Implementation of Backup Resource Management Controller for Reliable Function Allocation in Kubernetes

Mengfei Zhu, Rui Kang, Fujun He, and Eiji Oki
Graduate School of Informatics, Kyoto University, Kyoto, Japan

2021 IEEE 7th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0522-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/NetSoft51509.2021.9492724

***Abstract*—Resource allocation and management is a key role in network function virtualization to improve the reliability of network services. Kubernetes is a system to deploy and manage the virtual network functions automatically. Existing tools in Kubernetes does not provide a resource type to define the backup Pods. It does not provide automatic resource management based on the user requests for the backup Pods, either. This paper designs and implements a custom resource and the corresponding controller in Kubernetes to manage the primary and backup resources of network functions. The custom resource is a set of Pods with different types, which includes primary, hot backup, and cold backup Pods. The controller manages the set of Pods and maintains the current state of the different types of Pods to keep the current state consistent with the desired state of each type of Pod. Demonstration validates that the controller automatically manage the primary and backups resources correctly.**

***Index Terms*—Network function allocation, backup resource management, Kubernetes, implementation**

I. Introduction

Network virtualization decouples the hardware from the functions so that multiple functions can run on the same node. Network functions are usually packaged in virtual machines or containers. Kubernetes [1] is an open-source system for automating deployment, scaling, and management of containerized functions. A Pod is the smallest deployable unit of computing that can be created and managed in Kubernetes. The controller of a resource in Kubernetes adjusts the current state to the expected state through the control loop [2].

Backup resource management plays a key role in network function virtualization since a network function may fail due to hardware failures, overloads, configuration errors, and service interruption [3]. Backup resources are used to improve the ability of a function to continuously deliver services in the presence of failures and other undesired unavailability by adopting different strategies: cold backup (CB) and hot backup (HB). The backup resource protected with the CB strategy is only reserved without being activated; the backup resource protected with the HB strategy is activated and synchronized with the primary resource before any failure occurs [4].

The costs for resource management and maintenance account for a large part of the entire life cycle of a network software. Considering that a shorter expected unavailable time for recovery can improve the user experience, the works in [4], [5] adopted the two strategies for primary and backup resource allocation. The resource allocation model can be extended to handle the dynamic reconfiguration based on the requirement. Taking a dynamic reconfiguration triggered by a failure as an example, one of the backup resources protecting the failed function needs to be activated for recovery. A new backup resource needs to be allocated to replace the activated one which is used for the recovery of the failed primary function for higher availability. In order to adapt the dynamic reconfiguration requirement, the backup resource management considering the dynamic reconfiguration is required.

This work was supported in part by JSPS KAKENHI, Japan, under Grant Number 18H03230.

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

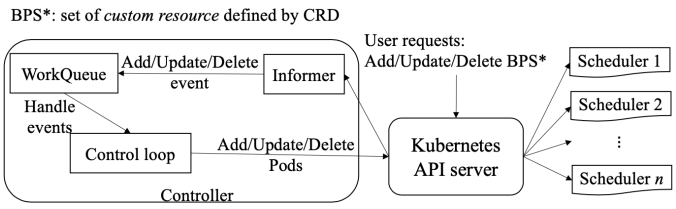

Fig. 1. Overall structure.

However, Kubenetes does not provide a resource type to define the backup Pods with considering the different backup strategies. In addition, it does not provide automatic resource management based on user requests for the backup Pods.

This paper designs and implements a controller to manage the primary and backup resources of network functions. We introduce a new resource type called *backup Pod set (BPS)*, which is a *custom resource* in Kubernetes [6]. *BPS* includes a certain number of different types of Pods which are the primary, HB, and CB Pods. The transitions of different types of Pods can be customized by cooperating with the allocation-model-based scheduler introduced in [7] or randomly. Demonstrations validate the effectiveness of the controller.

II. Design and Implementation

A. Overall structure

Figure 1 overviews the structure of the reported controller and the cooperation between the controller and Kubernetes application programming interface (API) server. The controller handles events triggered by BPS instance operations; BPS is defined by custom resource definitions (CRD) [6].

When a BPS instance is requested to be created, updated, or deleted, *Informer*, which is a bridge between the API server and the controller, receives the notification from the API server and pushes the events caused by BPS instance operations to a First-In, First-Out (FIFO) queue called *WorkQueue*. *Control loop* is the core of the controller, which handles the events in *WorkQueue* by maintaining the current state of a BPS

Authorized licensed use limited to: Gheorghe Asachi Technical University of Ia¿i. Downloaded on December 14,2023 at 07:16:24 UTC from IEEE Xplore. Restrictions apply.

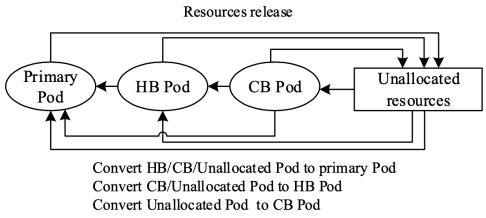

Fig. 2. Pod state transition diagram.

instance until it is consistent with the desired state. *Control loop* maintains the number of Pods for each type by resource releasing and converting, until the number of Pods for each type is consistent with the desired number requested by the user requirement, as shown in Fig. 2. The scheduler decides the allocations of the newly created Pods with the default scheduler or allocation-model-based scheduler in [7].

B. Backup Pod set (BPS)

A BPS instance is a *custom resource* defined by CRD. CRD includes the required information and specifications for creating a BPS instance with three parts: *Metadata, Spec,* and *Status*. *Metadata* provides descriptive information of a BPS instance including name and creation time. *Spec* provides the specifications and desired state of a BPS instance including the desired numbers of the different types of Pods, the strategies for state transitions, and the name of the specified scheduler. *Status* contains the current state of a BPS instance, including the current numbers and names of the Pods. *Status* is updated periodically in the control loop.

Primary, HB, and CB Pods are distinguished by the labels, each of which is attached to the Pod when a service is created. Each Pod with a *primary* label is used to balance the traffic for services. Each Pod with a *hot backup* label is activated without being exposed to the service; the Pod takes over the tasks of primary Pod once a failure is detected. When a Pod with a *cold backup* label is requested to be created, we add an *init container* [8] in the CB Pod. *Init containers* run before the other containers in the same Pod. Each *init container* must be completed successfully before the activation of other containers in the same Pod. The *init containers* in this demonstration is implemented as a transmission control protocol (TCP) server, which is used for waiting for the activation message from the control loop.

C. Control loop

The control loop handles the add, delete, and update events of a BPS instance, as shown in Algorithm 1.

Algorithm 1 Control loop

```
1: Update Status defined in CRD
2: if the instance is newly created then
3:     Create desired numbers of primary/HB/CB Pods
4: else
5:     if Current primary Pods < desired primary Pods then
6:         Convert an HB Pod to the primary Pod until there is no more HB
   Pods or the primary Pods reach the desired number.
7:         Convert a CB Pod to the primary Pod until there is no more CB
   Pods or the primary Pods reach the desired number.
8:         Create a primary Pod until the desired number of primary Pods.
9:     else if Current primary Pods > desired primary Pods then
10:        Delete a primary Pod until the desired number of primary Pods.
11:    end if
12:    if Current primary Pods = desired primary Pods then
13:        if Current HB Pods < desired HB Pods then
14:            Convert a CB Pod to the HB Pod until there is no more CB
   Pods or the HB Pods reach the desired number.
15:            Create HB Pod until the desired number of HB Pods.
16:        else if Current HB Pods > desired HB Pods then
17:            Delete HB Pod until the desired number of HB Pods.
18:        end if
19:        if Current HB Pods = desired HB Pods then
20:            if Current CB Pods < desired CB Pods then
21:                Create a CB Pod until the desired number of CB Pods.
22:            else if Current CB Pods > desired CB Pods then
23:                Delete a CB Pod until the desired number of CB Pods.
24:            end if
25:        end if
26:    end if
27: end if
28: Requeue in WorkQueue
```

When a BPS operation is requested, the desired numbers of Pods for different types are changed. The control loop focuses on the comparison between the desired number and the current number of each type of Pods. The control loop automatically maintains the current BPS state until it reaches the desired state by following the transition of different types of Pods with resource releasing and converting shown in Fig. 2. Note that, for a different conversion unavailable time, the resource conversion has a priority policy. Compared with the Pods with longer unavailable time, the Pods with shorter time are prioritized to be converted. For example, the conversion of the HB Pod to the primary Pod has higher priority than the conversion of the CB Pod to the primary Pod. A candidate of the same type of Pods can be chosen based on a given policy; it is chosen randomly in this paper.

When the current number of the primary/CB/HB Pods is larger than the desired number, each unnecessary primary/CB/HB Pod with the number of the difference between the desired number and the current number of primary/CB/HB Pods is selected by the given strategy and deleted; the occupied resources of the unnecessary Pods are released.

When the current number of primary Pods is smaller than the desired number, firstly, the current HB Pod is converted to the primary Pod and exposed to the service. When all the HB Pods are converted and the number of current primary Pods is still less than the desired number, secondly, the current CB Pod is converted to the primary Pod with being activated and exposed to the service. When all the CB Pods are converted to the primary Pods and the number of current primary Pods is still less than the desired number, the remaining primary Pods are created with using unallocated resources.

When the current number of the HB Pods is smaller than the desired number, firstly, the current CB Pod is converted to the HB Pod with being activated by the *init container*. When all the CB Pods are converted and the number of current HB Pods is still less than the desired number, the remaining HB Pods are created with using unallocated resources.

When the current number of the CB Pods is smaller than the desired number, the insufficient CB Pod is created following a given strategy. The resource of the CB Pod is reserved without activated; they wait for being activated by the *init container*.

Authorized licensed use limited to: Gheorghe Asachi Technical University of Ia¿i. Downloaded on December 14,2023 at 07:16:24 UTC from IEEE Xplore. Restrictions apply.

```
apiVersion: backup.example.com/v1alpha1
kind: BackupPodSet
metadata:
  name: bps-sample
spec:
  replicas: 2
  hotBackups: 2
  coldBackups: 2
  strategy:
    init: "random"
    unallocatedToColdbackup: "random"
    unallocatedToHotbackup: "random"
    unallocatedToReplicas: "random"
    coldbackupToHotbackup: "random"
    coldbackupToReplicas: "random"
    hotbackupToReplicas: "random"
  schedulerName: "default-scheduler"
  wakeupTimeout: 5
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
```

Fig. 3. Configuration file of a BPS instance.

```
ubuntu@ubuntu:~/BackupResourcesController_k8s$ kubectl apply -f deploy_sample.yaml
backuppodset.backup.example.com/bps-sample created
ubuntu@ubuntu:~/BackupResourcesController_k8s$ kubectl get pods -L type
NAME                          READY   STATUS     RESTARTS   AGE   TYPE
bps-sample-3f583358bca8       1/1     Running    0          28s   hotbackup
bps-sample-4cd4ff0807ed       1/1     Running    0          28s   hotbackup
bps-sample-58fe7068249d       0/1     Init:0/1   0          28s   coldbackup
bps-sample-59c4f1208433       0/1     Init:0/1   0          28s   coldbackup
bps-sample-9be87e6d4e00       1/1     Running    0          28s   primary
bps-sample-e6e44b332f58       1/1     Running    0          28s   primary
ubuntu@ubuntu:~/BackupResourcesController_k8s$ kubectl get bps
NAME         NAMESPACE   REPLICAS   HOTBACKUPS   COLDBACKUPS   AGE
bps-sample   default     2/2        2/2          2/2           50s
```

Fig. 4. List of resources created by the BPS instance.

III. Demonstrations

We implement the controller by Operator SDK v1.4.2 [9], Golang 1.16 [10], and Kubernetes v1.20.4 [1] running on an Intel Core i7- 10510U 1.80 GHz 2-core CPU, 4 GB memory. We deploy the designed controller as a *deployment* in Kubernetes with relative resources, e.g., namespace, CRD, and permissions for modifying the BPS instances and Pods. We create a BPS instance with two primary Pods, two HB Pods, and two CB Pods by a YAML file shown in Fig. 3. Figure 4 shows the created Pods in the BPS instance. The time for the controller to handle the BPS creation request is 0.211 [s]. The total creation time for the instance is 5.344 [s].

When the traffic increases, the request of updating the number of Pods for each type in a BPS instance from two to three for load balancing arrives. The controller compares the current and desired states of the different types of Pods; one Pod for each type is added. Figure 5(a) shows the results. The Pod index refers to the name of Pod in Fig. 4. One HB Pod, Pod 3, is converted to a primary Pod. Two CB Pods, Pods 5 and 6 are activated and converted to HB Pods. Three Pods, Pods 7, 8,and 9 are newly created as the CB Pods. The time for the controller to handle the BPS updating request is 0.113 [s]. The total transition time from the last state to the current one of the BPS instance is 3.681 [s].

When the traffic decreases, the request of updating the number of Pods for each type to one for higher utilization arrives. Figure 5(a) shows the results after resource releasing. The controller compares the current and desired states of the different types of Pods and decreases the numbers of Pods for all types to one randomly with releasing the corresponding resources. The time for the controller to handle the BPS updating request is 0.226 [s]. transition time from the last state to the current one of the BPS instance is 2.137 [s].

In the case that a primary Pod failure is detected, the HB and CB Pods are activated to take over the task of the failed primary Pod. We delete a primary Pod, Pod 3, to demonstrate the case of primary Pod failure. Figure 5(b) shows the maintenance results in confronting with a failure. The controller maintains the number of Pods for different types until the current BPS instance state becomes the desired state.

Initial		Increase the number of Pods for each type		Decrease the number of Pods for each type	
Pod	Type	Pod	Type	Pod	Type
1	Primary	1	Primary		
2	Primary	2	Primary		
3	HB	3	Primary	3	Primary
4	HB	4	HB		
5	CB	5	HB		
6	CB	6	HB	6	HB
		7	CB		
		8	CB		
		9	CB	9	CB
Time for controller: 0.211 [s]. Total creation time: 5.344 [s].		Time for controller: 0.113 [s]. Total transition time: 3.681 [s].		Time for controller: 0.226 [s]. Total transition time: 2.137 [s].	

(a) List of resources after updating the BPS instance.

Before failure		After failure	
Pod	Type	Pod	Type
3 X	Primary		
6	HB	6	Primary
9	CB	9	HB
		10	CB
Time for controller: 0.079 [s]. Total transition time: 1.933 [s].			

X: failure

(b) List of resources after failure of primary Pod.

Fig. 5. State transition of Pods triggered by requests.

```
ubuntu@ubuntu:~/BackupResourcesController_k8s$ kubectl delete pod bps-sample-4cd4ff0807ed
pod "bps-sample-4cd4ff0807ed" deleted
ubuntu@ubuntu:~/BackupResourcesController_k8s$ kubectl get pods -L type
NAME                          READY   STATUS     RESTARTS   AGE   TYPE
bps-sample-386d11b75f44       1/1     Running    0          11m   hotbackup
bps-sample-58fe7068249d       1/1     Running    0          13m   primary
bps-sample-92cea736875d       0/1     Init:0/1   0          34s   coldbackup
ubuntu@ubuntu:~/BackupResourcesController_k8s$ kubectl get bps
NAME         NAMESPACE   REPLICAS   HOTBACKUPS   COLDBACKUPS   AGE
bps-sample   default     1/1        1/1          1/1           13m
```

Fig. 6. List of resources after deleting a primary Pod in BPS instance.

The HB Pod, Pod 6, is converted to a primary Pod and is exposed to the service. The CB Pod, Pod 9, is successfully activated and converted to an HB Pod. A CB Pod, Pod 10, is newly created. The controller is normally running as shown in Fig. 6. The time for the controller to handle a primary Pod failure is 0.079 [s]. The total transition time is 1.933 [s].

IV. Conclusion

This paper designed and implemented a custom resource and the corresponding controller in Kubernetes to manage the primary and backup resources of network functions. The controller manages the state of each BPS instance to keep the current state consistent with the desired state. Demonstration validated that the controller automatically manages the resources correctly and rapidly. The time for the controller to handle a BPS operation is within one second. The total transition time from the last state to the current one of the BPS instance is within four seconds.

References

[1] The Kubernetes Authors, "Kubernetes," https://kubernetes.io/, accessed Mar. 18, 2021.

[2] ——, "Controllers," https://kubernetes.io/docs/concepts/architecture/controller/, accessed Mar. 18, 2021.

[3] R. Kang, F. He, T. Sato, and E. Oki, "Virtual network function allocation to maximize continuous available time of service function chains with availability schedule," *IEEE Trans. Netw. Service Manag.*, Jul. 2020, early access.

[4] M. Zhu, F. He, and E. Oki, "Optimal multiple backup resource allocation with workload-dependent failure probability," in *IEEE Globecom*, 2020, pp. 1–6.

[5] ——, "Optimal primary and backup resource allocation with workload-dependent failure probability," in *2020 Int. Conf. on Inf. and Commun. Technol. Convergence (ICTC)*. IEEE, 2020, pp. 1–6.

[6] The Kubernetes Authors, "Custom resources," https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources, accessed Mar. 18, 2021.

[7] R. Kang, M. Zhu, F. He, T. Sato, and E. Oki, "Design of scheduler plugins for reliable function allocation in Kubernetes," in *Conf. Design of Reliable Commun. Netw. (DRCN2021)*. IEEE, 2021.

[8] The Kubernetes Authors, "Init containers," https://kubernetes.io/docs/concepts/workloads/pods/init-containers, accessed Mar. 18, 2021.

[9] Red Hat, Inc., "Operator SDK," https://sdk.operatorframework.io, accessed Mar. 18, 2021.

[10] Google, "The Go programming language," https://golang.org, accessed Mar. 18, 2021.

Authorized licensed use limited to: Gheorghe Asachi Technical University of Ia¿i. Downloaded on December 14,2023 at 07:16:24 UTC from IEEE Xplore. Restrictions apply.