2022 International Conference on Artificial Intelligence, Information Processing and Cloud Computing (AIIPCC) | 978-1-6654-6287-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/AIIPCC57291.2022.00051

Horizontal Pod Autoscaling based on Kubernetes with Fast Response and Slow Shrinkage

Qizheng Huo [1,a]
[1]Institution of Systems Engineering
Academy of Military Sciences
Beijing, China
[a]e-mail:1716176610@qq.com

Shaonan Li [2,b]
[2]Institution of Systems Engineering
Academy of Military Sciences
Beijing, China
[b]e-mail:tankshaonan@gmail.com

Yongqiang Xie [3,c*]
[3]Institution of Systems Engineering
Academy of Military Sciences
Beijing, China
[*] Corresponding author [c]e-mail: fitz2020@foxmail.com

Zhongbo Li [4,d*]
[4]Institution of Systems Engineering
Academy of Military Sciences
Beijing, China
[*] Corresponding author [d]e-mail: 2016302580347@whu.edu.cn

***Abstract*—Autoscaling is an important part of Kubernetes. Horizontal Pod Autoscaling (HPA) schedules cluster resources according to the service load status to ensure that services are still running normally when the load increases or decreases. But the default HPA is slow and inflexible. In scenarios with large load changes, it will scale at a high frequency, resulting in a waste of cluster resources and reducing the robustness of the cluster. This paper proposes a Fast-response and Slow-shrinkage algorithm. By setting the expansion rate, tolerance, window time, etc., the flexibility of scaling is increased. When facing the pressure of a sudden increase in traffic, the rapid expansion can be optimized in minutes. The algorithm also adopts a cautious shrinking strategy to reduce and prevent failures caused by secondary access peaks.**

Keywords-component; Resource scheduling; elastic scaling; cloud computing; Kubernetes

I. INTRODUCTION

The main technology of cloud computing is virtualization technology, which is to virtualize computer resources and provide external services with a unified virtualization standard. Virtualization technology[1, 2]significantly improves the utilization rate of computer resources and realizes the decoupling of software and hardware through the virtualization layer. The most widely used virtualization technologies in cloud computing are hypervisor[3] and containers. Before containers, the vast majority of cloud computing vendors provided external services using hypervisor virtual machines. However, a virtual machine has a complete operating system, which has disadvantages such as long boot time, strong environment dependence, high consumption of computing resources, and slow response. The design principle of the container [4, 5] is taken from Linux, and its application process runs directly in the kernel of the host.

Since 2013, with the rise of container technology[6, 7], many PaaS public cloud vendors such as Google and IBM have taken advantage of the above advantages of containers and began to use container technology. Deploying microservices has been widely practised and verified in actual industrial production environments. Many container cloud platforms provide application service running platforms through technologies such as Docker[4] containers and Kubernetes [8]. Kubernetes is an open-source project of Google derived from the internal project Borg[9]. For managing containers, Kubernetes provides a management and orchestration ecosystem, including container management, resource monitoring, log collection, service discovery, and container networking. At present, various cloud computing vendors such as Google, Tencent Cloud, Amazon, and Alibaba Cloud mainly use Kubernetes to orchestrate and manage containers, which greatly reduces the investment of operation and maintenance personnel.

Kubernetes provides horizontal scaling (Horizontal Pod Autoscaling, HPA), which is elastic scaling. It means that the system releases resources during business troughs and expands resources during peak hours. This ensures that the load of the service cluster is always maintained within a reasonable utilization range. Based on certain performance indicators, the application load request changes are judged, and the number of application instances is dynamically adjusted. However, there are certain problems with the existing automatic scaling services of Kubernetes. The original mechanism sets the threshold and scales according to the set speed. When the server access traffic soars, it may cause insufficient expansion resources and interrupt service access. Since the monitored indicators also fluctuate in a short period, only setting the threshold will cause the server cluster to respond frequently, waste resources and increase costs.

For the Kubernetes HPA scheduling policy, Luciano Baresi [10]proposed a new automatic scaling scheme KOSMOS. By developing HPA and VPA components, it overcomes the problem of inconsistent original indicators, and configuring existing containers does not require stopping and restarting. On the cluster, the application is scaled through a heuristic-based controller. Tu Xuezhen[11]designed and proposed a superimposed elastic expansion and contraction system E-HPA to provide rich user-customizable metrics, achieve threshold-based scaling, and design predictive scaling. However, the configuration method is complex and needs to be developed and

978-1-6654-6287-7/22/$31.00 ©2022 IEEE
DOI 10.1109/AIIPCC57291.2022.00051

implemented through other modules. It does not optimize the strategy for HPA.

The above research is suitable for the automation of specific complex scenarios, and there are high requirements for user configuration and component development. In more scenarios, it is necessary to configure scaling strategies that are adaptable and easy to implement. Reasonably formulating HPA custom policies with strong versatility is of great significance to ensure cluster resources' stability and improve resource scheduling's flexibility. Therefore, this paper proposes a strategy of fast response and slow scaling. By setting the expansion rate and tolerance, the flexibility of scaling is increased. When faced with the pressure of a sudden increase in traffic, the speed of rapid capacity expansion is more than twice the default, achieving minute-level optimization. The algorithm is more cautious when scaling down to prevent failures caused by secondary access peaks and ensure the stability of the cluster.

II. HPA DEFAULT ALGORITHM

The full name of Kubernetes HPA is Kubernetes Horizontal Pod Autoscaling. HPA[12] monitors the real-time usage of Pod resources by controlling the controller and essentially adjusts the number of Pod replicas through Deployment and ReplicaSet. The mechanism of HPA is shown in Figure 1.

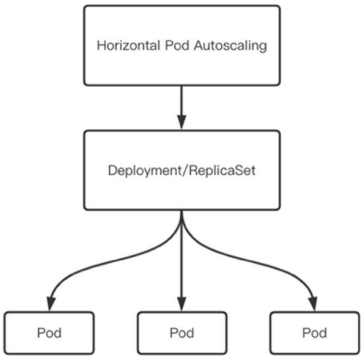

Figure 1. The mechanism of HPA

During the operation of HPA, it will detect the ratio of the resource.

$$\text{DisredPod} = \text{Ceil}[\text{NowPod} * \left(\frac{\text{NowUse}}{\text{DesiredUse}}\right)] \quad (1)$$

DisredPod represents DisredPod Replicas. NowPod represents the Replicas of the current Pod. NowUse represents the current usage of detection indicators. DesiredUse represents the expected usage of the detection indicator. And Ceil returns the smallest integer greater than or equal to the specified expression.

For example, when the current number of replicas is 2, the monitoring indicator is CPU, NowUse is 50m(m means one-thousandth of a core), and DesiredUse is 25m, then the number of replicas will be expanded to 4. Because 2*(50/35) rounded up is 4.

HPA polls every 30s. The number of scaled replicas is further obtained by calculating the ratio of the specified metric usage data to the threshold data for each running replica. During expansion, the number of copies will be doubled every 15 seconds or 4 copies will be added every 15 seconds. Every 15 seconds, currentReplicas*100% of the replicas are reduced at most (but the final replica cannot be less than minReplicas), and the final replica after scaling cannot be lower than the maximum number of historical replicas calculated in the past 300s. To prevent the impact of frequent data fluctuations, the HPA default policy sets a window time. The window time is 3 minutes when scaling up and 5 minutes when scaling down.

The original scaling strategy is not flexible and very simple. When the access of the server increases, it needs to wait for 3 minutes and expands the capacity at a limited speed, which cannot achieve a fast response. The server may not be accessible at this time. Secondly, when scaling down, although there is a 5-minute time window limit, the resource recovery is not careful, and there may be a secondary traffic surge, and the service cannot be accessed again.

III. FAST-RESPONSE&SLOW-SHRINKAGE ALGORITHM

Rapid response, rapid capacity expansion, and careful capacity reduction are the basic requirements of the service in the scenario of a sudden traffic increase. Customizable metrics are also available in HPA. Since usage fluctuations are common, to avoid cluster scaling too frequently, we set the tolerance of the scaling indicator to 0.2. The algorithm sets the expansion speed ScaleUp to adapt to different expansion rates and the cooling down time window for expansion to 0s. When scaling down, the cooldown time of the window has been extended to 9 minutes. HPA also provides other custom indicators to provide the possibility to increase the richness of the strategy.

The Fast-response and Slow-shrinkage algorithm consists of two parts: the Fast-response algorithm and the Slow-shrinkage algorithm.

A. Fast-response Algorithm

Algrithom1: Fast-response Algorithm

```
Inputs:
DesiredUse: Expected usage of the detection indicators.
NowUse: Actual usage of the detection indicators.
maxReplicas: The maximum number of copies to set.
NowPod: The number of replicas of the running pod.
DisredPod: Desired number of replicas for  Pod.
Tolerance: The magnitude of the fluctuation in the ratio.
ScaleUp: speed of expansion.
run Algrithom1
init NowPod = 1
while true do
 if [NowUse /DesiredUse > (1+ Tolerance)] {
   DesiredPod=Ceil[NowPod*( NowUse /DesiredUse)]
   if (DesiredPod < maxReplicas){
    DesiredPod = DesiredPod
   } else{
    DesiredPod = maxReplicas
   }
   While NowPod < DesiredPod do
     NowPod = NowPod *(1+ ScaleUp）
     if DesiredPod < NowPod *(1+ ScaleUp）
       NowPod = DesiredPod
       end while
}
```

Fast-response Algorithm (FA) uses a variety of metrics as input, for example, DesiredUse (Expected usage of the detection indicators), NowUse(Actual usage of the detection indicators), Tolerance(The magnitude of the fluctuation in the ratio), ScaleUp(speed of expansion), etc.

According to the requirements of service, set the initial number of replicas, the maximum number of replicas maxReplicas to 8, and the minimum number of replicas minReplicas to 1, and set the CPU threshold to 60 (the parameter is 0-100, meaning a percentage). Tolerance is set to 0.2 and set the stability window response period to 0s. The ScaleUp is set to 9. The FA expands immediately after being triggered, and the expansion speed is 10 times. The number of Pods is calculated according to Equation 2.

$$N = \left\lceil \frac{\Sigma_{i \in \text{NowPod}} U_i}{DesiredUse} \right\rceil \qquad (2)$$

N represents the number of DisredPod. NowPod represents the existing Pod set. U_i represents the resource usage rate of the Pod, and DesiredUse represents the expected usage rate.

Assuming DesiredUse = 60%. Running 6 copies of the application. The CPU utilization of each Pod is 74%, 78%,80%, 82%, 85%, and 87% respectively. For the next control cycle, the algorithm determines that deployed Pod N = 8. After the expansion, the utilization rate of the Pod can be obtained from Equation 3.

$$\frac{\sum_{i=1}^{N} U_i}{N} = 60.75 \qquad (3)$$

B. Slow-shrinkage Algorithm

When Slow-shrinkage *Algorithm* (SA) performs a scaling operation, it reduces the number of Pods with a step size of 1 until it reaches the required minimum value. When the ratio of NowUse and DesiredUse is lower than 0.48, the scaling operation will be triggered.

Algrithom2: Slow-shrinkage Algorithm

```
Inputs:
DesiredUse: Expected usage of the detection indicators.
NowUse: Actual usage of the detection indicators.
minReplicas: The minimum number of copies to set.
NowPod: The number of replicas of the running pod.
DisredPod: Desired number of replicas for Pod.
Tolerance: The magnitude of the fluctuation in the ratio.
run Algrithom1
init NowPod = 1
while true do
 if [NowUse /DesiredUse < (1- Tolerance)] {
  DesiredPod=Ceil[NowPod*( NowUse /DesiredUse)]
  if (DesiredPod > minReplicas){
   DesiredPod = DesiredPod
  } else{
   DesiredPod = minReplicas
  }
  While NowPod > DesiredPod do
   NowPod = NowPod -1
   if DesiredPod > NowPod -1
    NowPod = DesiredPod
    end while
}
```

Assume DesiredUse = 60%. 6 application replicas are running, and the CPU utilization of each Pod is 48%, 46%, 42%, 50%, 41% and 43%. In the next control cycle, the algorithm determines that deployed Pod N = 5. The utilization rate of the Pod can be obtained from Equation 4

$$\frac{\sum_{i=1}^{N} U_i}{N} = 54 \qquad (4)$$

In the next step, the number of Pods will remain unchanged, and it will take 9 minutes to shrink again.

IV. Experiment and Analysis

A. Experimental Environment

The experimental machine is Lenovo Blade 7000, processor Intel Core I7-9700K, memory 32G, hard disk capacity 1T, operating system Windows10, CentOS7.9, software VMware. This section uses three virtual machines with CentOS 7.9 installed locally to build a Kubernetes cluster. The relevant configurations of the virtual machines are shown in Table 1. The version number of Kubernetes is 1.19.

TABLE 1. CONFIGURATIONS OF THE VIRTUAL MACHINES

Node Name	Attributes	
	CPU （Core）	***Memory （MB）***
Master	4	4096
Node0	2	2048
Node1	2	4096

B. Experimental procedure

1) We use Postman to perform stress testing, observe Pod changes, and verify the expansion and contraction of HPA.

2) We recorded the changes in the number of Pods and made statistical analyses on the data of expansion and contraction.

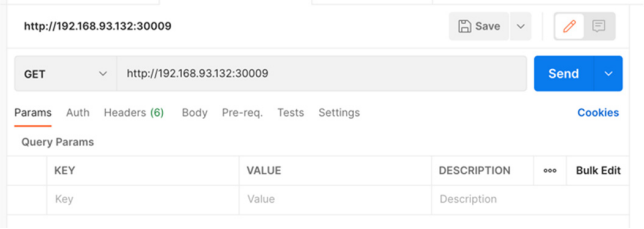

Figure 2. The Dashboard of Postman

As shown in Figure2. We use Postman to configure the IP and port of the service and set it up to automatically send 200,000 HTTP get requests, stress test the nodes running Nginx pods, and verify the HPA mechanism of the custom configuration, that is, the mechanism can respond according to the established parameter settings.

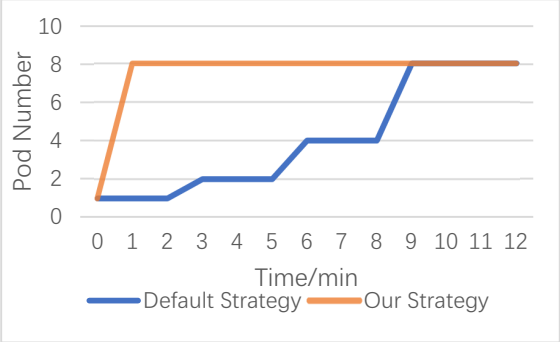

Figure 3. Expansion Pod Comparison

As can be seen from the above figure 3, there are too many service requests to access Nginx, and the capacity expansion is triggered when the service traffic increases. The default strategy is to expand the capacity in a cycle of three minutes. Each time the capacity is expanded, the capacity is expanded to twice the current capacity, and the maximum number of replicas is reached in about 9 minutes. By setting a high expansion speed, our strategy can achieve the maximum number of replicas with only one expansion, which takes only 1 minute, effectively dealing with burst traffic.

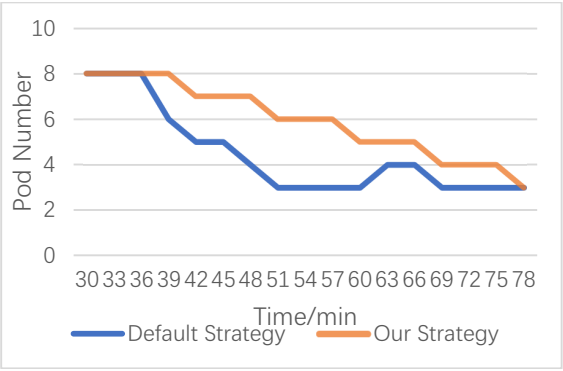

Figure 4. Comparison chart of shrinking Pod

As shown in figure 4, the original strategy is scaled down through a 5-minute window when the scaling operation is performed. Due to careless scaling down, 5 Pods were reduced at 36-51 minutes, resulting in another expansion at 63 minutes. However, our strategy time window was extended to 9 minutes, and a total of 5 shrinkages were experienced, and the number of Pods reduced each time was 1. Compared with the original secondary expansion, our strategy realizes the operation of careful scaling, reduces the expansion and contraction of the system, and ensures the response of the service.

V. Conclusions

The original HPA default algorithm has the problems of high-frequency scaling of resources and poor flexibility. We used a Fast-response and Slow-shrinkage algorithm. The new algorithm can quickly expand the capacity when the traffic suddenly increases, and shrink the capacity carefully after the traffic peak, which ensures the utilization of cluster resources and improves the stability of the cluster. According to the experimental data, in the face of high-traffic usage scenarios, the new algorithm has a faster expansion speed and better ability to deal with emergencies. In actual usage scenarios, appropriate selection of monitoring indicators and selection of multiple indicators will improve the utilization efficiency of cluster resources. Accumulating time series data of indicators and making predictions to allocate resources in advance is also a direction that can be optimized in the future.

Reference

[1] Joy, Mary A. Performance comparison between Linux containers and virtual machines; proceedings of the Computer Engineering & Applications, F, 2015 [C].

[2] Memari N, Hashim S, Samsudin K B. Towards virtual honeynet based on LXC virtualization; proceedings of the Region 10 Symposium, F, 2014 [C].

[3] Zimmer V J, Rasheed Y. HYPERVISOR RUNTIME INTEGRITY SUPPORT [Z]. 2009

[4] Merkel D. Docker [J]. Linux Journal, 2014.

[5] Ander N J, Payam E K, Stephanie H, et al. Container-based bioinformatics with Pachyderm [J]. Bioinformatics, 2018, (5): 5.

[6] Kek P Q. Docker : build, ship, and run any app, anywhere [J]. 2017.

[7] Boettiger C, Boettiger C. Using Docker for portable, scalable & reproducible research [J]. 2016.

[8] Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes [J]. Cloud Computing, IEEE, 2014, 1(3): 81-4.

[9] Wilkes, John, Brewer, et al. Borg, Omega, and Kubernetes [J]. Communications of the Acm, 2016.

[10] Baresi L, Hu D Y X, Quattrocchi G, et al. KOSMOS: Vertical and Horizontal Resource Autoscaling for Kubernetes [Z]. Service-Oriented Computing: 19th International Conference, ICSOC 2021, Virtual Event, November 22-25, 2021, Proceedings. Dubai. 2021: 821-9

[11] Xuezhen T, Haichao Y. Horizontal elastic scaling system based on Kubernetes [J]. Computer and Modernization, 2019, (7): 7.

[12] Phuc1 L H, Phan1 L-A, Kim1 T. Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure [J]. School of Information and Communication Engineering, Chungbuk National University, Cheongju, South Korea, 2022, Vol.10: 18966-77.