods. The reward function is designed to consider both local and global rewards, evaluated based on resource utilization. The action space is defined as a discrete space where each agent makes binary decisions on

whether to accept waiting pods for their assigned node. The Deep Q-Network algorithm is implemented for the agent's policy.

The proposed distributed scheduler follows a stepped design similar to the Kubernetes default scheduler. First, in the pre-filter step, each scheduler examines its node's resource information and conditions. Next, during the filter step, the MARL model is implemented where each scheduler determines whether to accept or reject the pod. If a pod is accepted, the node undergoes evaluation and receives a score. In the scoring step, agents share their actions and scores, with the best node selected based on sorted results. Finally, once the decision is made, the model updates in response to the outcome.

The proposed schedulers share decisions through a distributed leadership model. When the scheduler initializes, one instance is elected as the leader, which then establishes heartbeat monitoring and launches a socket server. The remaining scheduler instances transition into a monitoring role. During normal operation, all scheduler instances actively monitor pods and generate scheduling decisions. However, only the leader scheduler has the authority to receive these decisions, determine the optimal node placement, and execute pod binding operations. To ensure high availability, non-leader instances continuously monitor the leader's health through heartbeat checks. If the leader becomes unresponsive, the non-leader instances trigger a new leader election process. The newly elected leader assumes all leadership responsibilities, ensuring uninterrupted system operation.

The verification of this research consists of operational validation and performance evaluation. For operational validation, I confirmed the startup, scheduling and failure handling capabilities of the proposed distributed scheduler and I also measured computation, communication, and resource overheads. The performance evaluation involved deploying 105 pods across three different types of microservices, testing them under both uniform and random workload distributions for the resource utilization compared to the DRS.

As a result of my experiments, regarding system stability, the distributed scheduler successfully maintained cluster stability and scalability during node additions and failures, effectively addressing the single point of failure issue. The distributed architecture proved effective, with individual schedulers making independent decisions while the leader scheduler successfully aggregated these for final scheduling. About the performance evaluation results, under uniform workload distribution, the system achieved superior memory utilization and disk I/O stability, while maintaining stable network bandwidth and CPU utilization. With random workloads, the distributed scheduler demonstrated better performance through stabilized memory utilization, reduced network load, and more consistent disk write speeds compared to DRS. The scheduler also showed efficient resource management by appropriately reduc-

ing resource utilization as tasks completed. Overall, the proposed distributed scheduler showed marked improvements over previous research, particularly in stabilizing memory utilization, reducing network load, and optimizing disk I/O control.

However, due to the short experimental period, I could not comprehensively evaluate the algorithm's performance and stability during long-term operation. While my evaluation was conducted on a Kubernetes cluster with four worker nodes and one master node, real-world environments typically consist of tens to hundreds of nodes. Therefore, evaluation of scalability and resource utilization in practical-scale cluster environments is necessary. Also, the current Python implementation has limitations where only a single scheduler can receive new pod information when deployed as a DaemonSet in the Kubernetes cluster environment. A potential solution is migrating to Go language, which would enable the proposed scheduler to be deployed as a DaemonSet with guaranteed pod instances on each node through the Informer feature. Additionally, long-term experiments are needed to verify algorithm convergence performance and system stability, particularly against various complex workloads. Finally, the implementation of an automatic deployment mechanism remains a crucial area for future development.