A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes Cluster

Nguyen Thanh Nguyen, Younghan Kim*
School of Electronic Engineering, Soongsil University, Seoul, Korea
{ntnguyen, younghak}@dcn.ssu.ac.kr

2022 27th Asia Pacific Conference on Communications (APCC) | 978-1-6654-9927-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/APCC55198.2022.9943782

***Abstract*—Nowadays, for multi-tenant cloud, there are plenty of Kubernetes-based open-source projects that do extraordinary in management, isolation multitenancy. These systems allow multiple customers to share the same clusters by dividing them into multiple logical clusters. To satisfy each tenant different workload demand, resource allocation is handled manually by the multi-tenant system administrator. However, this static resource provisioning method cannot adapt to address workload bursting or scaling enough Pod to serve at peak hour and scaling down the number of Pods at least hour. In our paper, we propose a design of dynamic resource allocation in Kubernetes multi-tenancy system to address the missing dynamic resource allocation in the multi-tenant Kubernetes control plane. We start with a controller interacting with the multi-tenant system, which offers dynamic resource allocation, automated scheduling workloads, and policies-based placement. We also present the preliminary results demonstrating the applicability of allocating resources to tenants' applications and services to adapt to workloads changes. Our initial results show fast Pod creation time with the help of a policies-based scheduler and a resources allocator to tenant's applications.**

Keywords—Kubernetes, Kubernetes multi-tenant, cloud management, multicluster, geo-distributed.

I. INTRODUCTION

Due to the dynamic and variability of cloud infrastructures, multi-tenant computing is challenging. This is caused by varying performance, workload, and application characteristics on the cloud. Currently, Kubernetes is mostly recognized as a platform for container orchestration. In addition, a wide range of innovative tools and services have developed around the Kubernetes APIs. Kubernetes have weak multi-tenant because these namespaces could not fully isolate each other, so some multi-tenancy systems have been used to divide physical clusters into multiple instances [1].

Multi-tenancy system allows multiple customers to work on the same Kubernetes clusters. That helps the infrastructure owner save the cost while customers also have the flexibility to use the cloud infrastructure and optimize their budget for their business.

Presently, multi-tenant systems lack the facility of allowing clients to dynamic change their resources based on their business demands or create and allocate resources for new tenants. Multi-tenant system administrator manually does all the work of allocation or changing tenant's resources. Resource allocation is challenging to save cost and maximize utilization of resources while not affecting other tenants [2].

In this paper, we take an approach toward dynamic resources allocation, workload scheduling and deploying a multi-tenant with dynamic scheduling and provisioning based on a multi-tenant system Kubernetes control plane. Our dynamic resource provisioning ensures that the set of provisioned resources is able to meet the demand and requirements of tenancy and not affect the other tenants using available resources.

Our paper is organized as follows: Section II introduces background multi-tenancy cloud and related works. Our architecture and dynamic provisioning models used to represent resources are explained in Section III. Section IV shows the our preliminary implementation. Section V shows the preliminary experimental results. Finally, we present the conclusion and future works in Section VI.

II. BACKGROUND

A. Multi-tenancy Cloud

To achieve maximum utilization of infrastructure resources, one method to use a physical server is divided infrastructure into multiple instances, and each instance is assigned to one tenant - who use resource and infrastructure for their businesses [2]. Multi-tenancy is a principle in software architecture where a single instance of the software runs on a server, serving multiple users/organizations (tenants). Multitenancy contrasts with multi-instance architectures where separate software instances (or hardware systems) operate on behalf of different client organizations [2].

In general, multi-tenancy in Kubernetes clusters falls into two wide categories [1]:

- Multi-tenant Team: This type shares a cluster or multi-cluster between multiple teams within an organization.

- Multi-tenant Customer: This type is strongly isolated between each tenant. The cluster is the separation into multiple instances for tenant purposes. Multi-tenant customer requires isolation of both data plane and control plane. Isolation and cost optimization are the most critical requirements. We only focus on multi-tenant customers in our paper.

There are current multi-tenant Kubernetes projects: KCP [3], Capsule [4] and Kiosk [5]. All these projects deploy multitenancy systems on a physical cluster by dividing a physical cluster into multiple logical clusters. Their approach uses an operator inside each Kubernetes to provide isolation and control plane for each tenant. But with Capsule [4] and Kiosk [5] use a Kubernetes extension to access their resources, while KCP has an API server to provide customers with an easy way to access their resources. In addition, KCP provides an abstraction for the entire infrastructure below that can support advanced features that are hardly implemented on other multi-tenant systems i.e., inheritance, resources allocation, automatic management, etc. Each logical cluster has its own API server control plane and data plane, so it was fully isolated [3]. Two terminologies of KCP for multi-tenant cluster management are logical cluster and workspace - which use for abstraction and advanced features of KCP.

*Corresponding author: Younghan Kim (younghak@dcn.ssu.ac.kr)

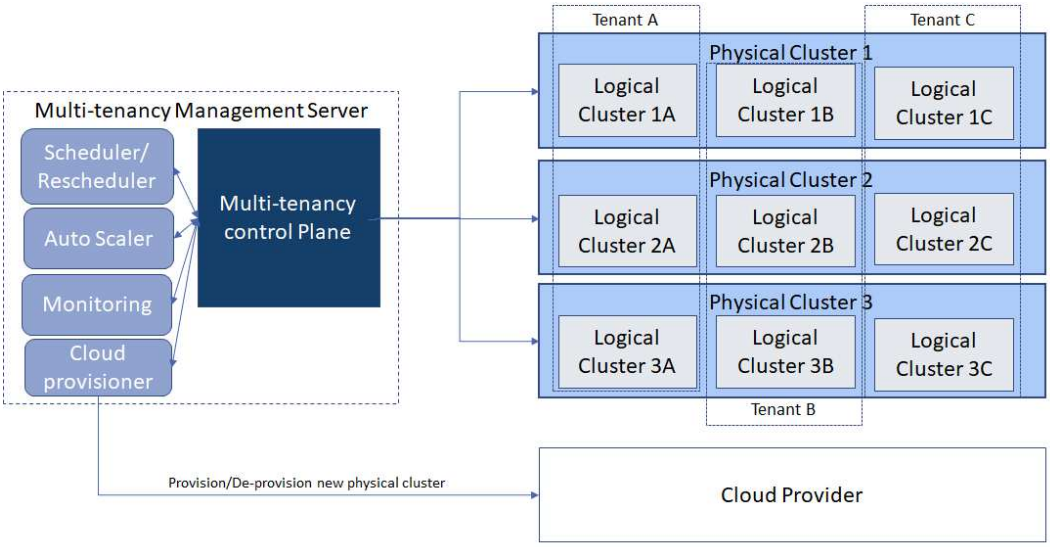

Fig. 1. The design multi-tenancy architecture overview

1) KCP – Minimal True Multi-tenant Kubernetes Control Plane

KCP is a generic Custom Resource Definition (CRD) API-server that is divided into multiple "logical clusters" that enable multitenancy of cluster-scoped resources such as CRDs and Namespaces [3]. Each logical cluster is fully isolated from the others, allowing different teams, workloads, and use cases to live side by side. KCP used two primary ways to share a Kubernetes cluster for multi-tenancy: using Namespaces (i.e., a Namespace per tenant) and virtualizing the control plane (i.e., Virtual control plane per tenant).

a) Logical Clusters

Logical cluster [3] is a terminology of KCP that represent a virtual cluster. From the user's point of view, each logical cluster does not differ from a physical cluster. The one concrete component that cannot be tenanted is API resources - from a Kubernetes perspective. Each logical cluster has its own API resources that store separated in etcd database of KCP server. A logical cluster is created with the lowest cost - nearly zero in terms of resource utilization & cost for an empty cluster.

b) Workspaces

Workspace [3] is the generic representation of the concept to represent a set of user-facing APIs for CRUD. A logical cluster backs each workspace. To a user, a workspace appears to be a Kubernetes cluster minus all the container orchestration-specific resources. Workspace is a part of an organization.

2) Cluster API

Cluster API is a project that provides declarative APIs and tools for provisioning, upgrading, and operating the Kubernetes cluster [6]. Cluster API can connect to cloud providers and expose API to users that enables user can that automate cluster creation, configuration, operation, deployment, and management of their workloads across various infrastructure environments, avoiding vendor lock-in [6,7].

B. Related Works

Recently, several works have utilized the advantages of models or algorithms to schedule and allocate resources in software-as-a-service, cloud environments. Ramachandran et al. [8] implemented a model user interface, which enables tenants can customize their system or dynamic change system resources and requirements. Their model help infrastructure owner and tenants save time and energy in configuring and provision resources. Still, the limitation of [8] is that the user interface only has static configurations and is not based on the business application's demands. Also, using dynamic resources allocation, author of [9] propose a prediction to predict and allocate virtual machine to minimize time provisioning. They used the best-fit algorithm for allocation, but it reveals some cases do not meet the tenant requirement. The approach of [9] is only considered for a dedicated multi-tenant system that provides tenancy on the virtual machine, not Kubernetes cluster.

Meanwhile, Zhiming Shen et al. [10] propose a resource scaling system to automate scale resources to meet tenant application requirements. However, the schemes in [10] apply for a dedicated multi-tenant system based on virtual machines. Besides, [10] has several limitations: complicated thresh hold of requirements and high prediction errors for long-time prediction tasks. Our approach offers an allocation resource mechanism for multi-tenant system based on Kubernetes cluster to satisfy resources demand of tenant's application. We implemented a comprehensive system using a combination of several controllers integrated with KCP that aims to utilize resources, dynamic allocate resources, and adapt to the tenant's application demands.

III. SYSTEM DESIGN

This section introduces the design that represents KCP server, our controllers, logical clusters along with their tenants.

Fig. 1 depicts an overview of system architecture. Main components are a multi-tenancy management server, physical clusters, and cloud provider. Firstly, Multi-tenancy Control Plane is a KCP API-server along with several controllers.

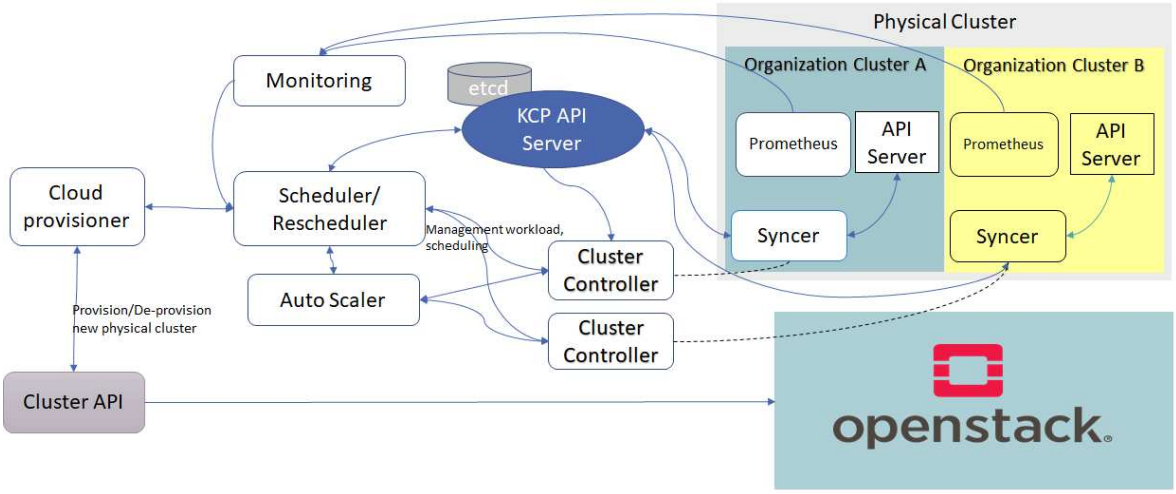

Fig. 2. Detail multi-tenancy architecture with KCP API-server, scheduler, autoscaler and cloud provisioner for dynamic scheduling and provisioning resources

The KCP server is responsible for maintaining connections to physical clusters and managing all logical clusters running on physical clusters.

The scheduler/rescheduler controller interacts with KCP server to control scheduling workloads of tenants across multiple clusters, rescheduling workloads when one or some of the logical clusters got a problem, and workloads are migrated to another logical cluster to ensure no disruption of users. We offer policies-based scheduling workloads across multi-clusters. Based on available resources on multi-cluster, available resources on each cluster, and policies of tenant's application, the scheduler scales tenant's application in the multi-clusters environment that adapts to workload changes and requirements of tenant.

We used Prometheus to monitor and collect performance data from physical clusters. Prometheus collects metrics CPU, memory usage, and network metrics in each tenant [6]. According to the data of Prometheus, the scheduler controller interacts with KCP server to change the resources and the number of pods running. The auto-scaler component monitors tenants' workloads and scales tenant services to meet tenants' requirements.

The cloud provisioner is a controller which offers dynamic provisioning resources to the multi-tenancy control plane. Cloud provisioner has an algorithm based on the system's performance and scheduler to provision new clusters to adapt to changes in tenant's workloads during peak hours. The cloud provisioner directly interacts with Cluster API. Through Cluster API, the cloud provisioner automatically installs the logical cluster's dependencies into the new physical cluster provisioned. After that, the new logical cluster joins the Multi-tenancy system. This time, the scheduler can deploy more Pod to adapt to the tenant's workloads.

Cluster API is connected to cloud infrastructure, and via Cluster API cloud provisioner can send a request to provision a new physical cluster or remove a physical cluster when tenant's workloads are low.

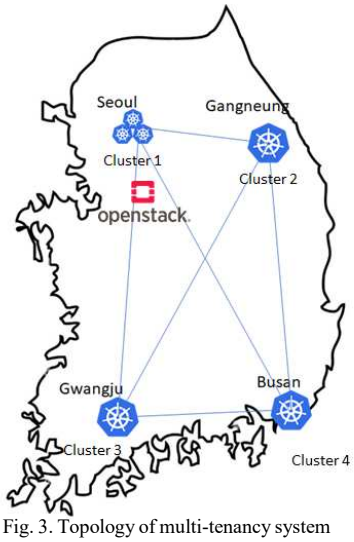

Fig. 3. Topology of multi-tenancy system

IV. Preliminary Implementation

A. Testbed

A multi-tenancy system with dynamic scheduling and provision resources that we implemented is shown in Fig. 3 by its topology.

To create a multi-cluster environment, we use a switch to simulate the environment as Fig. 4. We deploy a Kubernetes cluster with one master and one worker. There are four regions: the Gangeung, Gwangju, and Busan clusters, as well as the Seoul cluster. In order to create a multi-cluster environment, we use a switch to simulate that each cluster is located at these locations.

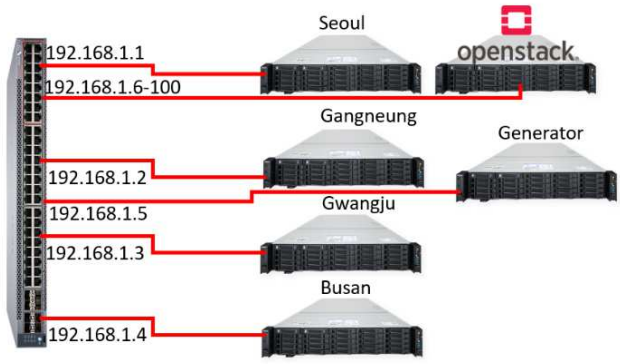

Fig. 4. Simulation of multi-cluster environment for testbed

We deploy a Kubernetes cluster with one master and one worker node. Seoul cluster uses nodes with 4 CPU cores and 16GB RAM, whereas others use nodes with 2 CPU cores and 8GB RAM.

We install Ubuntu server 18.04, Docker, and Kubernetes version 1.20 on each worker node and master node in these clusters.

B. *Testing Scenario*

We evaluate our proposal implementation system with a stateless application with two tenants in each physical cluster. Every physical cluster has two logical clusters for two tenants.

Our system will be tested in two scenarios:

- Scenario 1: Running application without provisioning new clusters, scheduler based on metadata: placement, policies, resources, workloads to deploy more Pod to adapt workload changes.
- Scenario 2: Running an application with a combination of policies-based and provision new cluster to deploy more Pod adapt user traffic bursting.

In scenario 1, we use a generator to generate user traffic to the tenant 1 application and collect information from two tenants' systems using Prometheus. This scenario tests the multi-tenancy system's ability to scale applications.

In scenario 2, we generate user traffic to tenant application to simulate the peak hour- when the user traffic is very high in a short time. The system now increases the number of Pod provisions and joins new clusters to serve the user request and meet the tenant requirements.

V. Preliminary Experiment Result

We represent the initial result of the evaluation of our system with scenario 1 - described in the previous section. The chart in Fig. 5 shows the change in the number of concurrent requests to the tenant's application, the number of Pods needed to deploy to serve user's requests, and the time taken for deploying Pods required (in second). The scheduler based on available resources of multi-cluster and available resources on each cluster, policies of tenant's application to make a decision to deploy more Pod on which cluster. The Pod creation time was calculated based on the Pod creation timestamp and the Pod's ready condition timestamp. We use the Pod configuration resources: CPU 200mili, RAM 200MB, to run a stateless application.

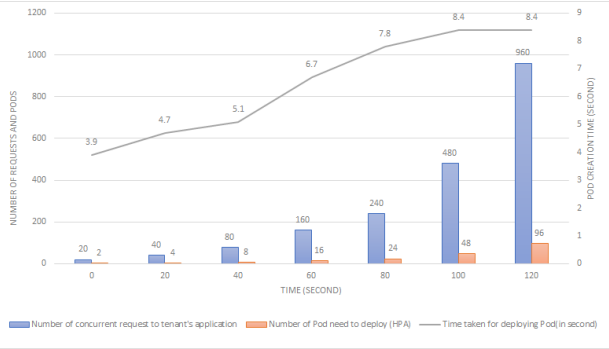

Fig. 5. Pod creation time with different workloads

Two columns demonstrate the number concurrent request and number (light blue color) and number of Pods needed (light pink color). When the concurrent request increase, the autoscaler calculates the number of Pods needed based on concurrent requests and the number of Pods. The creation time is slower when the number of Pods is high. After that, the autoscaler increases the number of Pods needed to serve users' requests.

VI. Conclusion and Future Works

This paper presents a design of multi-tenancy cloud management with dynamic scheduling and provisioning resources that based on Kubernetes and running across multi-cluster to maximize the utilization of resources usage. The system offers a dynamic provision cloud cluster and capabilities to scale multi-cluster applications to adapt to these workload changes even in worst-case cloud bursting. The design includes automatically scheduling workload between logical clusters and policies-based placement.

We show our initial implementation of the system and testing scenario. In the next version of this work, we plan to include our testing results and performance analysis of the system.

Acknowledgment

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. 2020-0-00946, Development of Fast and Automatic Service recovery and Transition software in Hybrid Cloud Environment), and (No.2022-0-01015, Development of Candidate Element Technology for Intelligent 6G Mobile Core Network).

References

[1] Kubernetes Documentation [Online], Available: https://kubernetes.io/, Accessed on: July 20, 2022.

[2] C. Zheng, Q. Zhuang and F. Guo, "A Multi-Tenant Framework for Cloud Container Services," in 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), DC, USA, 2021 pp. 359-369.

[3] KCP – Multi-tenancy Kubernetes control plane [Online]. Available: https://github.com/kcp-dev/kcp (Accessed: July 26, 2022)

[4] Capsule Documentation [Online]. Available: https://capsule.clastix.io/. Accessed on: July 20, 2022

[5] Kiosk Documentation [Online]. Available: https://github.com/loft-sh/kiosk. Accessed on: July 20, 2022

[6] Cluster API. Available [Online]: https://github.com/kubernetes-sigs/cluster-api. Accessed on: July 20, 2022

[7] OpenStack Documentation [Online]. Available: https://docs.openstack.org/yoga/. Accessed on: August 1, 2022

[8] Ramachandran, L., Narendra, N.C. & Ponnalagu, K. "Dynamic provisioning in multi-tenant service clouds". SOCA 6, 283–302 (2012)

[9] Verma, M., Gangadharan, G. R., Narendra, N. C., Vadlamani, R., Inamdar, V., Ramachandran, "Dynamic resource demand prediction and allocation in multi-tenant service clouds" in Concurrency And Computation: Practice And Experience, 28(17), 4429-4442.

[10] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. "Cloudscale: elastic resource scaling for multi-tenant cloud systems". In Proc. of the 2nd ACM Symposium on Cloud Computing (SOCC). Cascais, Portugal, 2011