Implementation of Virtual Network Function Allocation with Diversity and Redundancy in Kubernetes

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

***Abstract*—Diversity in network function virtualization is to use a group of thin replicas to provide the network services under the required processing ability. Redundancy is to provide a certain number of replicas against function failures and improve network reliability. Kubernetes is a system to deploy and manage virtual network functions automatically. Existing tools in Kubernetes do not provide a resource type to provide required functions jointly considering VNF diversity and redundancy. This paper designs and implements a custom resource and the corresponding controller in Kubernetes to manage the VNF diversity and redundancy jointly. The controller selects suitable replicas from a pool of replica templates to satisfy the required processing ability with the minimum required number of replicas and converts the backup functions to the primary functions when the primary functions cannot provide the required ability. Demonstration validates that the controller automatically manages the resources correctly, improves the resource utilization, and increases the number of acceptable requests.**

***Index Terms*—Virtual network function allocation, redundancy, network diversity, controller, Kubernetes, demonstration**

I. Introduction

Virtual network functions (VNFs) are usually packaged in VMs or containers. Kubernetes [1] is an open-source system for automating deployment, scaling, and management of containerized applications. A Pod is the smallest deployable unit of computing that we can create and manage in Kubernetes. The deployment with a given number of the replicas of Pods is a realization of a network function in Kubernetes. The controller of a resource in Kubernetes adjusts the current state to the expected state through the control loop [2]. For example, the deployment controller maintains the desired number of Pod replicas. The providable processing ability of a network function can be required instead of the number of active replicas of the function.

VNF diversity uses a group of replicas with different processing abilities and resource requirements to replace a single VNF instance, which can fully utilize server computing resources, especially for edge computing devices. The replicas of a VNF are chosen from a pool of replica templates, which is given by the cloud provider. The service providers allocate replicas with the requirements of resources to provide certain processing abilities. VNF diversity may increase the risk of service unavailability since it allocates replicas to different servers. The unavailability of a server leads to service performance degradation. VNF redundancy which provides backups for replicas is adopted to increase the service reliability. The works in [3] and [4] provided allocation models considering VNF diversity and redundancy jointly, which improves the service resiliency.

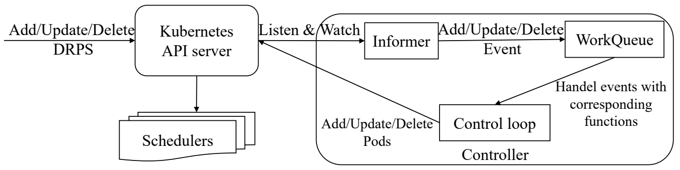

Fig. 1. Overall structure.

However, the current deployment in Kubernetes is based on the allocation of a given number of replicas with fixed specifications. The fixed specifications may not fully utilize the computing resources of servers. If the deployment can be realized by the replicas with different resource requirements and processing abilities, the deployment becomes flexible to be deployed to the servers with different capacities, and the computing resources of servers can be fully utilized. The automatic selection, creation, and management of diversity and redundancy resources are not considered in Kubernetes.

This paper designs and implements a custom resource controller in Kubernetes based on the processing ability. The custom defined resource (CDR) jointly considers the diversity and redundancy of VNFs. We call it *diversity and redundancy Pod set (DRPS)*. We use exact and approximate methods to select suitable replicas from a pool of replica templates to satisfy the required processing ability with the minimum required number of replicas. At last, we perform demonstrations of the controller including the function allocation and the switching from primary replicas to backups, to show the improvement of server utilization by adopting DRPS.

II. Design and Implementation

A. Overall structure

Fig. 1 overviews the structure of the reported controller and relative components. The service providers submit the configurations including adding, updating, and deleting, in a form defined in DRPS definition to the Kubernetes application programming interface (API) server. Informer listens to the changes of the DRPS instances and pushes the corresponding events to the WorkQueue. The unterminated control loop handles the events in WorkQueue with the corresponding functions to let the current state of the DRPS instances keep pace with the desired state of the DRPS instances configured by the users. The key point is how to use the controller to

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

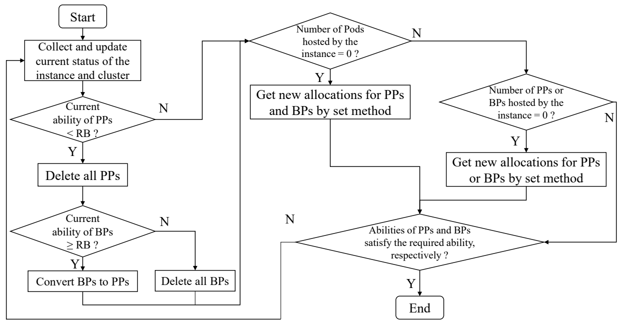
Start
Collect and update current status of the instance and cluster
Current ability of PPs < RB ?
N
Y
Delete all PPs
Current ability of BPs ≥ RB ?
N
Y
Convert BPs to PPs
Delete all BPs
Number of Pods hosted by the instance = 0 ?
N
Y
Get new allocations for PPs and BPs by set method
Number of PPs or BPs hosted by the instance = 0 ?
N
Y
Get new allocations for PPs or BPs by set method
N
Abilities of PPs and BPs satisfy the required ability, respectively ?
Y
End

Fig. 2. Flow chart of controller. PP: primary Pod. BP: backup Pod. RB: required ability.

cooperate with the "add", "delete", "update", and "get" of the DRPS instances in Kubernetes. The scheduler decides the locations of the created Pods with the default scheduler or allocation-model-based scheduler in [5].

B. Definition of DRPS instance

The definition of a DRPS instance includes three parts: *metadata, specification, and status*. *Metadata* contains the information that distinguishes different DRPS instances, e.g., name and creation timestamp. *Specification* contains the properties of a DRPS instance, e.g., required processing ability, specified solution methods, and the pool of replica templates. The pool of replica templates contains the templates of Pods, which includes the processing ability of the replica, the resource requirements, and the container image. *Status* contains the latest status of the CDR instance, e.g., the number and the names of Pods hosted by the instance. *Status* is updated periodically or updated after modification in the control loop.

C. Control loop

The control loop receives and handles the create, update, and delete events. When a DRPS instance is created, the Pods are created and allocated to nodes. The selection of replica templates from pools and the scheduling of the created Pods can be determined by two methods: an exact method, e.g, integer linear programming (ILP), and an approximate method, e.g., heuristic algorithms. When some primary Pods hosted by the instance fail, backup Pods are converted to primary Pods. The remaining primary Pods are deleted and a group of new backup Pods is created and allocated. Alternatively, the remaining primary Pods are converted to backup Pods and new backup Pods are created to compensate for the loss of processing ability. Fig. 2 shows the flow chart of the controller.

III. Demonstrations

We implement the controller by Operator SDK v1.4.2, Golang 1.15, and Python 3.7 in Kubernetes 1.20 on a five-node Kubernetes cluster (one master node and four worker nodes). The memory and central processing unit (CPU) of four nodes are up to 2.3 GB and two cores, 2.3 GB and two cores, 4.2 GB and three cores, and 4.2 GB and four cores, respectively. We deploy the controller as a deployment on the master node. We create a DRPS instance by applying a configuration file shown in Fig. 3 and the list of Pods is shown in Fig. 4. A primary Pod hosted by the instance fails, the backup Pods is switched to primary Pods and new backup Pods are generated, as shown in Fig. 5. We can observe that the backup Pod is converted to the primary Pod and a new backup Pod is started.

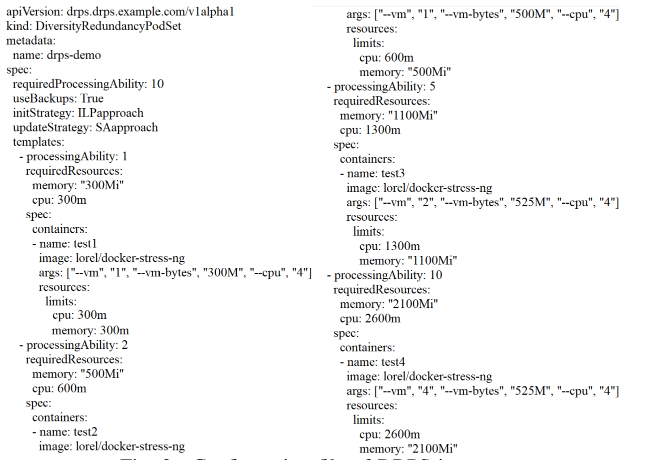
```
apiVersion: drps.drps.example.com/v1alpha1
kind: DiversityRedundancyPodSet
metadata:
  name: drps-demo
spec:
  requiredProcessingAbility: 10
  useBackups: True
  initStrategy: ILPapproach
  updateStrategy: SAapproach
  templates:
    - processingAbility: 1
      requiredResources:
        memory: "300Mi"
        cpu: 300m
      spec:
        containers:
        - name: test1
          image: lorel/docker-stress-ng
          args: ["--vm", "1", "--vm-bytes", "300M", "--cpu", "4"]
          resources:
            limits:
              cpu: 300m
              memory: 300m
    - processingAbility: 2
      requiredResources:
        memory: "500Mi"
        cpu: 600m
      spec:
        containers:
        - name: test2
          image: lorel/docker-stress-ng
          args: ["--vm", "1", "--vm-bytes", "500M", "--cpu", "4"]
          resources:
            limits:
              cpu: 600m
              memory: "500Mi"
    - processingAbility: 5
      requiredResources:
        memory: "1100Mi"
        cpu: 1300m
      spec:
        containers:
        - name: test3
          image: lorel/docker-stress-ng
          args: ["--vm", "2", "--vm-bytes", "525M", "--cpu", "4"]
          resources:
            limits:
              cpu: 1300m
              memory: "1100Mi"
    - processingAbility: 10
      requiredResources:
        memory: "2100Mi"
        cpu: 2600m
      spec:
        containers:
        - name: test4
          image: lorel/docker-stress-ng
          args: ["--vm", "4", "--vm-bytes", "525M", "--cpu", "4"]
          resources:
            limits:
              cpu: 2600m
              memory: "2100Mi"
```

Fig. 3. Configuration file of DRPS instance.

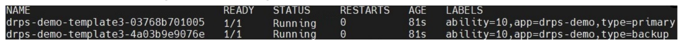
```
NAME                                READY   STATUS    RESTARTS   AGE   LABELS
drps-demo-template3-03768b701005    1/1     Running   0          81s   ability=10,app=drps-demo,type=primary
drps-demo-template3-4a03b9e9076e    1/1     Running   0          81s   ability=10,app=drps-demo,type=backup
```

Fig. 4. Pod list after allocation.

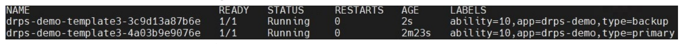
```
NAME                               READY   STATUS    RESTARTS   AGE     LABELS
drps-demo-template3-3c9d13a87b6e   1/1     Running   0          2s      ability=10,app=drps-demo,type=backup
drps-demo-template3-4a03b9e9076e   1/1     Running   0          2m23s   ability=10,app=drps-demo,type=primary
```

Fig. 5. Pod list after primary Pod fails.

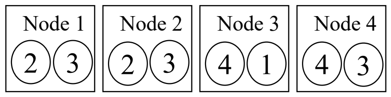

Fig. 6. Allocation results of DRPS instance. A circular is a Pod. The number in a circle means the template which the Pod uses.

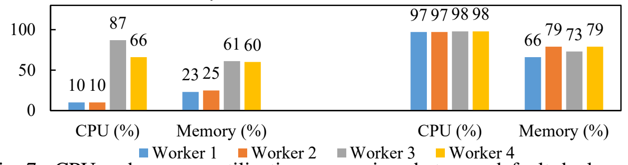

Fig. 7. CPU and memory utilization comparison between default deployment method (left) and DRPS (right).

We deploy the instances with the same required ability and the pool in Fig. 3 by using two types: traditional deployment and DRPS. The traditional deployment accepts one request of the instance and completes the allocations of a primary Pod and a backup Pod in 0.004 [s]. The DRPS instance accepts two requests and completes the allocations in 2.931 [s]. The allocation of the DRPS instance is shown in Fig. 6. The CPU and memory utilizations are compared in Fig.7. We can observe that DRPS can accept more requests and improve the system resource utilization compared with the default deployment method with the cost of longer deployment time.

References

[1] The Kubernetes Authors, "Kubernetes," https://kubernetes.io/, accessed Mar. 8, 2021.

[2] ——, "kube-controller-manager," https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/, accessed Mar. 8, 2021.

[3] A. Alleg, T. Ahmed, M. Mosbah, and R. Boutaba, "Joint diversity and redundancy for resilient service chain provisioning," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 7, pp. 1490–1504, 2020.

[4] R. Kang, F. He, and E. Oki, "Resilient resource allocation model in service function chains with diversity and redundancy," in *2021 IEEE Int. Conf. on Commun. (ICC)*. IEEE, Jun. 2021, pp. 1–6.

[5] 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.* IEEE, Apr. 2021, pp. 1–3.