System Operator: A Tool for System Management in Kubernetes Clusters

Jiye Yu
Services Computing Research Dept.
Center for Technology Innovation - Digital Technology
Hitachi, Ltd. R&D Group
Email: jiye.yu.kb@hitachi.com

Yuki Naganuma
Services Computing Research Dept.
Center for Technology Innovation - Digital Technology
Hitachi, Ltd. R&D Group
Email: yuki.naganuma.mk@hitachi.com

Takaya Ide
Services Computing Research Dept.
Center for Technology Innovation - Digital Technology
Hitachi, Ltd. R&D Group
Email: takaya.ide.ap@hitachi.com

***Abstract*—Kubernetes is the most popular container orchestration system for automating application deployment. To adapt thousands of applications' working pattern, Kubernetes Operators are proposed as the default approach for packaging, deploying and managing an application in Kubernetes. Now, different kinds of Operators are developed to support applications in various categories. However, a single Operator only applies to a single application. Users still need to pay effort to deploy, monitor or maintain a system which is formed by a class of applications. Thus, Our System Operator is created to provide a help. It is able to connect applications in Kubernetes by connecting applications' Operator in Graphic User Interface (GUI) canvas. Instead of users, System Operator can help maintain the whole system according to organized pattern. It will be a great help for the flexible utilization of Kubernetes.**

Keywords–Container; Kubernetes; Kubernetes Operator; System Operator

I. INTRODUCTION

Virtual machines used to be one of the best options when companies deploy their services. At that time, OpenStack and Amazon Web Services (AWS)) are famous for its stable virtual server quality. During that period, a large number of software applications are designed in monolithic architecture pattern [1]. Monolithic architecture pattern, trending to integrate all components in one server, tends to cause problems like long building time, poor resilience to failures, incompatibility issues. Then, container [2] is invented to be a new form of operating system virtualization. Kubernetes [3], an open source container platform, is raised by Google to help manage containers and automates many of the manual processes in container's deployment and management. Because of its convenience and powerful advantages, now Kubernetes is becoming the most popular container orchestration tool in IT industry.

Along with the benefits Kubernetes brings to us, it also introduces some new issues. Due to the abundant functionality of Kubernetes, in order to get qualified as a Kubernetes engineer, meticulous training is commonly required. Engineers who are not familiar with infrastructure technology will feel it difficult to try Kubernetes because of its complexity [4].

On the other side, concept Kubernetes Operator [5] raised by CoreOS is designed for packaging, deploying and managing a Kubernetes application automatically. With Kubernetes Operators, people are able to share their knowledge on application management, as well as save effort and time on DevOps. Operators bring convenience to Kubernetes users. However, users still need to deploy Operators by themselves.

With Kubernetes and Operator, more and more companies trend to migrate their legacy systems to modern architecture. However, lack of specialized knowledge becomes a barrier for the migration.

Legacy system requires system management as a entirety, while Kubernetes allows container-specific management of distributed system. Even with Kubernetes Operator's help, engineers need to deploy individual applications and connect them to build an entire system. Engineers who are used to legacy management mode need to try hard to break down barriers.

In the other hand, the Cloud Native Ecosystems like Cloud Native Computing Foundation (CNCF) Cloud Native Interactive Landscape [6] and OperatorHub [7] obtain favorable development. Containers based applications and related Operators are developed and released as Open Source Software (OSS), which are available to anyone. This is a great benefit to develop our proposal, System Operator.

The remaining of this paper is organized as follows. Section 2 gives a brief introduction of System Operator. Section 3 explains how System Operator works and what System Operator can be used to do. Finally, in section 4, we make the conclusion, and list out our future work at the same time.

II. SYSTEM OPERATOR

In order to solve above issues, we designed a new tool, System Operator. System Operator is used to create and maintain systems which are composed by various applications.

System Operator is designed to meet following targets:

1) Easy to use: reduce the requirements on user's knowledge on Kubernetes.
2) High applicability: by choosing proper Application Operators, users are able to create all appropriate systems they want.
3) Expandability: users can apply their own configs to System Operator to achieve their requirements on system maintenance.

The complete workflow of System Operator will be divided into several steps. First, users need to select all necessary Application Operators, connect and configure them to make up a system. Then, System Operator will automatically generate

 ISBN: 978-1-61208-778-8

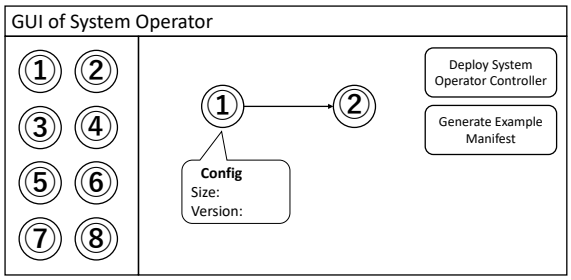

Figure 1. GUI of System Operator

the deployment manifest. Using the generated manifest, users can easily deploy the system to target Kubernetes cluster.

Besides deployment, System Operator can also continuously monitor all resources created by these Application Operators. According to the result of monitoring, adjustment will be executed to keep the system stable. To achieve this aim, there is an issue needs to be solved first. Application Operator is always designed as a black box. Only part of the Application Operators are providing the API for calling corresponding resource's status. Due to this reason, it is not easy for System Operator to get status of resources created by selected Application Operators. Detailed information will be introduced in next section "How System Operator works".

A. Graphic User Interface (GUI)

GUI support is important on improving the usability of this tool. Inspired by OpenStack Heat Dashboard [8], GUI of the System Operator is designed as Figure 1. Various kinds of Operator icons are listed on the left side. Users need to select Application Operators with different functions from the left-hand column. Then, drag and drop selected Operator icons to the canvas on the right side. After dragging and dropping Operator icons, users can connect these icons to build the skeleton of their target system. The line used to connect two icons is a directed arrow. An arrow (x, y) is considered to be directed from icon x to icon y, another arrow (y, z) is considered to be directed from icon y to icon z. Thus, icon y is a previous node of icon z, and icon x is also previous node of icon z. These arrows are used to arrange the network traffic in target system.

By connecting the Operators, the skeleton of system will be presented in the canvas. In order to generate the manifest, users are also required to fill the config for each Operator. System Operator controller which is used to manage these selected Operators in Kuberntes cluster will also be deployed by clicking the button on canvas.

III. How System Operator works

System Operator is used to manage the Application Operators instead of human. By monitoring the status of resources, System Operator can make rapid reaction when sudden events happen. In the following subsections, we will introduce how System Operator works from brief architecture to details.

A. Architecture of System Operator

Figure 2 shows a brief architecture of System Operator as well as the steps System Operator will do to deploy a system in the Kubernetes cluster.

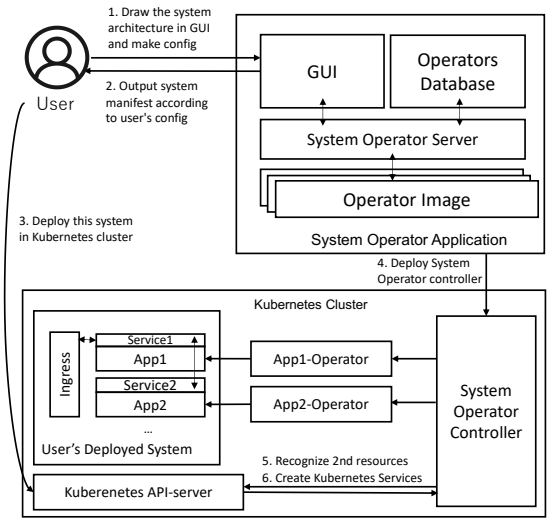

Figure 2. Architecture of System Operator

First, System Operator Application will provide a GUI for accepting user's request, a database and a storage to store the information for Application Operators and Application Operator images. After System Operator accepting user's request (step 1), it will generate a manifest and respond to users (step 2). After deploying of the manifest (step 3), System Operator Application will also deploy corresponding System Operator controller in the same Kubernetes cluster (step 4) to maintain the deployed system. Then, System Operator controller will call Kubernetes API server to recognize all secondary resources of each Application (step 5). At last, System Operator controller in Kubernetes will complete the user's system deployment in Kubernetes cluster by connecting all applications deployed by adding Kubernetes Services among them (step 6). Detailed description will be introduced in following subsections.

B. Use Kubernetes Service resource to connect applications

In normal cases, an integrated system is composed by several applications. In legacy system, engineers use IP address or hostname to make the connection for integrated system. In Kubernetes cluster, in order to generate invariable cluster IP, System Operator will use Kubernetes Service resource to bind applications and make the connection. In order to bind Kubernetes Services with application resources, we need to recognize these application resources first. Not only the Custom Resource [9] defined by Custom Resource Definition (CRD), but also the secondary resources of those CRD resources.

C. Secondary resources

Secondary resources are defined as resources created by Application Operator and managed by Operator's CRD resources. Taking Nginx Operator as an example, Nginx is the CRD resource and deployment created by Nginx is its secondary resource. For keeping the active status of CRD resources, the status of secondary resources is important. Also, we need to bind Kubernetes Service resources to secondary resources in order to integrate the whole system. That is the reason why we need to recognize all applications' the secondary resources.

 ISBN: 978-1-61208-778-8

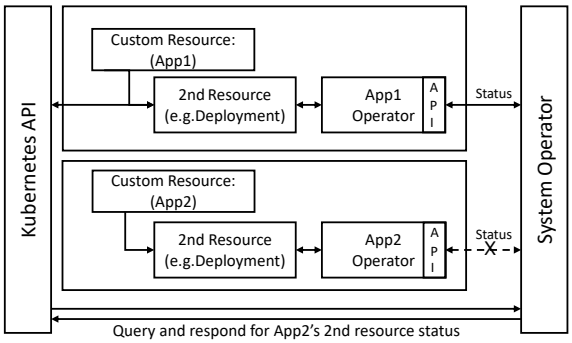

Figure 3. Not all Application Operators provide API for status of secondary resources

D. How to recognize secondary resources

Basically, an Application Operator is designed as black box. Only a few APIs are provided by an Application Operator to communicate with users. In most cases, Application Operator will provide the API to show its secondary resources' status. However, there is no assurance that every Application Operator provides this feature. As Figure 3 shows, without these specific APIs, it is hard for System Operator to recognize secondary resources of Application Operators. System Operator needs to first recognize secondary resources for those Application Operators. Then, System Operator can query secondary resources' status by calling Kubernetes API directly.

As a solution, System Operator can utilize the name and created time of secondary resource to do the reverse inference. In order to accelerate this process, System Operator will maintain a database to record all resource's information collected after manifest applied. Items like resource name, resource type, created time and current status will be recorded in the database. We suppose that there are n Resources in this Kubernetes cluster $(R_1, ..., R_n)$, and in our target system, there are m Custom Resources are deployed $(C_1, ..., C_m)$. What we want to do is to select resources which belong to specified Custom Resource. According to the information recorded in the database, we use following evaluation methods to evaluate the belonging of these secondary resources.

1) Time period evaluation: We define the interval between resource created time and manifest applied time as α. First, we should note that in various Kubernetes clusters, time used to create a resource is not fixed. That means α in various Kubernetes clusters is not a constant. It depends on the transmission delay, computing capability of the hosts and some other factors. In order to keep the accuracy of this evaluation, we need to eliminate this interference caused by the variable α. System Operator will first do dry run several times to create several mock resources. By recording the time difference every time, System Operator can calculate the average value as α in each specific Kubernetes cluster.

Resources whose creating time is close to (manifest applied time + α) tend to be real secondary resources. Either too early or too late will reduce the possibility. To emphasize this characteristic, we can use exponential function to make this evaluation:

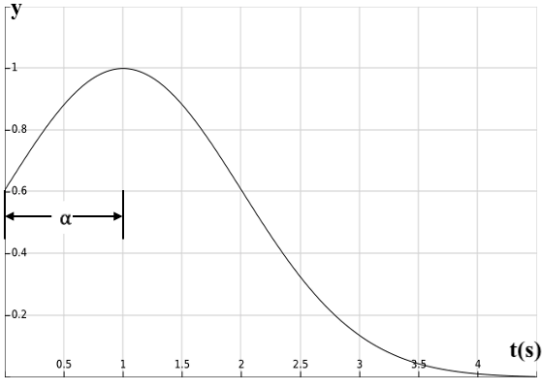

Figure 4. Graph of the time period based evaluation function, suppose $\alpha = 1(s)$

$$y_{ij} = e^{-\frac{(t_{ij}-\alpha)^2}{2}}, \quad (t_{ij} \geq 0) \tag{1}$$

Figure 4 shows the graph of this function. Here, t_{ij} is the difference between Resource R_i's created time and manifest of Custom Resource C_j's applied time. When t_{ij} is equal to α, the evaluation will meet the largest value: 1.

2) Name and label mapping evaluation: According to the convention that the resource name should include the CRD type name as much as possible. We can set the second evaluation equation as follows:

$$x_{i,j} = \frac{(l^1_{i,j} + l^2_{i,j})^2}{4L_j^2}, \quad (l^1_{i,j}, l^2_{i,j} \leq L) \tag{2}$$

Here L_j is the character length of CRD type name of Custom Resource C_j; $l^1_{i,j}$ is the matched character length between Resource R_i's name and Custom Resource C_j's CRD type name; $l^2_{i,j}$ is the matched character length between Resource R_i's label and Custom Resource C_j's CRD type name.

For instance, there are two resources R_1 and R_2, whose names are example-keycloak (R_1) and example-kafka (R_2). R_1's label is $instance = example - keycloak$ while R_2's label is $instance = example - kafka$. Then, comparing with Custom Resource C_1 Keycloak, we can count that $L_1 = 8$, $l^1_{1,1} = l^2_{1,1} = 8$, $l^1_{2,1} = l^2_{2,1} = 1$. Then, finally calculate the value $x_{1,1} = 1$, $x_{2,1} = \frac{1}{64}$.

3) Joint Evaluation: An much more exact and rational result can be obtained by joint optimization of (1) and (2). We can get the possibility of the belonging of each resource is shown as following:

$$r_{i,j} = x_{i,j} \cdot y_{i,j} \tag{3}$$

For each Resource R_i, we can find the Custom Resource C_j to meet the largest value. Then, we can find out the Custom Resource which Resource R_i belongs to by following equation:

$$\arg \max_{j \in [1,M]} \{r_{i,j}\} \tag{4}$$

 ISBN: 978-1-61208-778-8

We should note that some resources already exist before target system's deployment. That means not every Resource R_i belongs to some Custom Resource C_j. For this reason, we should set an threshold θ_j for each Custom Resource C_j to cut those confusing resources. For a Resource R_i, if its max possibility value $\max_{j\in[1,M]}\{r_{i,j}\} < \theta_j$, we can confirm that this Resource R_i does not belong to any Custom Resource C_j.

Regarding to the estimation of the variable θ_j, there are several algorithms to determine the threshold automatically. Since the possibility $r_(i,j) \in [0,1]$, some automatic image thresholding algorithms like Otsu's method [10] can be applied here for threshold determining.

4) Reversing verification: After matching Custom Resources and secondary resources, System Operator needs to do the reversing verification to confirm this resource recognition is correct. Check the network traffic between neighbor secondary resources to verify the secondary resources recognition is a good method. Since System Operator already knows the connection of Custom Resources and traffic according to user's definition, the data stream used for testing should be able to pass through all related secondary resources in turn.

After secondary resources recognition, System Operator can get the status of recognized secondary resources by calling Kubernetes API directly. System Operator will do the regular polling to watch all related resources' status and update them in its internal database.

E. System regulation

Besides system deployment, another function of System Operator is system regulation. In current stage, we have designed two application scenarios for System Operator to handle the whole system. System Operator will do something when:

1) Error happens on secondary resource.
2) Upgrade is executed.

System Operator will watch the status of all secondary resources and make rapid reactions to any changes on the system level. Once unhealthy status happens on any secondary resource, System Operator should send a "stop signal" to all previous Operators of the error one. Here the sequential order of Application Operator can be decided by directed arrows connected in GUI by users.

By default, the Operator received "stop signal" will do nothing, and users can define what should the Operator do properly according to Kubernetes resource ConfigMap. On the other hand, when the unhealthy status recover, System Operator will also send another signal "recover signal" to previous Operators to tell them issue settled. After receiving "recover signal", Operators which have made any changes will revert to the original status.

Similar to the case error happening, when users are making upgrade to some application through Application Operator, System Operator will detect this upgrade action and send stop signal to previous Application Operators. After upgrade, recover signal will be sent as well.

Figure 5 shows an example for application upgrade case. When users applied an updated manifest to upgrade App2, System Operator can easily detect the status change on App2's resource. Then, a stop signal will be sent to the previous node (App1). After completing the upgrade, recover signal will be sent to previous node to end this upgrade process.

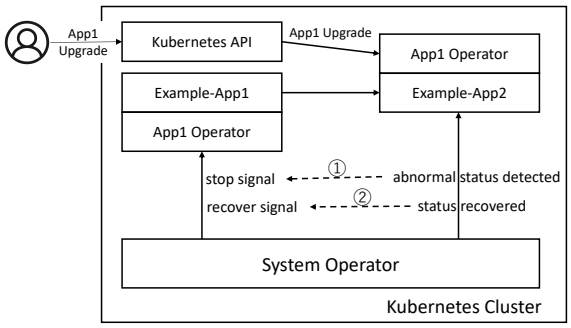

Figure 5. System regulation illustration

IV. Conclusion

In conclusion, System Operator allows Kubernetes users to design their own integrated system by connecting applications with various functionalities and provide simple approach for deploying this system in Kubernetes cluster, as well as subsequent operations. With System Operator's help, the difficulty of using Kubernetes will be greatly reduced.

We believe that System Operator is a promising project to develop and operate application system in Kubernetes clusters. Currently, we just proposed a basic prototype for it. Details and part of concept are still working in progress. For example, by utilizing Prometheus Operator and Prometheus third-party exporters, we can definitely enhance the monitoring feature for System Operator. Mature and well-tested system "recipe" can be spread among users for efficient system construction. System Operator can be more powerful and useful than it seems.

References

[1] M. Mosleh, K. Dalili, and B. Heydari, "Distributed or monolithic? a computational architecture decision framework," IEEE Systems journal, vol. 12, no. 1, 2016, pp. 125–136.

[2] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," IEEE Transactions on Cloud Computing, vol. 7, no. 3, 2019, pp. 677–692.

[3] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," IEEE Cloud Computing, vol. 1, no. 3, 2014, pp. 81–84.

[4] "Kubernetes: Advantages and Disadvantages - The Business Perspective," 2019, URL: https://devspace.cloud/blog/2019/10/31/advantages-and-disadvantages-of-kubernetes [retrieved: July, 2020].

[5] "Introducing Operators: Putting Operational Knowledge into Software," 2016, URL: https://coreos.com/blog/introducing-operators.html [retrieved: July, 2020].

[6] "CNCF Cloud Native Interactive Landscape," URL: https://landscape.cncf.io/ [retrieved: July, 2020].

[7] "Welcome to OperatorHub.io, a new home for the Kubernetes community to share Operators." URL: https://operatorhub.io/ [retrieved: July, 2020].

[8] "OpenStack Documentation: Welcome to Heat Dashboard!" URL: https://docs.openstack.org/heat-dashboard/latest/ [retrieved: July, 2020].

[9] "Kubernetes Documentation, Concepts: Custom Reources," URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ [retrieved: July, 2020].

[10] N. Otsu, "A threshold selection method from gray-level histograms," IEEE transactions on systems, man, and cybernetics, vol. 9, no. 1, 1979, pp. 62–66.

 ISBN: 978-1-61208-778-8