An Auto Scaling System for API Gateway Based on Kubernetes

Meina Song, Chengcheng Zhang and Haihong E
School of Computer Science
Beijing University of Posts and Telecommunications, Beijing 100876, China
mnsong@bupt.edu.cn, sweetzcc@163.com, ehaihong@bupt.edu.cn

Abstract- The micro-service approach is a new term in software architecture patterns which is gaining popularity due to its flexibility, granular approach and loosely coupled services [1]. In this paper, first, we present an API Gateway System as the entrance to backend services. It can effectively decrease the amount of remote calls between applications and backend services, and simplify the complexity of internal services caning each other. Secondly this paper designs an auto scaling system for API Gateway System based on Kubernetes and Prometheus which can dynamically adjust the number of applicatlen instances according to its own 10alL It can improve the utilization of system resources while ensuing the high availability and quality of the applicatlon's service.

Keywords-Micro-service, Auto Scaling, API Gateway, Kubernetes

I. INTRODUCTION

The micro-service is a new term in software architecture patterns which divides the complex system into a set of small independent services. Each of the services is running independently in its own process which performs a specific task [2-3]. Services can communicate with each other by a number of lightweight mechanisms such as HTTP, RPC and so on [4].

In the micro-service architecture, the method called within the process become remote procedure calls [5]. In traditional communication pattern, applications directly call the backend services. It is the simplest way. However, a request for an application may call various backend services. The masses of remote calls could cause the delays and impact on customer experience. With the development of business, the backend service may need to be further divided into a set of small independent services. The direct communication pattern could result in much modification in all applications to adapt services' changes. By using API Gateway System as the middle layer between the applications and backend services[6-7], it can hide some limitations and details for applications. For example the changes of backend service are invisible to the applications. But since the API Gateway System is the only entrance to backend services, and it's breaking down can lead to all services unavailable. So high availability of API Gateway services must be guaranteed

The main contributions of this paper are:

- *API Gateway:* we present an API Gateway System which can effectively decrease the amount of remote calls and enable the changes of backend services invisible to applications.
- *Auto Scaling System:* we designed an Auto Scaling System for API Gateway. It can dynamically adjust the number of instances of API Gateway according to its workload, which can improve the utilization of system resources while ensuing the high availability and quality of the API Gateway's service.

The rest of the paper is divided into followings sections. In section II we present the API Gateway as entrance to backend services. The auto scaling System is presented in section III. Tests and Results are presented in section IV. Section V is the conclusion of the paper.

II. THE DESIGN OF API GATEWAY

A. Communications in micro-service architecture

In micro-service architecture, the basic communication pattern consists of two parts: application and backend service, and it is shown in the Fig 1. The backend service perfonns a specific function and exposes the ability with service address. The application is the consumer which call the backend service such as mobile application and web service client.

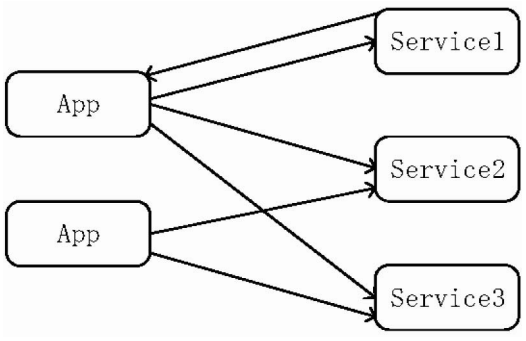

Fig. 1. communication directly.

Directly calling the backend services is the most flexible way for applications. But the application may call various backend services for some particular request which leads to the delays for remote calls [2]. The backend services are usually

978-1-5386-6565-7/18/$31.00 ©2018 IEEE

dynamic based on micro-service architecture. As the growth of business, backend services may need to be further split, which will result in all types of applications need to be modified for adapting services' changes.

We built an API gateway using node 1S and consul [8] as the middle layer between the application and backend services which allows users to invoke the services required without the need for any knowledge pertaining to the structure of the system[7]. This new communication pattern through API gateway has three parts, and is shown in Fig 2.

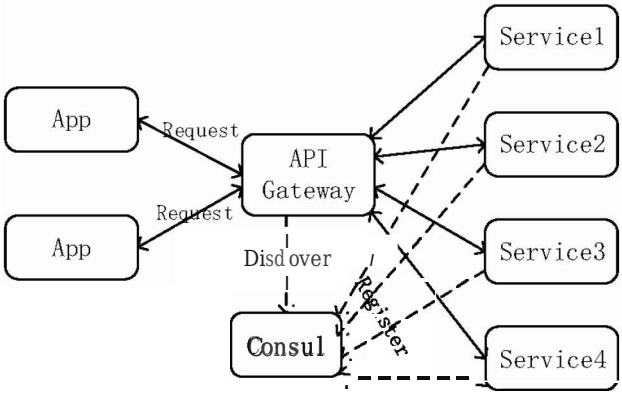

Fig. 2. communication by API gateway.

In this communication pattern, the application and backend service only need to communicate with the API Gateway separately. The backend service will register service address in the consul when it starts. The API gateway receives the application requests and makes a service discovery in the consul , then routes to the backend services, and finally returns the result to the application.

B. The design of API Gateway System

The API Gateway System has three subsystems: Core subsystem, monitor subsystem and admin subsystem. The architecture of API Gateway system is shown in Fig 3.

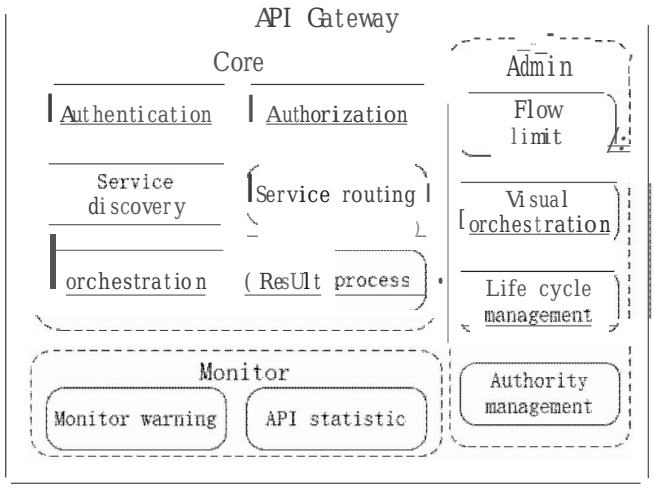

Fig. 3. the architecture of API Gateway.

- *Core subsystem:* It includes several components: user authentication, authorization, service discovery, service orchestration, service routing and result processing. And it is the core function of API Gateway System. Its processing is when receiving a request, firstly authenticates the request, then discovers the service from Consul, at last routes the request to the backend service. Finally the core subsystem processes the result according to the type of application .
- *Monitor subsystem* [9]: It mainly includes monitor warning and API statistics. The main responsibility of monitor subsystem is to monitor the health and performance of the core subsystem, such as the API responses time and the number of API calls and so on.
- *Admin subsystem:* It is mainly about the control and configuration of the core subsystem. And it's mainly functions include flow limit, visual orchestration, API lifecycle management, and privilege management.

This application communicate with backend services by API Gateway System. It can manage all backend services and simplify the complexity of backend services communication.

III. THE DESIGN OF AUTO SCALING SYSTEM BASED ON KUBERNETES AND PROMETIIEUS

In the micro-service architecture, The API Gateway System is the only entrance to backend services, when a large amount of requests invokes API Gateway service concurrently, it could cause the API Gateway service blocking or even breaking down. If API Gateway service breaks down, it makes all services unavailable. To resolve this problem, we use Kubernetes and Promethues to build an auto scaling system to dynamically scale API Gateway System.

A. Kubernetes and Promethues

Kubemetes is an open source cluster manager for Docker containers [10]. It provides management of containerized clusters based on the CAAS (Container as a service) level. The Kubernetes architecture is shown in Fig 4.

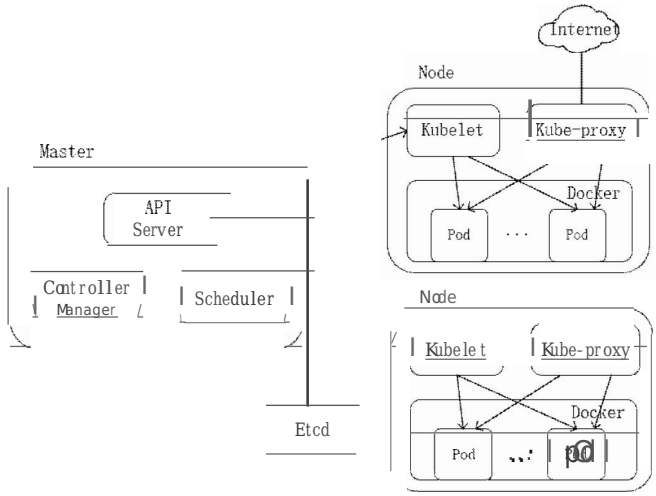

Fig. 4. the architecture of Kubernetes.

In Kubernetes clusters, Pod is the basic deployment unit that can be both operated and managed. It is also native support for monitoring the state of services within pod.

- *Master:* as the control node, it is responsible for the of containers' arrangement and management in cluster. It includes three components. The API server is the entrance to all services, the Scheduler is responsible for scheduling the application to the node for deployment which based on the state of the cluster. The Controller manages the state of the application in the cluster as the desired state that we set.
- *Node:* Node is the working node and responsible for actually running the containers. It includes two components. The Kubelet manages the lifecycle of containers in the cluster, as well as Volume and network management. The Master node manages the containers through Kubelet. Kube-proxy provides service discovery and load balancing within the cluster.
- *Etcd:* The Etcd cluster stores the state of the entire cluster, allowing the other components of the master to monitor the state of the cluster in the etcd to keep the applications in the desired state.

Prometheus is an open source monitoring alarm system by SoundCloud [11]. It uses pull model and HTTP protocol to collect data indicators. As long as the application system can provide the HTTP interface, it can access the monitoring system. Prometheus supports the monitoring of containers and it can be easily deployed to Kubernetes cluster as a monitor system.

B. *The design of Auto scaling system based on Kubernetes and Prometheus*

The Auto scaling system can dynamically adjust the number of application instances according the workload of application. It can ensure the high availability of the application and improve the utilization of the system resources. And its model is shown in Figure 5.

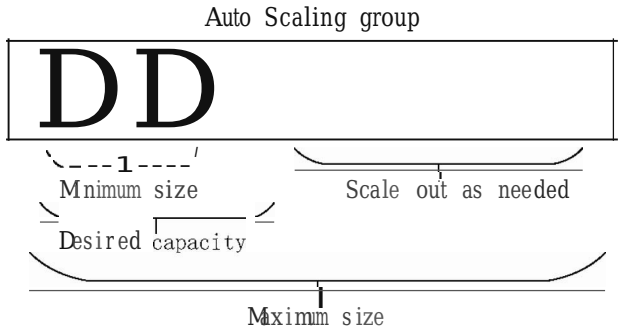

Fig. 5. the auto scaling model.

The maximum and minimum thresholds of the number of API Gateway instances can be specified by the system parameters. When the workload of API Gateway is low, only the minimum number of API gateway instance are deployed. With the workload growing, it may exceed the workload threshold of current active instances. Auto scaling system will scale out as needed to allocate more instances for balancing the load. Meanwhile, the system sets an expected state for each API gateway instance and monitor the real-time status for them. When one of the API gateway instances exits abnormally or breaks down, Auto scaling system will automatically allocate a new instance and move the workload from the unavailable instance to this new instance to make sure the continuity and availability of API server. The architecture of Auto Scaling System based on Kubernetes and Prometheus is shown in Fig 6.

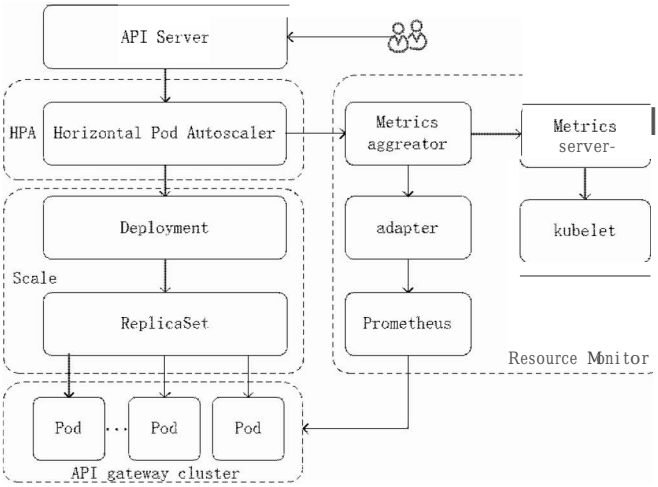

Fig. 6. the architecture of Auto Scaling System.

The System mainly includes four parts, API Gateway cluster, Resource metric monitoring, HPA, Scale.

- *API Gateway cluster:* API Gateway defines and exposes custom monitoring metrics and provides a HTTP interface for Prometheus to collect data periodically.
- *Resource metric monitoring:* It is mainly responsible for getting customer monitoring data from the HTTP interface provided by the API Gateway. The Adapter is responsible for retrieving customer monitoring data from Prometheus and processing and specification of the data for Metric aggregator calls. The metric aggregator combines the data collected by Prometheus and Kubelet for HPA call.
- *HPA:* it gets the monitoring metric data from Metrics Aggregator periodically and compared it with the threshold and calculates the number of instances that need to be adjusted according to the auto scaling algorithm.
- *Scale:* it is responsible for adjusting the actual number instances of API Gateway according to the number calculated by HPA.

The Auto Scaling System step is as follows:

I. The user creates the HPA through API Server, sets the quota utilization rate of the resource that needs reference and sets the maximum and minimum threshold of the number of API Gateway instances.

2. HPA gets the monitoring metric data collected by Prometheme and kubelet from Metric aggregator periodically.

3. Calculate the number of target instances by using the monitoring data compared with the threshold.

4. The target number of instances cannot exceed the maximum and minimum threshold that set in step 1. If it exceeds, it will be forced to the maximum number of instaces, If not, it is expanded to the calculated number of instances.

5. Repeat steps 2 to 4.

IV. EXPRIMENT AND RESULTS

We used five physical machines as our experimental environment, and set up the Kubemeter cluster with one master node and five working nodes in centos system. One master node and one working load is installed in the same physics machine. Kubernetes is 1.9.6 version with 64GB memory and 24 cores for each node. And the version of Docker is 1.13.1.

We deployed two instances of API gateway in the default namespace, and created the HPA of API Gateway with two monitoring metrics the utilization ratio of CPU and QPS. The duration of the experiment is 35 minutes. And we used test programs to simulate user requests, started with a gradual increase in requests from zero, maintained a linear increase in requests, and finally stabilized 110 requests per second. Finally, we stopped the emulator's access to the application. The results are shown in Figure 7.

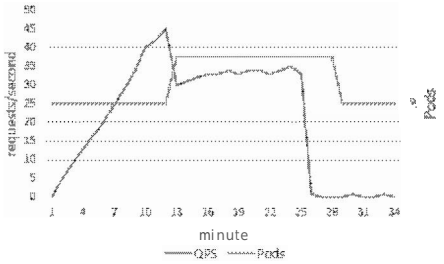

Fig. 7. QPS and the number of Pods.

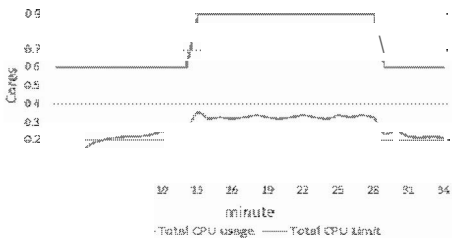

Fig. 8. The usage of CPU.

As shown in Figure 7, when the request load of API Gateway reaches 45 QPS in about 12 minutes, it began to auto scaling, and created an instance. After it expanded, the load of the API Gateway service decreased, then began to grow slowly, and finally stabilized around the 34 QPS. When we stop the test program, it started to shrink, and changed the number of instances from three to two. And the total CPU usage and limit is shown in Figure 8. Over the experiment results, we can see that the Auto Scaling System can dynamically adjust the number of API Gateway instances according to its workload.

V. CONCLUSION

In this paper, we have present an API Gateway System as the entry point for the applications and backend service in micro-service architecture. And We designed an Auto Scaling System based on Kubemetes and Prometheus for the API Gateway System. Then we tested the Auto Scaling System with the test program.

As the experiment shown, after using the API Gateway mechanism, it can decrease the number of remote calls and simplify the complexity for backend services calling each other. The applications and backend services only need to communicate with API Gateway, we can dynamically replace or modify the backend services which is invisible to the applications.

Meanwhile the Auto Scaling System can dynamically adjust the number of the service instances according to the load of API Gateway. It uses the least number of instances to balance the workload when the load is low. When it load exceeds the specified threshold, more instances are created dynamically to balance the workload. It can improve the utilization of system resources while ensuing the high availability and quality of service of the API Gateway.

ACKNOWLEDGMENT

This work is supported by the Ministry of Education-China Mobile Science Research Foundation(MCM20l 7021O); Engineering Research Center of Information Networks, Ministry of Education.

REFERENCES

[1] Singh V, Peddoju S K. Container-based microservice architecture for cloud applications[C]llComputing, Communication and Automation (ICCCA), 2017 International Conference on. IEEE, 2017: 847-852.

[2] Namiot D, Sneps-Sneppe M. On Micro-services Architecture[J]. International Journal of Open Information Technologies, 2014, 2(9):24-27.

[3] Z. Xiao, 1. Wijegunaratne and X. Qiang, "Reflections on SOA and Microservices," 2016 4th International Conference on Enterprise Systems (ES), Melbourne, Australia, 2016, pp. 60-67

[4] Uckelmann, Dieter, Mark Harrison, and Florian Michahelles. "An architectural approach towards the future intemet of things." Architecting the internet of things. Springer Berlin Heidelberg, 2011. 1-24.

[5] Davidson T J, Kelley M T. Method and system for implementing remote procedure calls in a distributed computer system[J]. 1994.

[6] Zhang Q, Chu H, Li M, et al. A unified API gateway for high availability clusters[C]/lMechatronic Sciences, Electric Engineering and Computer (MEC), Proceedings 2013 International Conference on. IEEE, 2013: 2321-2325.

[7] Fan C Y, Ma S P. Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment *Report[C]IIAI* & Mobile Services (AIMS), 2017 IEEE International Conference on. IEEE, 2017: 109-112.

[8] Al-Ayyoub M, Jararweh Y, Daraghmeh M, et al. Multi-agent based dynamic resource provisioning and monitoring for cloud computing systems infrastructure[J]. Cluster Computing, 2015, 18(2):919-932.

[9] Consul. [Online]. Available: https:llgithub.comlhashicorp/consul

[10] Bernstein D. Containers and Cloud: From LXC to Docker to Kubemetes[J]. IEEE Cloud Computing, 2015,1(3):81-84.

[11] Promethes. [Online]. Available: https:llgithub.com/prometheus/prometheus