Sangfor PaaS Platform – KubeManager White Paper

Sangfor Technologies Inc.

September 27, 2020

Contents

1 PaaS Overview

With the rapid evolution of mobile Internet, big data, cloud computing, and other information technologies, China actively promotes the integrated development of high-tech and traditional industries. As cloud computing and virtualization technologies grow increasingly important for IT management and innovation in various sectors, it has become an irresistible trend for the digital transformation of enterprises to build a PaaS cloud platform with more flexible resource allocation, more centralized and intelligent data utilization, more unified and efficient service integration, faster and more convenient application development. So it can realize the unified management of IT resources (data, services and applications). The PaaS platform is downward compatible with IaaS resources (VM resources, physical resources, load balancing and networking). Besides, it is upward support application management, container management, continuous integration, distributed storage, and cluster management. It provides such services as one-key deployment, elastic scaling, self-healing, consolidates basic resource management, application & environment management, so as to improve the efficiency of IT resources and reduce the O&M costs.

1.1 Development of Docker

Traditional VMs, such as OpenStack and VMWare, need to simulate the whole machine including its hardware. And each VM runs its own OS. Once a VM is enabled, all the resources pre-allocated to the VM will be occupied. Each VM requires applications, necessary binaries and databases, and a complete user OS.

The container technology shares hardware resources and OS with the host, to realize the dynamic allocation of resources. The container integrates applications and all the dependent packages thereof but shares the kernel with other containers. It runs as a separate process in the user space of the host OS.

The container technology shows us a path to realize OS virtualization. It can run applications and the dependent environment in the process where resources are isolated. By using containers, we can easily pack the code, configuration, and dependent environment of applications. And we can turn them into building blocks easy to use, thus achieving a variety of goals, such as environment consistency, operational efficiency, developer's productivity, and version control. Containers can help us deploy applications quickly, reliably, and consistently, regardless of the deployment environment. They also enable more refined control over resources and improve the service efficiency of infrastructures. The following figure intuitively illustrates the difference between the VM and container.

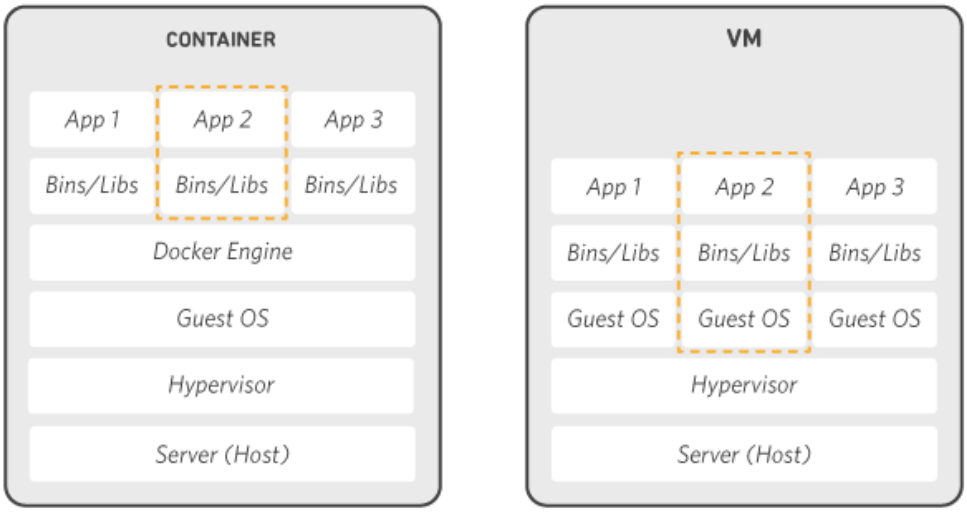

Figure 1: Comparison of Container and VM

Docker, as a package of Linux containers, provides an easy-to-use container interface. It is the most popular Linux container solution currently.

Linux container is another virtualization technology derived from Linux. Simply speaking, Linux container does not simulate a complete OS but isolates the process, just like putting a protective layer outside the normal process. For the process in a container, all resources it contacts are virtual. In this way, it realizes isolation from the underlying system.

Docker packs applications and their dependencies in a file. After this file is run, a virtual container will be created. Programs run in this virtual container as if they were running on a real physical server. With Docker, you don't have to worry about the environmental problem.

To sum up, Docker's interface is quite simple. Users can easily create and use containers and put their applications into containers. These containers are also subject to version management, replication, sharing, and modification.

Docker outperforms traditional virtualization methods in the following aspects:

- Docker starts quickly in seconds, while a VM needs several minutes to start.
- Docker requires fewer resources and carries out virtualization at the OS level. And Docker container interacts with the kernel, causing almost no performance loss and better than virtualization through the Hypervisor layer and kernel layer.
- Docker is lighter, as its architecture can share a kernel and application databases, thus occupying a very small memory. In the same sized hardware environment, Docker runs far more images than do VMs and has an ultra-high system utilization rate.
- Manageability: Via K8s or other orchestration tools, Docker can realize automatic management over large-scale and high-density container clusters.
- High availability and recoverability: The platform realizes high availability in multi-cluster and data replicas mode and provides the self-healing function.
- Rapid creation and destruction: Docker creates virtualization in seconds while VMs do so in minutes. The rapid iteration of Docker can save lots of time for development, testing, and deployment.
- Delivery and deployment: VMs can realize consistency in environment delivery through mirroring, but mirror distribution cannot be systematized. Docker records the container building process in Dockerfile and can realize rapid distribution and deployment in the cluster.

1.2 Kubernetes

Kubernetes is a brand new cutting-edge distributed architecture based on container technology. Kubernetes (K8s) is an open-source container cluster management system launched by Google (inside Google: Borg). Based on Docker technology, Kubernetes provides a series of complete functions (deployment, resource scheduling, service discovery, dynamic scaling, etc.) for containerized applications, to facilitate large-scale container cluster management.

Kubernetes is a complete distributed system supporting platform with the cluster management capability, multi-expansion and multi-level security protection and access control mechanism, multi-user application support capability, transparent service registration and discovery mechanism, built-in intelligent load balancer, powerful fault discovery and self-healing capability, service rolling upgrade and online expansion capability, scalable automatic resource scheduling mechanism, and multi-granularity resource quota management capability. Moreover, Kubernetes provides perfect management tools, covering all aspects of development, deployment testing, and O&M monitoring.

Concerning cluster management, Kubernetes divides the devices in the cluster into a Master node and many working nodes. The Master node runs a group of processes related to cluster management, including kube-apiserver, kube-controller-manager, and kube-scheduler, which realize the management capabilities of the entire cluster covering resource management, Pod scheduling, elastic scaling, security control, system monitoring, and error correction for the automatic run. Nodes, as the working nodes in the cluster, run real applications. On a Node, the minimum operating unit governed by Kubernetes is Pod. Kubernetes has its service processes Kubelet and kube-proxy run on Nodes. These service processes are responsible for Pod creation, startup, monitoring, restart, destruction, and realizing the load balancer in software mode.

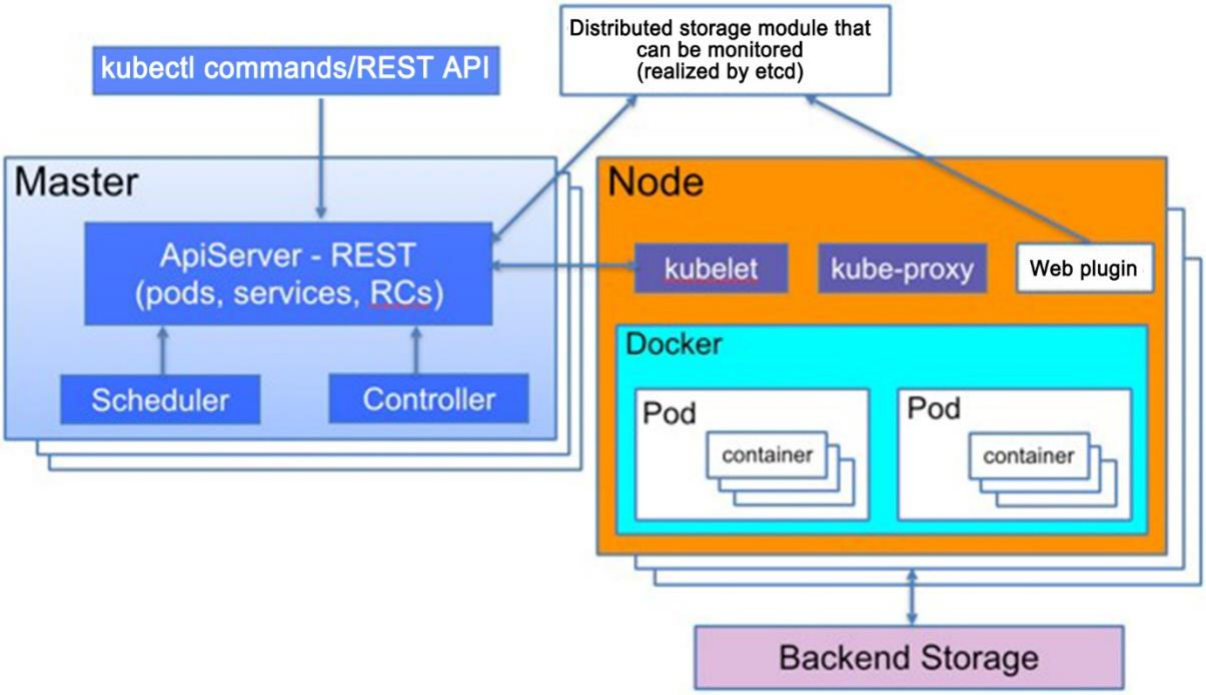

Figure 2: K8s Architecture

Kubernetes advantages:

- Container orchestration
- Lightweight
- Open source
- Elastic scaling
- Load balancing

1.3 Sangfor PaaS Platform – KubeManager

KubeManager is a visual console that manages multiple clusters across different service domains, multiple IaaS cloud platforms, and multiple data centers. It can simplify unified management and control in multi-cluster scenarios.

2 Overall Structure

2.1 KubeManager Architecture

2.1.1 Overall Architecture

The overall architecture of KubeManager system is shown below:

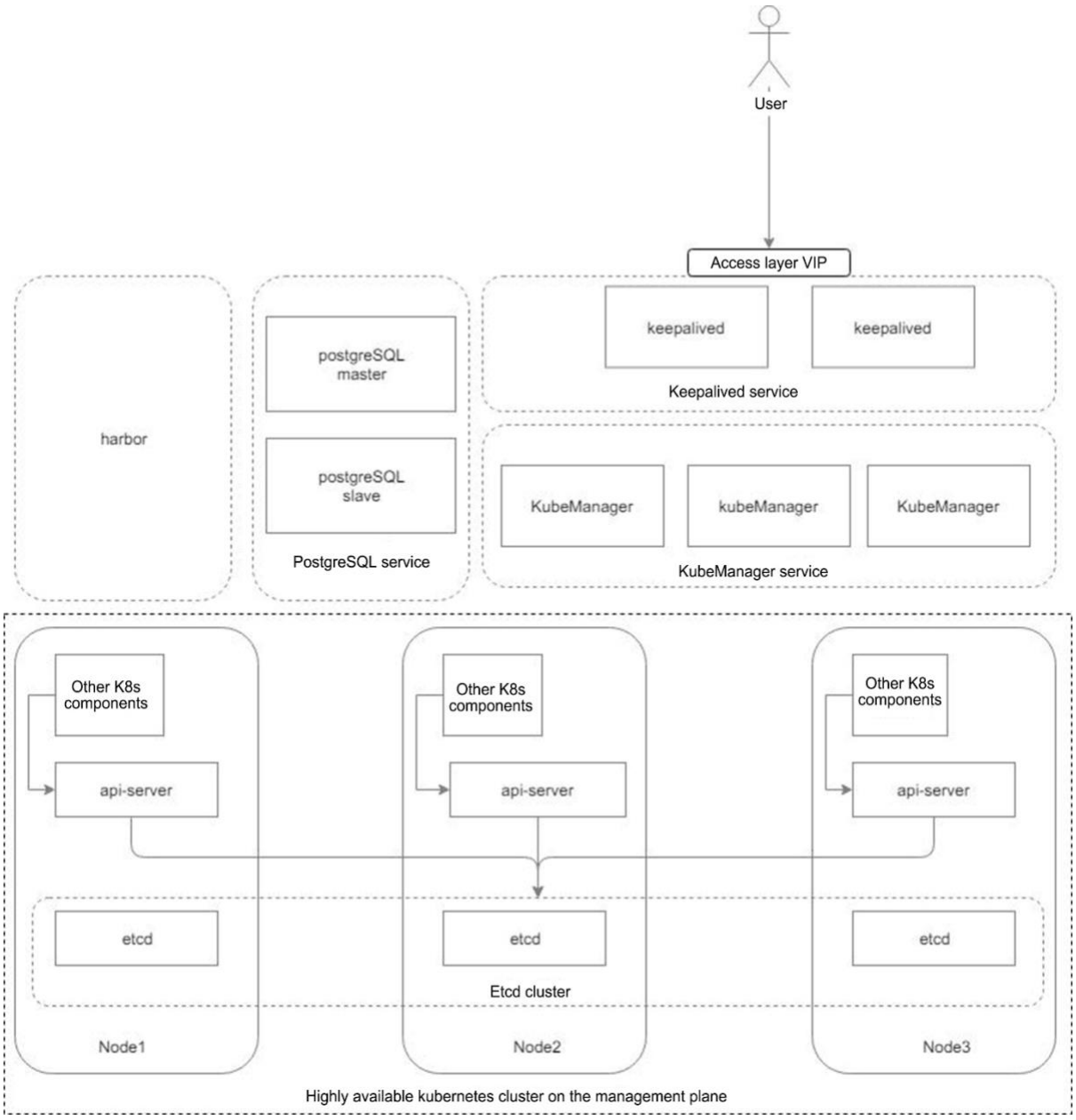

Figure 3: Overall Architecture

From the above figure, we can see that the underlying layer of KubeManager is a Kubernetes system. KubeManager runs on Kubernetes as an application. The entire system also provides database, Harbor and other services. We provide a highly available mechanism for each component of the entire KubeManager system. The underlying layer also reuses the high availability of Kubernetes. This ensures high availability at the system level. Failure of a node or a copy in any service will not interrupt the system for providing external services.

The following subsystems ensure the high availability of the entire system:

1. The keepalived service ensures the high availability of interfaces through floating IP addresses.
2. The KubeManagerdaemonset service ensures the access performance expansion and high availability of services.
3. The postgreSQL service component ensures the high availability of databases. The operator component allows automatic configuration of various database modes (single host, master-slave, and one-master-to-multi-slave) and can be used to store audit logs or

user information of Harbor .

4. The etcd cluster ensures the high availability of configuration data.
5. The deployment of Kubernetes service determines the high availability of the management cluster. The deployment mode is shown in the figure. For a three-node etcd cluster, the api-server on each node is connected to each etcd node. And other components of Kubernetes are only connected to the api-server of their respective nodes. When the api-server in a single node is unavailable, Kubernetes sets the node to "not ready". The controller will sense this information and disable the Pod on this node. In this case, the Kubernetes system will kick the Pod out of the service.
6. Other components of Kubernetes achieve high availability in one-master-to-multi-slave mode.

2.1.2 KubeManager Architecture

KubeManager is the core of the entire system. And its design is critical. The following figure shows the overall service architecture of KubeManager:

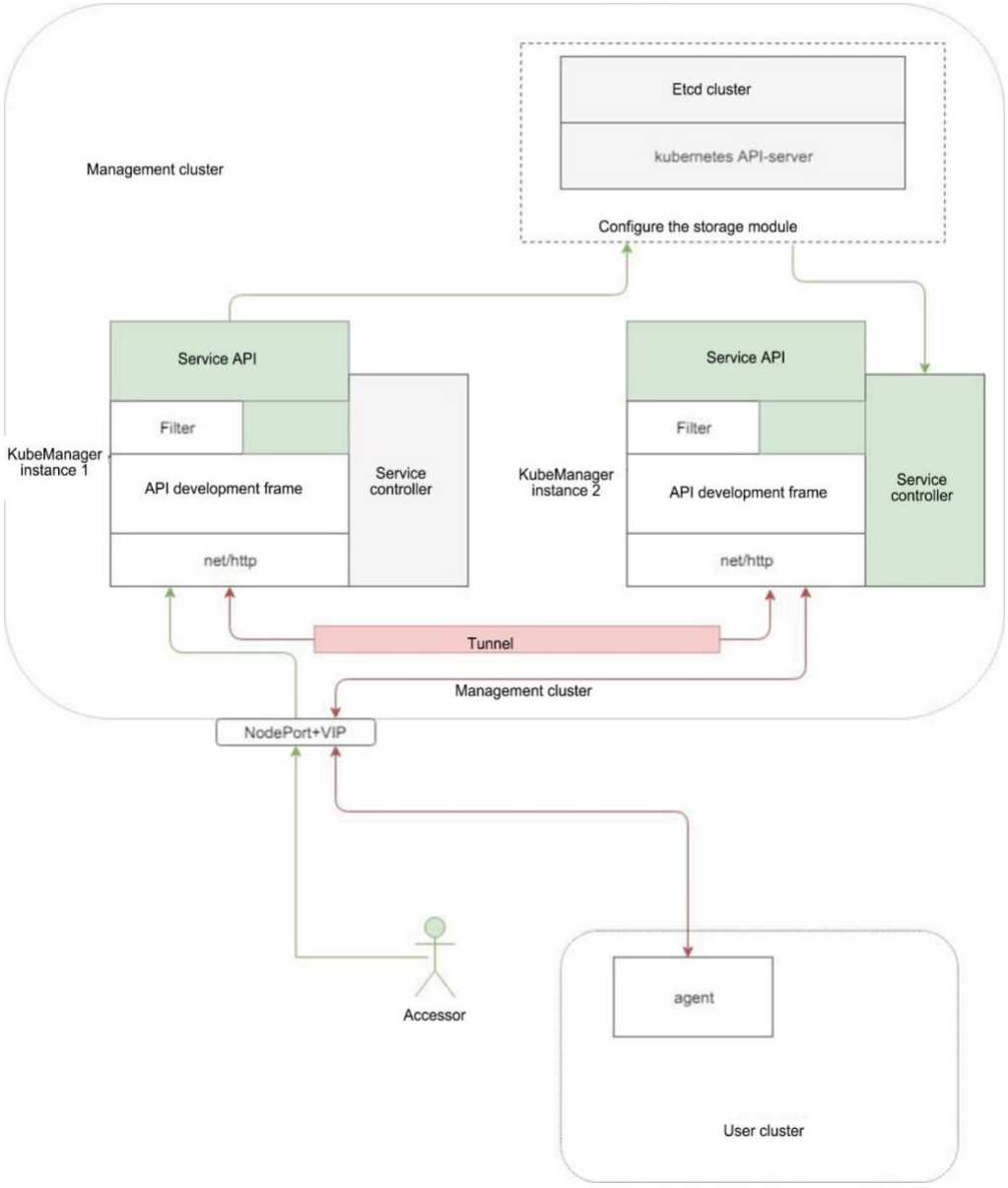

Figure 4: KubeManager

The KubeManager system is essentially an API server running in the Kubernetes cluster. It uses many features of the Kubernetes system. For example, its configuration is realized by the CRD resources of Kubernetes. The specific services afforded by this configuration are realized based on Kubernetes controller. And load balancing is realized through the service mechanism of Kubernetes. Furthermore, it uses the distributed resource lock provided by Kubernetes to prevent competition between controllers of multiple instance services. The development of a service logic mainly includes the service API development and the service controller implementation. To improve the development efficiency, we provide an automatic code generation tool similar to operator-sdk to generate the interactive logic between the controller and kube-api.

The access performance of API can be improved by adding instances to realize access at high concurrency. And the final access performance of API only depends on the read-write performance of Kubernetes' etcd cluster. However, the controller for specific service implementation is provided with a lock to prevent competition between equivalent controllers in multiple instances. That is, controllers of all services can only run in one instance. And those in other instances can only serve as backup controllers. Of course, some controllers not involved in the competition will run in every instance.

The data interaction between the hypervisor and user cluster is achieved totally through the tunnel established between agents running in the user cluster and the hypervisor. Because the multi-instance scenario has a problem that users cannot link to the user cluster since user access and tunnel connection are carried by different instances. The hypervisor will use Kubernetes' endpoint mechanism to register its own IP address. Other instances, after monitoring this endpoint and obtaining the address of a new instance, will establish a tunnel connection with the new instance and record the instance information of the owner of the tunnel connected with the user cluster. When a user accesses the user cluster through the hypervisor, the data will be forwarded to the instance with a tunnel connected with the user cluster. Then, the request will be sent to the user cluster through this tunnel. The reply data of the user cluster follows a similar procedure.

2.1.3 User Cluster Deployment Architecture

The Kubernetes cluster deployed by KubeManager to run user services also fully realizes the high availability of the system without wasting user resources. Its architecture is shown below:

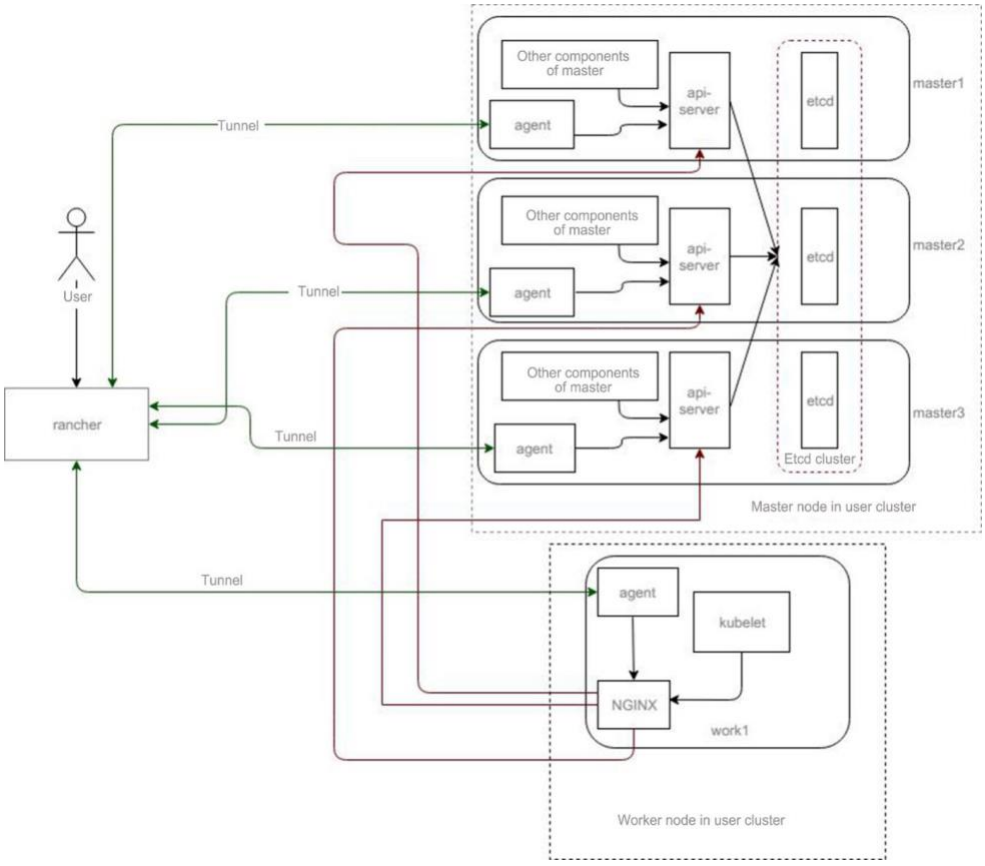

Figure 5: User Cluster

1. The high-availability rule of the master node in the user cluster works the same way as the management plane and is not discussed here.
2. When the worker node (kubelet) in the user cluster and Pod on the node access api-server, they achieve high-availability access and load sharing for api-server through reverse proxy of locally deployed Nginx.
3. The connection selection module inside KubeManager ensures the high availability of user cluster tunnel. As long as KubeManager is connected with any agent, it can manage the user cluster normally.

2.2 Environment Deployment

2.2.1 Environment Requirements

Cluster Type	Role	CPU Cores	Memory (GB)	System Disk (GB)	Data Disk (GB)
Management cluster covering registry	master1	4	8	80	250
	master2	4	8	80	250
	master3	4	8	80	250
User cluster	master1	4	8	100	150
	master2	4	8	100	150
	master3	4	8	100	150
	worker1	8	16	100	150
	worker2	8	16	100	150
	worker3	8	16	100	150

Figure 6: Environment Configuration Requirements

2.2.2 Deployment of User Cluster

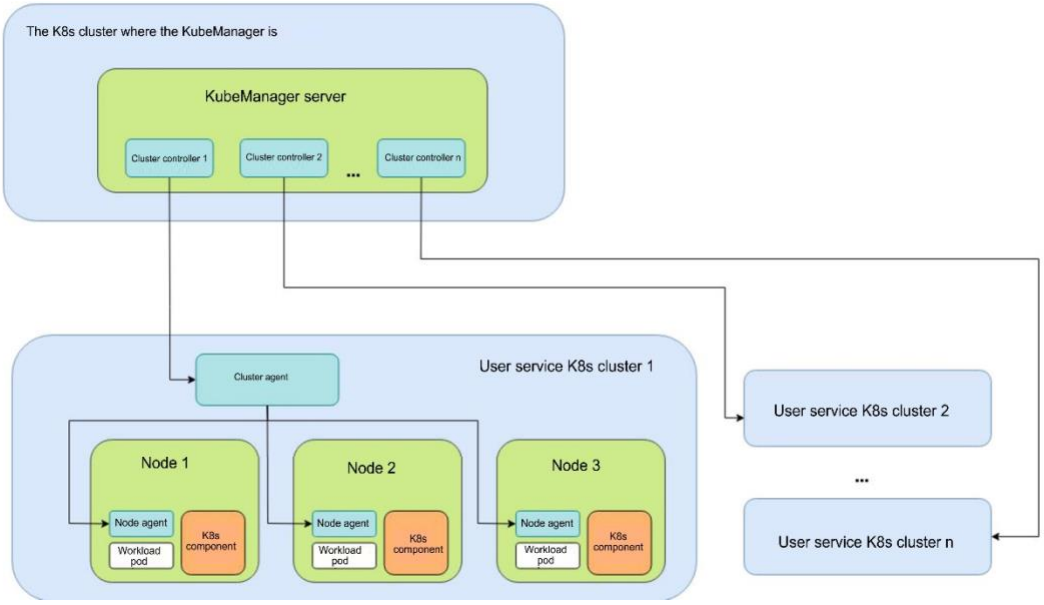

Figure 7: User Cluster

2.2.3.1 Architecture of User Cluster

2.2.3.2 Installation Procedure

1. Firstly, KubeManager starts the agent container on the node through ssh connection. Then, the agent container and KubeManager server establish a tunnel in between.
2. The KubeManager server generates relevant Docker tasks according to the node role and cluster configuration.
3. Whenever a new etcd or control node joins the cluster, KubeManager calls the ske engine to generate a relevant license and informs the node to execute the Docker task through the tunnel, so as to obtain the license and start related K8s containers for rapid deployment of the K8s cluster.
4. After etcd and kube-apiserver in the cluster start running, the worker node starts to join the cluster. By requesting the KubeManager server to obtain the Docker task to run K8s related components, the node is registered in kube-apiserver to join the K8s cluster.

2.2.3.3 Cluster Controller

After the K8s user cluster is ready, the KubeManager server starts the controller of the K8s user cluster. It monitors the K8s resource changes of the user cluster, and transmits the user's instructions of resource creation, editing, and deletion to the cluster controller.

The cluster controller then transmits such instructions to the cluster agent of the user cluster and finally calls the K8s cluster API through the cluster agent to complete these instructions.

2.2.3.4 KubeManager-Agent

KubeManager deploys the node Agent DaemonSet and cluster Agent Deployment in each user cluster to keep the connection between the cluster agent and skm.

The cluster agent provides the following functions:

- Connect and use the Kubernetes API in the Kubernetes cluster.
- Manage workloads, pod creation and deployment within the cluster .
- Transmit messages (events, indicators, health status, node information, etc.) between the cluster and SKM Server

Next is about node agent.

The KubeManager server communicates through the cluster agent by default. If the cluster agent becomes unavailable, a node agent in the cluster will create a communication channel, which will be connected to the cluster controller, to enable the communication between the cluster and users.

Through the kubectl command line of the cluster on the page, users can interact with the K8s cluster in shell. This function is realized by accessing the node agent's container. Moreover, other cluster operations, such as creating the etcd node, backing up and restoring the etcd node, and license rotation, are also realized through the node agent.

3 Basic Functions of KubeManager

3.1 User System

3.1.1 Overview of User System

KubeManager includes local accounts and third-party accounts. The user system of KubeManager depends on the strong authentication and authorization of Kubernetes. To learn more about these, please refer to “authorization and authentication”. The user system of KubeManager is the unified authentication center of all user clusters. A user, after passing the authentication, can play its role through Impersonate-User and conduct the permission check by the authorization mechanism of Kubernetes.

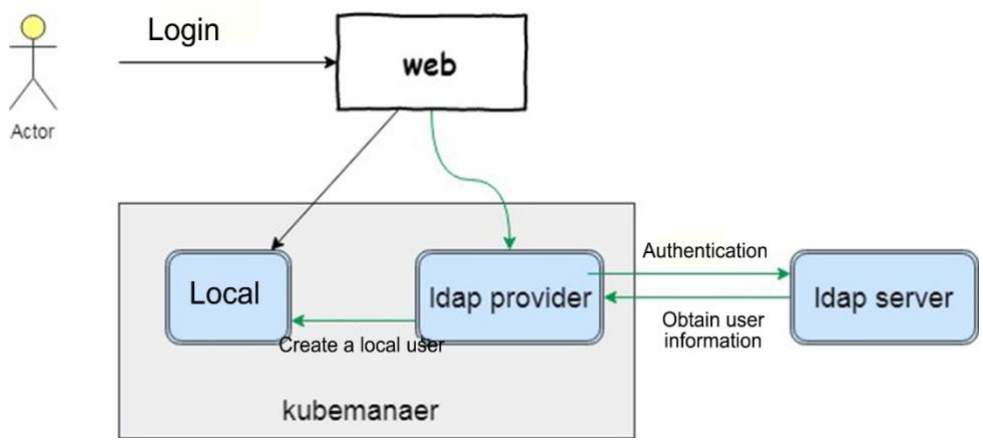

Figure 8: Account System

3.1.1.1 Local user

The administrators can add, edit, delete and disable users. Besides, it can authorize resources and projects to users.

3.1.1.2 Docking of third-party account

The administrator can configure OpenLDAP and FreeIPA servers. After the service account is authenticated, the administrator can choose to import all ldap users or specified ldap users. After users are imported, resources can be authorized to them. Selecting “Allow all valid users”, users from any ldap can log in to KubeManager, including those created on ldap after being imported.

3.1.2 Role Configuration

Kubernetes supports RBAC and allows users to access any resources. KubeManager defines new resource types (roletemplate and roletemplatebinding) for resource binding. KubeManager-defined controller will automatically convert roletemplatebinding into Kubernetes-defined rolebinding and clusterrolebinding. And it will issue appropriate permissions to each user cluster.

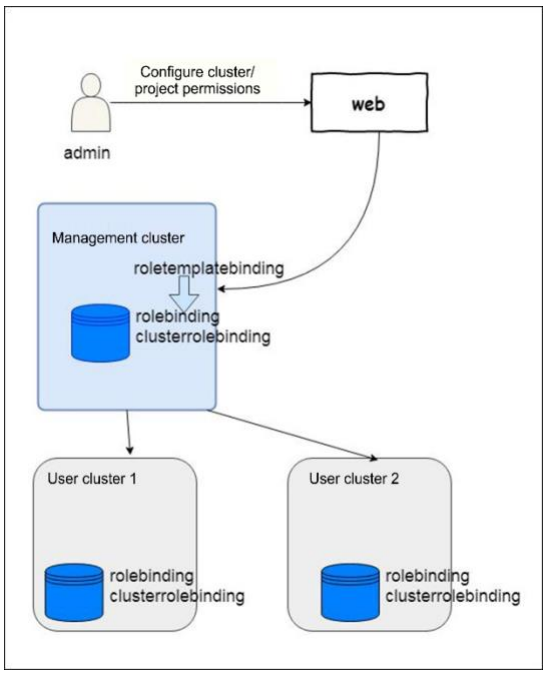

Figure 9: Role Configuration

KubeManager provides permissions at three levels: global, cluster, and project. The global level includes the non-K8s cluster permissions and KubeManager custom permissions, such as user and third-party authentication. The administrator can operate all users and clusters. The cluster level permissions allow you to configure permissions for clusters. For example, the cluster administrator can edit the cluster, while members can only view the cluster; the cluster administrator can operate all items, while members can only create items and cannot operate existing items. The project level permissions allow you to configure the native resources of K8s, such as namespaces, deployment, and Pod. The project administrator can add project members and edit the project. The project members can operate namespaces, applications, etc.

3.2 Registry

3.2.1 Overview of Registry

The registry is mainly used to store Docker images, which are used to deploy container services. Except external image storage and management, the registry also provides many other functions (security authentication, user interface, image scanning, inter-registry image synchronization, image cleaning, helm integration, component HA deployment, event audit, image security, multi-tenancy management, RBAC integration, open API, and multi-user source authentication). It even supports the storage and management of cloud native production data compliant with the OCI specifications.

3.2.2 Image Type

Currently, the registry supports Docker Hub images and user's private images. By default, KubeManager deploys a Harbor enterprise registry for users to deploy Kubernetes workloads and store finished images.

3.2.3 Image Lifecycle

Image lifecycle includes the generation, upload, and deletion of image versions. Image lifecycle usually interfaces with external systems. For example, DevOps provides a flexible and agile registry.

3.2.4 Helm Chart Management

The registry integrates the management and storage of Helm Charts and provides an app store similar to the software repository provided by Kubernetes.

3.3 Overview of Harbor Registry Architecture

3.3.1 Overall Harbor Architecture

KubeManager provides the Harbor enterprise registry for user requirements on image usage. Harbor is an open-source artifact registry. It can protect artifacts (container images, Helm Chart, etc.) through policies and role-based access control. It can also scan images, eliminate harms from security vulnerabilities, and sign images as trusted contents. As a CNCF graduation project, Harbor provides compliance, performance, and interoperability to help users manage artifacts continuously and safely in cloud native platforms, such as Kubernetes and Docker.

This section introduces the design and implementation of Harbor architecture. The product architecture is shown below:

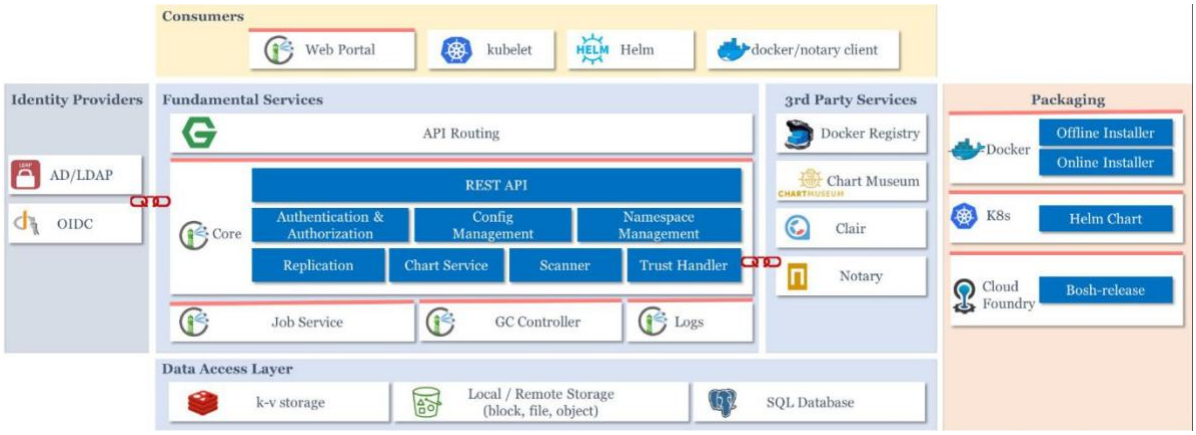

Figure 10: Harbor Architecture

3.3.2 Description

Harbor includes a series of components categorized into a 3-layer architecture as shown in the above figure.

3.3.3 Data Access Layer

This layer provides data storage and serves as a stateful data persistence layer.

1. k-v storage

   Redis can provide data caching and storage of task data

2. Data storage

   Provide storage support for numerous backend devices and realize data persistence for the registry and chart museum

3. Database

   PostgreSQL database can store Harbor data model, including projects, users, roles, replication policies, tag retention policies, scanners, charts, and images

3.3.4 Basic Service Layer

This layer realizes the processing workflow and major functions of Harbor. It serves as a stateless component layer.

1. Proxy provides a reverse proxy server and API routing by Nginx. It can access to core components of Harbor, such as core, registry, web portal, and token services. It delegates

browser requests and Docker client requests to functional components behind the reverse server.

2. Core function, to provide the following Harbor services:

1). Authentication and authorization

- Provide the authentication service to cover local accounts, AD/LDAP, or OIDC.
- RBAC authorization mechanism; some image operations (push and pull) need to be authorized.
- Token service provides token for Docker push or pull, and binds a project based on token and RBAC.

2). Configuration management

Cover the configuration of system management, such as setting the authentication type, email, and license.

3). Project management

Manage the data and metadata of a project. And provide isolated management data.

4). Quota management

Quota check, project quota management, etc.

5). Chart service

Delegate chart requests to backend chart museum middleware for scaling and improving the management capacity of chart.

6). Tag management

Manage and monitor tags to realize some extra functions.

7). Content trust

Support content trust through the backend Notar. Currently, it only supports content trust for the container image,

8). Replication copy

Manage the replication policy and registry adaptation, and monitor and trigger concurrent replications. Realize many types of registry adaptation:

- Distribution (Docker registry).
- Docker Hu.
- Huawei SWR.
- Amazon ECR.
- Google GCR.
- Azure ACR.
- Ali ACR.
- Helm Hub.

9). Scanning management

Provide adaptation for multiple provider scanning services at the backend. And it can provide the scanning service overview and report. Currently, the following security scanning

providers can be configured:

- Trivy provided by Aqua Security.
- Anchore Engine provided by Anchore.
- Clair provided by CentOS.

Currently, it only supports image scanning.

3. Webhook

   Provide the webhook event to flexibly interface with the workflow of external system.

4. Job Service

   Provide the task queue service and support asynchronous concurrency; provide other services and components with the task queue service through a simple RESTful API.

5. Logs

   Collect logs of other modules and save them to one place.

6. GC Controller

   Manage the GC policy settings, and start and track the removal of GC garbage.

7. Chart Museum

   Third-party chart repository service, providing chart management and call.

8. Docker Registry

   Third-party registry server, responding to the push and pull operations of the image. Harbor requires access control for image operation. For the push or pull operation, each client needs to carry a valid token.

9. Notary

   Third-party content trust server, responding to safe release and check contents.

3.3.5 Access Layer

The access layer provides external call, web interface, Kubernetes workload deployment, Helm application packaging, Docker container service, etc. To pull an image from the private registry, Docker and Kubernetes need a voucher. Docker enters the user package and password through the Docker client, to generate the voucher. The Kubernetes cluster uses the Secret of Docker-registry type to apply the voucher.

1. Docker login procedure

Run “docker login 192.168.1.10” to send a request to the Harbor registry. What happens to the Docker login procedure?

When you type “docker login” and press “Enter” on your device. The Docker client initiates the http GET request to the address "192.168.1.10/v2/".

1). Firstly, the request arrives at port 80 monitored by the Nginx proxy container for reverse service. Then, Nginx forwards the request to the backend service of the Registry container.

2). The Registry container is configured to provide token-based authentication. Therefore, registry returns a 401 error code to inform the Docker client that the request contains an invalid token and responds to a special URL. In Harbor, this URL points to the token service among core services.

3). The Docker client, upon receipt of the state code of error, sends the request header

embedded with the username and password to the token service URL, based on the basic authentication specifications of HTTP.

4). When the request embedded with the username and password arrives at port 80 monitored by the Nginx proxy container, Nginx forwards the request to the UI container according to rules. The token service in the UI container, after receiving the forwarded request, decodes the username and password from the header.

5). Upon receipt of the username and password, the token service checks the database and authenticates the users stored in the PostgreSQL database. The token service, when configured for AD/LDAP authentication, will send the username and password to an external AD/LDAP authentication server. After successful authentication, the token service returns a HTTP state code of success. The HTTP response contains the token generated based on the private key.

6). The Docker login procedure is done. The Docker client receives the state code of success and saves the token to "~/.docker/config.json".

2. Docker push procedure

After Docker login succeeds, Docker can upload images to the registry with the push command.

Run "docker push 192.168.1.10/library/hello-world" to send a request to the Harbor registry. What happens to the Docker push procedure?

1). Firstly, similar to the login procedure, Docker sends a request to the registry container and returns the URL provided by the token service.

2). When continuing to request from the token service, the Docker client builds a request with additional information, indicating that it is a push operation against the image.

3). The token service, after receiving the request forwarded by Nginx, will query the database for user rules and permissions concerning the push operation. If the user has the push operation permission, the token service decodes the information of the push operation, signs the generated token with the private key, and returns it to the Docker client.

4). The Docker client, upon receipt of the token, sends the request header containing the token to the registry. Once receiving the request, the registry decodes the token with the public key and checks the contents. The public key of the token service maps to the private key. For the push operation, if the registry finds a valid token, the image upload process begins.

3.4 High Availability and Deployment of Registry

3.4.1 Registry Deployment Architecture

This section describes the deployment of the registry architecture as shown in the following figure:

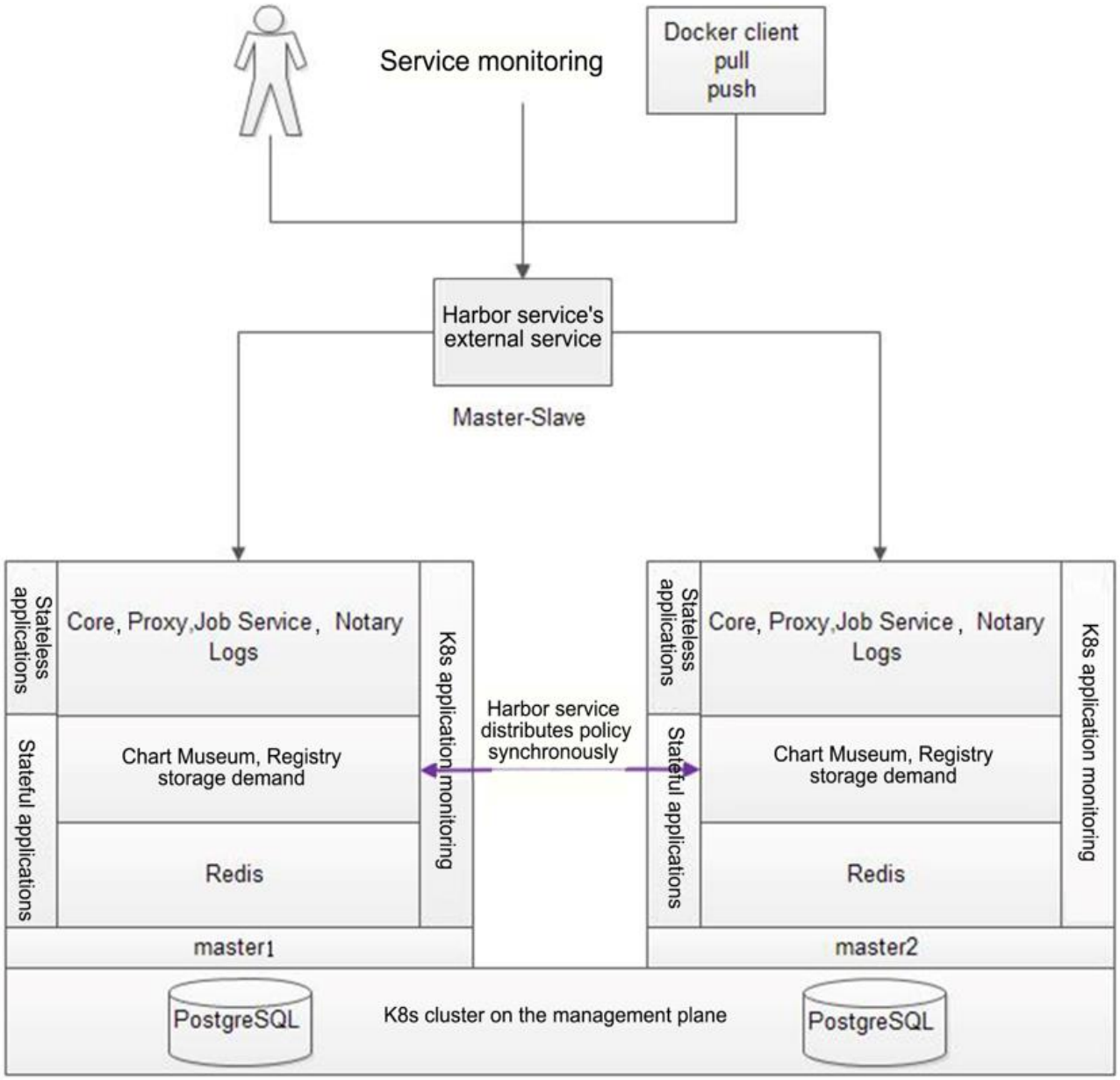

Figure 11: Registry Deployment Architecture

3.4.2 Registry Deployment

The KubeManager registry is deployed in a Kubernetes cluster and supports the single-node or dual-node high-availability architecture. Since the registry uses local storage without data protection and high availability, the single node suffers a single point of failure. Therefore, dual nodes are deployed to provide high availability for data in master-slave mode, in which the data are synchronized from the master node to the slave node using the synchronization policy of the application layer.

3.5 Workload

Workloads are the objects for setting the Pod deployment rules. According to these rules, Kubernetes implements the deployment and update corresponding fields of the workload using the current state of applications. Workloads enable you to define the scheduling, scaling, and upgrade rules of applications.

3.5.1 Pods

3.5.1.1 Overview

A Pod is the basic execution unit of Kubernetes applications. A Pod encapsulates an application container (or multiple containers in some cases), storage resources, unique network IP address, and options controlling the running of container(s). A Pod can include one or multiple containers working collaboratively. It is recommended to divide the containers into several Pods to facilitate infrastructure usage and application scaling.

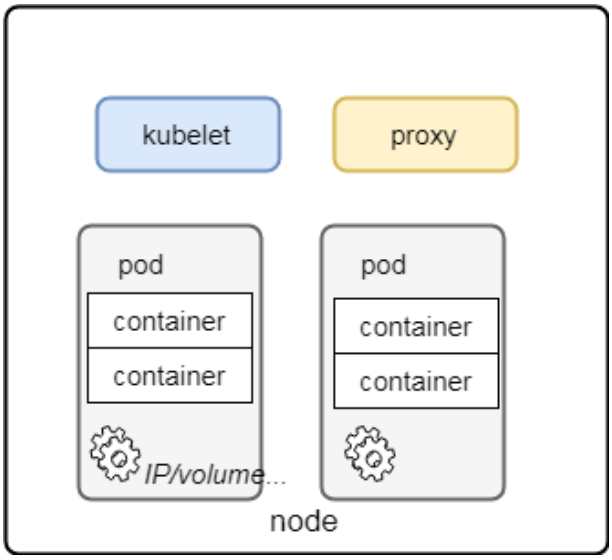

Figure 12: Pods

Some Pods have both initial containers and application containers. Initial containers run and complete before the startup of application containers.

3.5.1.2 Templates

```
apiVersion: v1
kind: Pod
metadata:
    name: myapp-Pod
    labels: # tags
        app: myapp
spec:
affinity: # affinity
PodAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
    ...
    topologyKey: kubernetes.io/hostname
containers:
-   name: myapp-container    # name
    image: busybox          # image
    imagePullPolicy: IfNotPresent# imagePullPolicy
```

```
    command: ...        # container startup command
    env:        # environment variable
    - ...
    livenessProbe: # health check
    ...
    ports:      # ports
    -     containerPort: 80
          protocol: TCP
    resources:      # resource limit
    ...
    volumeMounts:    # disk mounting
    - ...
initContainers:     # init container configuration
- command:
volumes:      # defining the disk
- ...
tolerations:    # toleration configuration
```

3.5.1.3 Lifecycle

It is defined by the phase field in a PodStatus object. The following are possible values of the phase:

- Pending: The Kubernetes system has accepted a Pod. But one or multiple container images are yet to be created. It may take some time to schedule the Pod and download the images from the network.
- Running: The Pod, in which all containers are created, has been bound onto a node. At least one container is already running, or is starting up or restarting.
- Succeeded: All containers in the Pod have stopped and will not restart.
- Failed: All containers in the Pod have stopped and at least one of them stops due to failure. In other words, the container exits with non-zero status or is stopped by the system.
- Unknown: The Pod status is unavailable due to certain reasons and it is typically a result of failed communication with the host.

A PodStatus object contains a conditions array, of which each element carries a type and a status field. A type field is a character string, with such possible values as PodScheduled, Ready, Initialized, and Unschedulable. A status field is also a character string, with possible values including True, False and Unknown.

3.5.2 Controllers

3.5.2.1 Workload Types

Kubernetes divides workloads into different types that match respective controllers in the controller-manager and are applicable to diverse scenarios.

Figure 13: Workload Types

- Deployment

  It is recommended for stateless applications. That is, you need not maintain the workload states. A Pod managed by a Deployment workload is considered independent and processable. If the Pod reports a problem, Kubernetes will delete it and create a new Pod. An instance for such applications is the Nginx Web Server.

- StatefulSet

  In contrast to Deployment, it is recommended to apply StatefulSet when an application needs to maintain its identity and store data. Zookeeper is one of such applications that save states.

- DaemonSet

  A DaemonSet workload ensures that each node in a cluster runs a copy of a Pod. Such workloads akin to daemon processes achieve the optimal effects for applications that collect logs (such as fluent) or monitor node performance (such as node-exporter).

- Job

  It starts one or multiple Pods and ensures that a specified number of them stop successfully. It is best to employ Job for some particular tasks including report creation, rather than management of applications that need to keep running.

- CronJob

  It automatically runs according to the definition-based cron schedule.

3.5.2.2 Kubernetes Controller

The Controller Manager, a control center in a cluster, manages the node, Pod copy, Endpoint, Namespace, ServiceAccount, and ResourceQuota. When a Node crashes by accident, the Controller Manager detects it in time and executes the automatic repair procedure to ensure that the cluster remains in the expected working status.

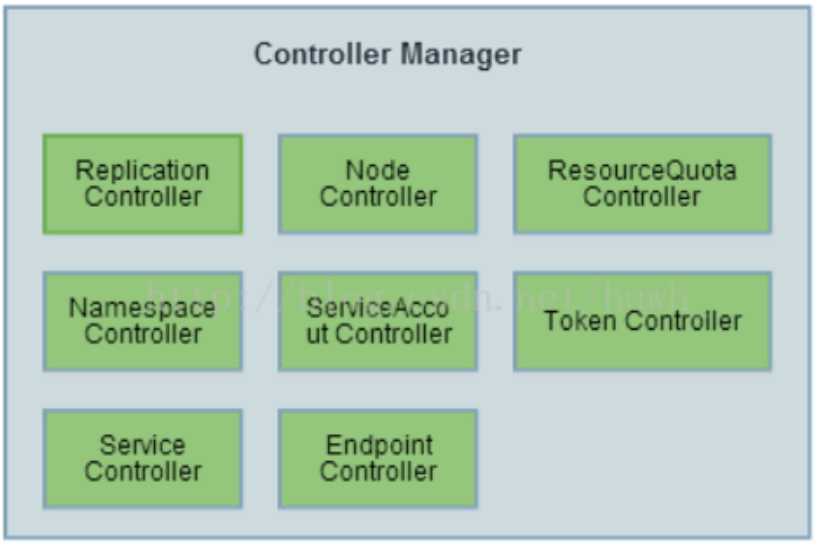

Figure 14: Controller Manager

The Controller consists of two major parts:

- The module that interacts with the kube-apiserver: Since we interact with kube-apiserver through HTTP, the module is generally the HTTP-Client encapsulated by SDK.
- The module that processes service logic:

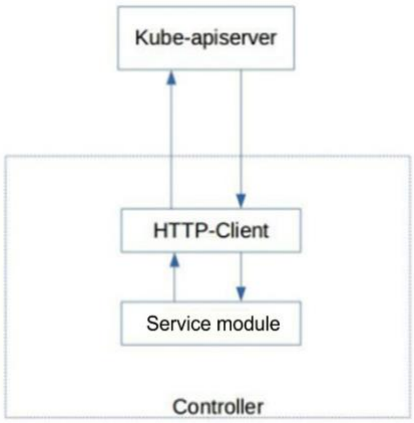

Figure 15: Interaction between the Controller and Apiserver

The Deployment controller workflow

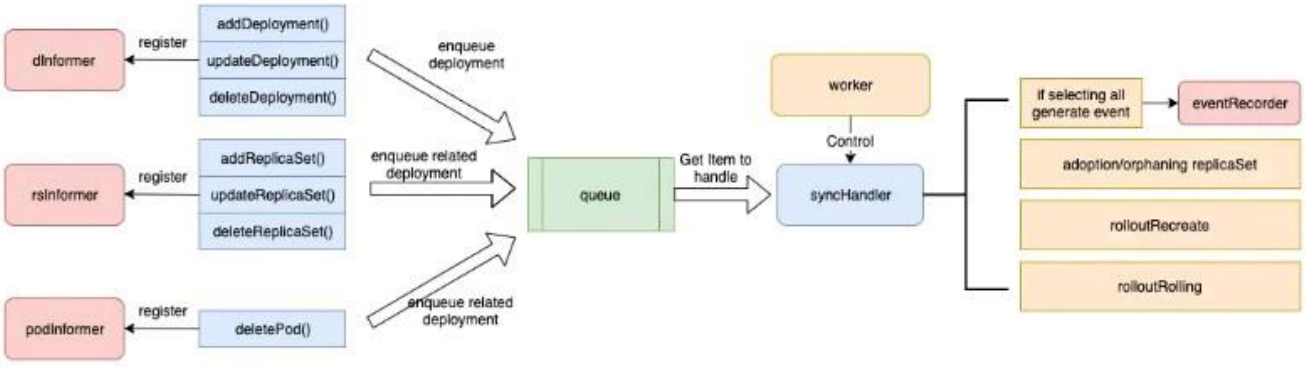

Figure 16: Controller Workflow

- Use the list-watch mechanism to monitor changes in deployments, replicaSets, and Pods.
- Put events into the first-in-first-out workqueue.
- The worker extracts items from the workqueue to execute syncHandler.
- Logics, including management of the number of copies, are processed in Handler.

3.5.3 Scheduling and Eviction

In Kubernetes, scheduling means to assign Pods onto proper nodes for the Kubelets of these nodes to run the Pods.

3.5.3.1 kube-scheduler

kube-scheduler is the default scheduler of Kubernetes and a Controller. It watches for newly created nodes that have not been assigned to Pods. This involves two steps:

- Filtering.
- Scoring.

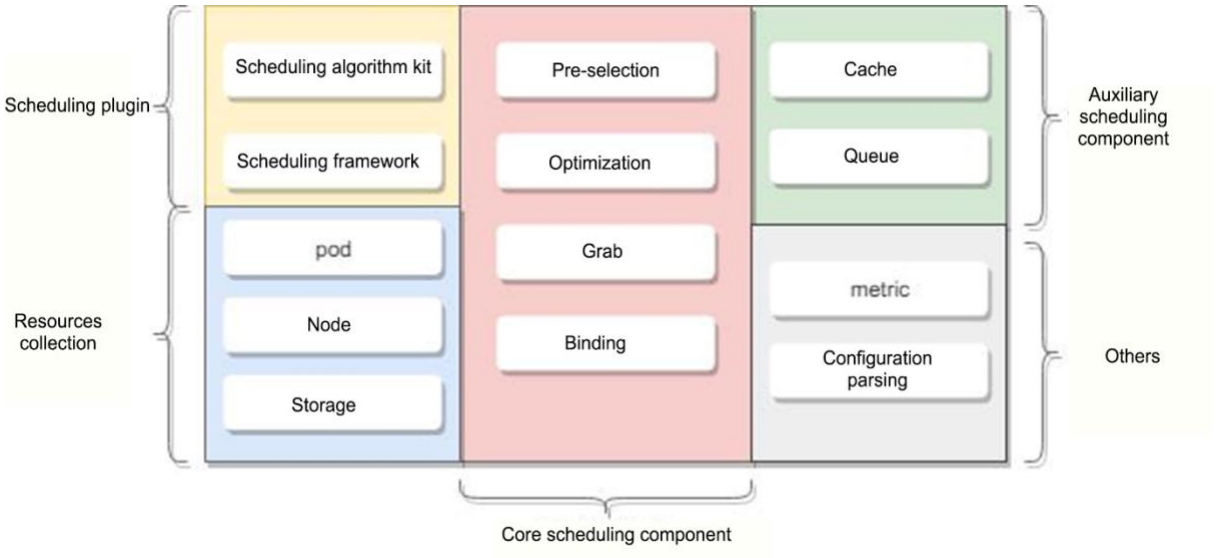

Figure 17: Scheduler Structure

Common pre-selection filters:

- PodFitsHostPorts: Checks if there is a port on the host requested by a Pod.
- PodFitsHost: Checks if a Pod specifies the hostname.
- PodFitsResources: Checks resources such as CPU and memory.
- PodMatchNodeSelector: Checks if a Pod specifies the node label.
- PodToleratesNodeTaints: Checks the tolerations defined by a Pod and the taint of a node.
- CheckVolumeBinding: Assesses if a Pod can adjust according to the volume that it requests.

Common optimization filters:

- InterPodAffinityPriority: Schedules two or more Pods onto one or different nodes according to the affinity and anti-affinity of the Pods.
- NodeAffinityPriority: The affinity and anti-affinity of a node.
- TaintTolerationPriority: Creates a priority list based on the number of taints that a node

cannot tolerate.

- ImageLocalityPriority: The node with a local image has a higher priority.
- BalancedResourceAllocation: The node where resources are used evenly has higher priority.

For more contents about affinity and anti-affinity of Pods, please refer to “affinity and anti-affinity”.

3.5.3.2 Taints and Tolerations

Taints and tolerations work together to ensure that Pods are not scheduled onto improper nodes. Tolerations are applied to Pods and taints to nodes.

For more contents about taints and tolerations, please refer to “taint and toleration”.

Application scenario:

- Dedicate a set of nodes for a specified user group and you can add a taint to those nodes.
- Special hardware devices.
- Eviction during maintenance of nodes.

3.5.4 Services and Load Balancing

3.5.4.1 Services Discover

The Kubernetes Service resource is a set of Pods with the same functions that are provided with single and unchanged network resources. When Service resources exist, their IP addresses and ports remain unchanged. You can make your Pods available for external access through Service resources or usage in the cluster only. A service discovery is created for each workload. The service discovery uses the following naming convention to enable DNS resolution for the container of workloads: <workload> .<namespace> .svc.cluster.local.

Each node of the k8s cluster runs a kube-proxy service, which implements a form of virtual IP address for Services other than ExternalName. A kube-proxy has three modes: userspace, iptables, and ipvs. When a kube-proxy detects the addition and deletion of Services and Endpoint objects through the watch mechanism, it creates the iptables or ipvs rules to import the network traffic into the backend Pods of Services.

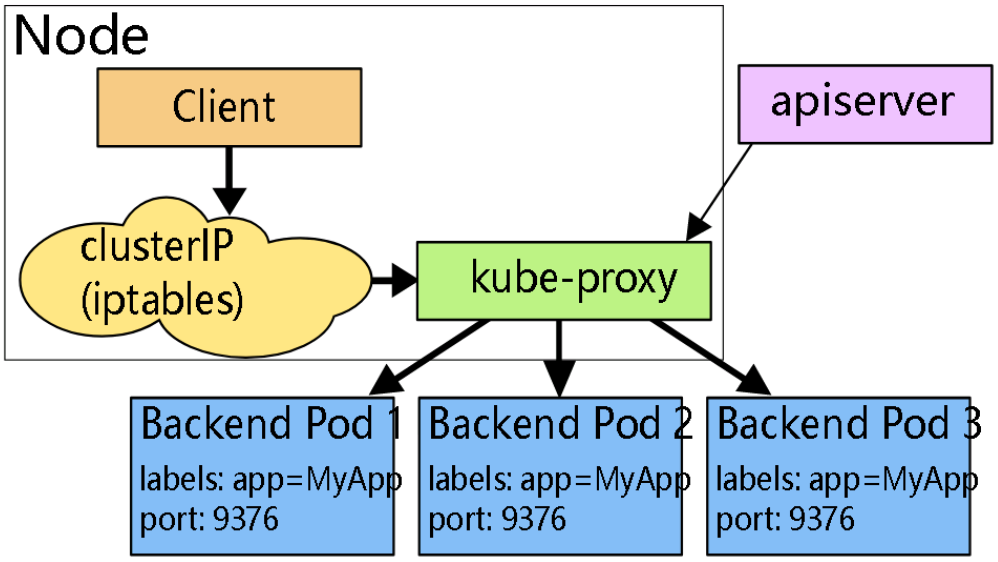

Figure 18: Service Discovery Workflow

There are four Service types in Kubernetes:

- ClusterIP: The default type internally used in a cluster.
- NodePort: Each node reserves this Service port.
- LoadBalancer: Uses cloud service providers' load balancers at a high cost as each balancer can only process one IP address.
- ExternalName: Is only supported by kube-dns version 1.7, CoreDNS version 0.0.8, or later versions.

3.5.4.2 Load balancing

Load balancing can also use ingress to expose Services to external access. Ingress is not a Service type, but as a resource type of k8s. It is a way for external access. In KubeManager, ingress and ingress controller are provided by Nginx.

For more information about ingress and ingress controller, please refer to the "KubernetesIngress Documentation".

3.6 Application Store

After the deployment of applications, one of the benefits of application store is that, compared with managing workloads/resources separately, it allows easy and unified management of many workloads and resources. The application store also enables clone, update and rollback of applications.

3.6.1 Deployment

3.6.1.1 The Architecture

KubeManager provides the application store function based on Helm 2.0. This function renders deployment and management of a same application much easier. The application store can be a GitHub codebase or a Helm Chart repository with applications that can be deployed. The applications are packaged in the object named Helm Chart.

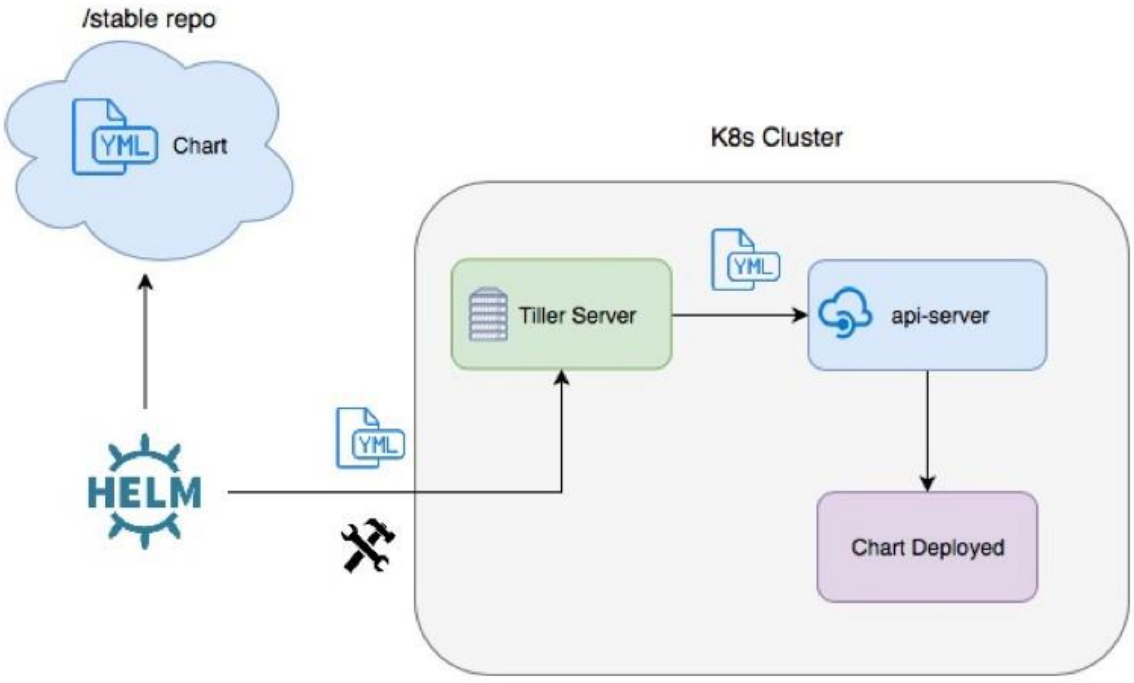

Figure 19: Harbor Structure

- Functions of Helm Client
  - Development of the local chart packages.
  - Repository management.
  - Communication with Tiller Server, such as creation, update and deletion of applications.
- Functions of Tiller Server
  - Processing requests from the client.
  - Communicating with the kube-apiserver.

3.6.2 Application Upload and Management

3.6.2.1 ChartPackage Introduction

Similar to an rpm and dpkg package, a Chart package has its format convention, which is composed of a catalog and a series of files in the catalog. The name of the catalog is the same as that of the Chart package. Take wordpress for instance:

```
wordpress/
    Chart.yaml    # Chart package information
    LICENSE       # OPTIONAL: A plain text file containing the license for the chart
    README.md     # OPTIONAL: A human-readable README file
    requirements.yaml    # OPTIONAL: A YAML file listing dependencies for the chart
    values.yaml   # The default configurations of this Chart package
    charts/       # The Chart package on which this Chart package is dependent
    templates/    # The Kubernetes file list used with values.yaml
    templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

For more information on Chart packages, please refer to “Chart”.

There is no Tiller Server in the Helm 3.0 as the Client directly communicates with Kubernetes.

3.6.2.2 Application Management

KubeManager provides a harbor image repository that can be used as a Chart repository at the same time. Application upload steps:

- Chart packaging: Identify the application package version according to the Chart.yaml of the catalog and create a .tgz file, that is, a Helm package. You can use the –sign parameter to create a .tgx.prov file.
- Upload: You can upload through Helm Client or the page.

3.6.2.3 Application Store Management

KubeManager provides three layers of application store: global, cluster and project. You need certain permissions to use the application store. For example, you need to have one of the following permissions to create a project-level application:

- The role as a project member of the target cluster: You can create, read, update, and delete workloads.
- The role as the owner of the target cluster: You can operate each application store with ease, such as cloning and updating.

3.7 Multi-Cluster Applications

In general, most applications are deployed in a single Kubernetes cluster. A multi-cluster application enables the deployment of multiple copies of an application in different clusters and/or projects. In KubeManager, a multi-cluster application uses Helm Chart and can deploy an application in multiple clusters. Therefore, it helps avoid human errors that may occur when a same operation is executed on each cluster. With a multi-cluster application, you can ensure that an application has identical configurations across all projects/clusters and override different parameters according to the target projects. Multi-cluster applications are regarded as a single application and are therefore convenient for management and maintenance.

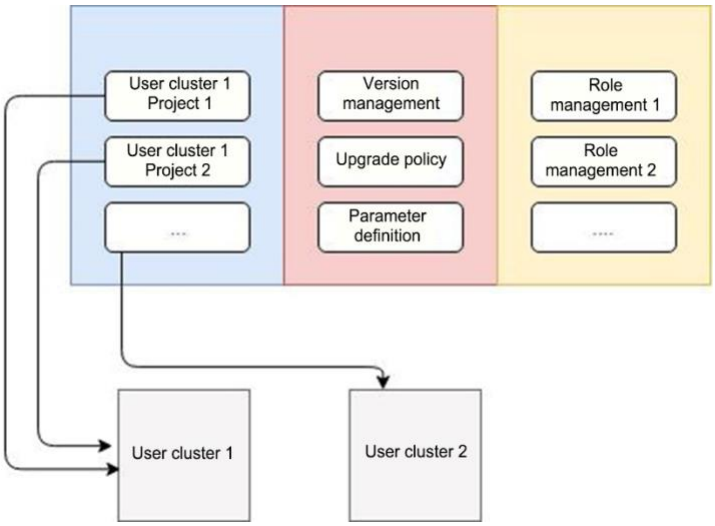

Figure 20: Multi-Cluster Applications

3.8 Network Ports and Layer 4 Load Balancing

3.8.1 Network Ports

3.8.1.1 Functions of Network Ports

NodePort and Loadbalance are the common methods to use Kubernetes to publish a service. Their respective features and application scenarios are as follows: Ingress cannot publish services of protocols other than layer 7 http/https services. In addition, it cannot provide an integrated and highly available IP address for access except with the support of an external load balancer. Such an access method of IP-NodePort is inconvenient for maintenance and without a unified IP address. The services will be immediately unavailable if the node that you access crashes, unless you switch the client to another IP address. Loadbalance is a better way to publish services. However, an external balancer is required that has been connected to the Kubernetes cluster, and can automatically detect Ingress configurations and publish services. Otherwise, you will have to configure again on Loadbalance whenever you publish a service.

To address the shortcomings mentioned above, KubeManager provides network ports to offer a highly available IP address for service access and the layer 4 load balancing to meet requirements of publishing services other than http/https ones.

3.8.1.2 Principles of Network Ports

The network ports of KubeManager are implemented with Keepalived. You can choose two worker nodes to form a Keepalived cluster. KubeManager automatically deploys the Keepalived cluster on the nodes. As an application in Kubernetes, the Keepalived configures floating IP addresses by sharing the network with the Host. In addition, the Keepalived cluster is responsible for managing floating IP addresses. This allows you to access services published externally only through the floating IP address with port.

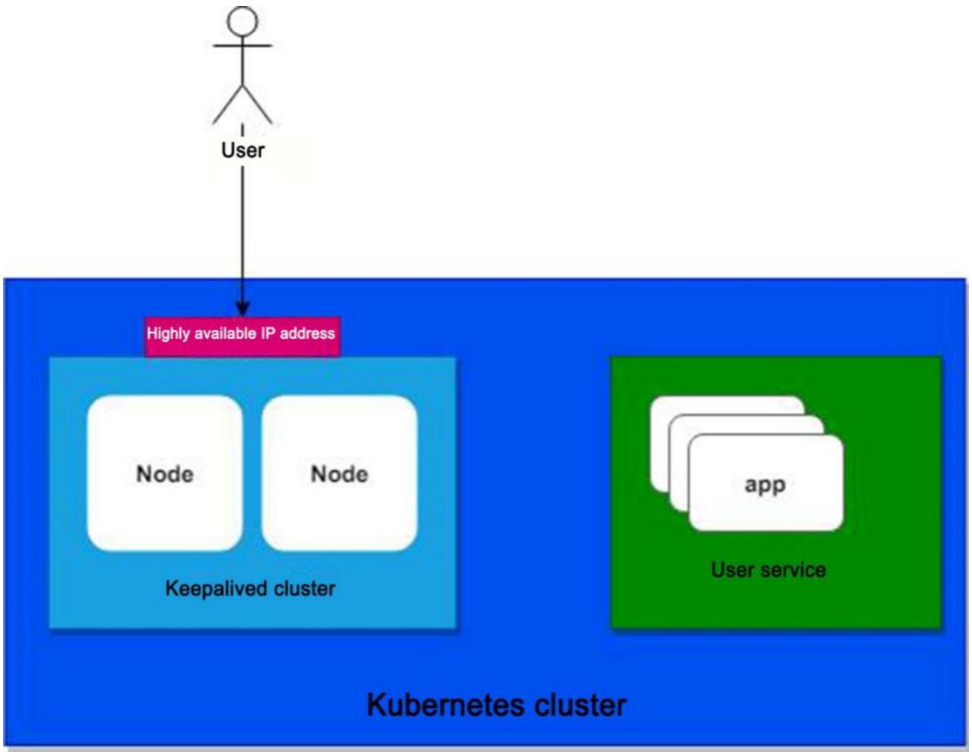

Figure 21: Network Ports

3.8.2 Layer 4 Load Balancing

3.8.2.1 Functions of Layer 4 Load Balancing

In general, Kubernetes publishes layer 4 services through nodePort. But nodePort is not applicable to manufacturing environments:

- It is difficult to manage services and the port of the NodePort needs to be recorded after a service is published.
- The default range of the port number of NodePort is from 30000 to 32767. It is not a standard service port and requires modifications to service configurations or codes before being accessed.
- A port can only publish one service.

To address the pain point in publishing non-http/https services, KubeManager provides layer 4 load balancing so that users can easily publish layer 4 services to external ends through the KubeManager interface. You can publish transmission control protocol (TCP) and user datagram protocol (UDP) services on a port at the same time and ensure that the traffic goes into the cluster through a specified node with the highly available IP address of the network port.

3.8.2.2 Principle of Layer 4 Load Balancing

The layer 4 load balancing provided by KubeManager is implemented through Nginx, which is deployed on each Kubernetes node by KubeManager. Incoming traffics over the floating IP address on the network port are forwarded to a floating IP address to directly enter the Nginx of that node. The Nginx distributes the traffic load to a specified service's Pod directly. This reduces performance loss by bypassing the native Service load balancing of Kubernetes.

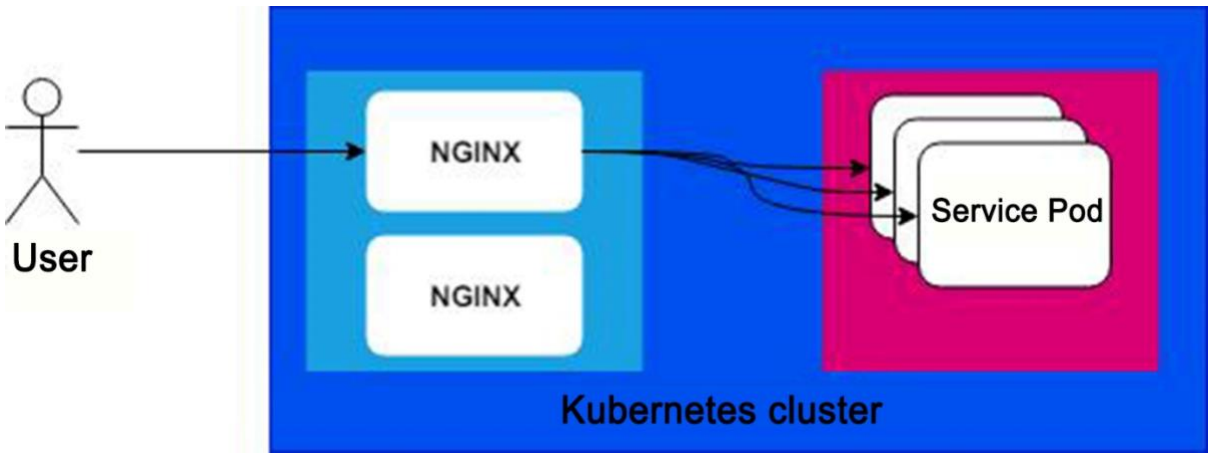

Figure 22: Load Balancing

3.8.2.3 Advantages of KubeManager

Using the network port and layer 4 load balancing of KubeManager as described in previous sections has the following advantages:

- Costs are reduced, as users do not need to provide external load balancing.
- Traffics are more controllable over centralized ingress and egress ports. The specified nodes are responsible for north and southbound traffics to increase cluster stability.
- The administrator can easily manage and configure all layer 4 load balancing settings on KubeManager.

3.9 Storage

3.9.1 Storage Server

3.9.1.1 Functions of Storage Server

To simplify the configuration workflow of storage classes and static PVs, the KubeManager abstracts the concept of storage server. Currently, NFS and Sangfor aSAN are supported and only the system and cluster administrators are allowed to configure the storage server. After the NFS or Sangfor aSAN storage server is configured, KubeManager automatically deploys the CSI plugin of NFS provisioner and Sangfor aSAN. It can support NFS and Sangfor aSAN to dynamically create PVs using storage classes.

3.9.1.2 Advantages of NFS/Sangfor aSAN Dynamic Provisioning

The NFS provisioner deployed by KubeManager can support multiple NFS servers simultaneously. It saves users' Kubernetes resources, relative to open-source NFS provisioners that only support a single NFS server and therefore require deploying multiple provisioners to support multiple NFS servers.

Sangfor aSAN storage improves service data reliability based on the stability and high performance of Sangfor's commercial storage. The Kubernetes CSI plugin developed by KubeManager relying on Sangfor realizes the correlation between Sangfor aSAN storage's functions and Kubernetes. This enables user applications to leverage advantages of the aSAN commercial storage to improve service reliability. The architecture principle of the Sangfor aSAN CSI plugin is shown in the figure below:

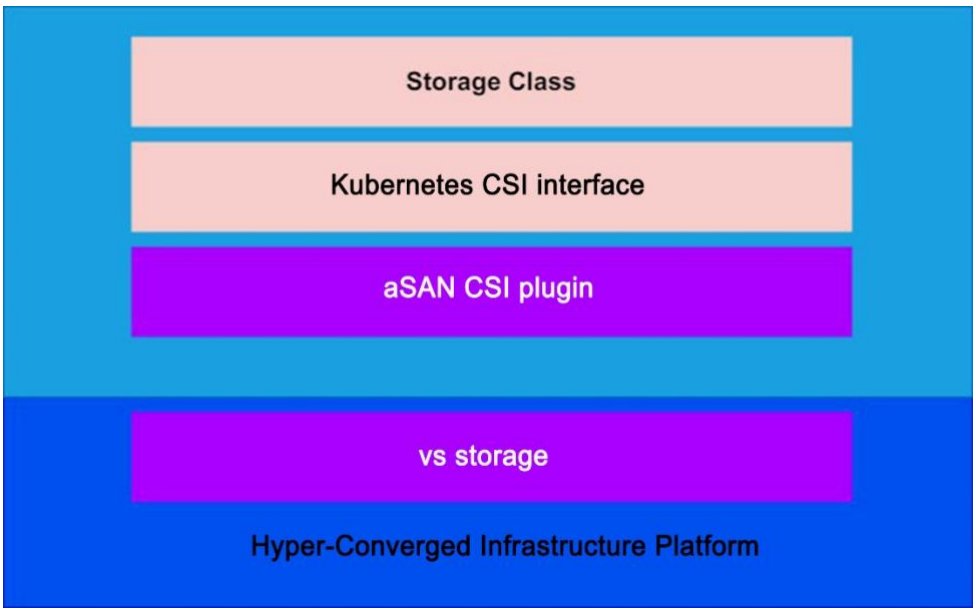

Figure 23: Sangfor aSAN CSI Plugin

For Kubernetes CSI extensions, refer to “Kubernetes CSI Documentation”.

3.9.1.3 Use of Storage Server

Users can select the storage server corresponding to the given storage, when configuring static PVs and storage classes at the cluster level. Information of the storage server is automatically added into the static PVs and storage classes and submitted to Kubernetes, significantly simplifying and speeding up configuration of PVs and storage classes for users.

3.9.2 Storage Volume and Class

3.9.2.1 PersistantVolume Introduction

A PersistentVolume (PV) is a uniform abstraction of storage by Kubernetes. Whereas a PersistentVolumeClaim (PVC) is a claim of request of an application for storage. Persistent storage can be used in Kubernetes in two methods:

- Using an existing PV.
- Configuring a new dynamic storage PV.

To use existing PVs, the application should use the PVCs already bound to the PVs that have the minimum resources (capacity) necessary for the PVCs. For example, the Kubernetes cluster is already bound to a PV, to which it sends a request for storing 5-gigabyte data. If the PV's remaining capacity is greater than 5 gigabytes, data can be stored in this PV. Otherwise, users have to configure a new dynamic storage PV, since the resources necessary for the PVC exceed those that the PV can provide. For dynamic storage configuration, the application should use PVCs already bound to storage classes that have permissions to create a new PV.

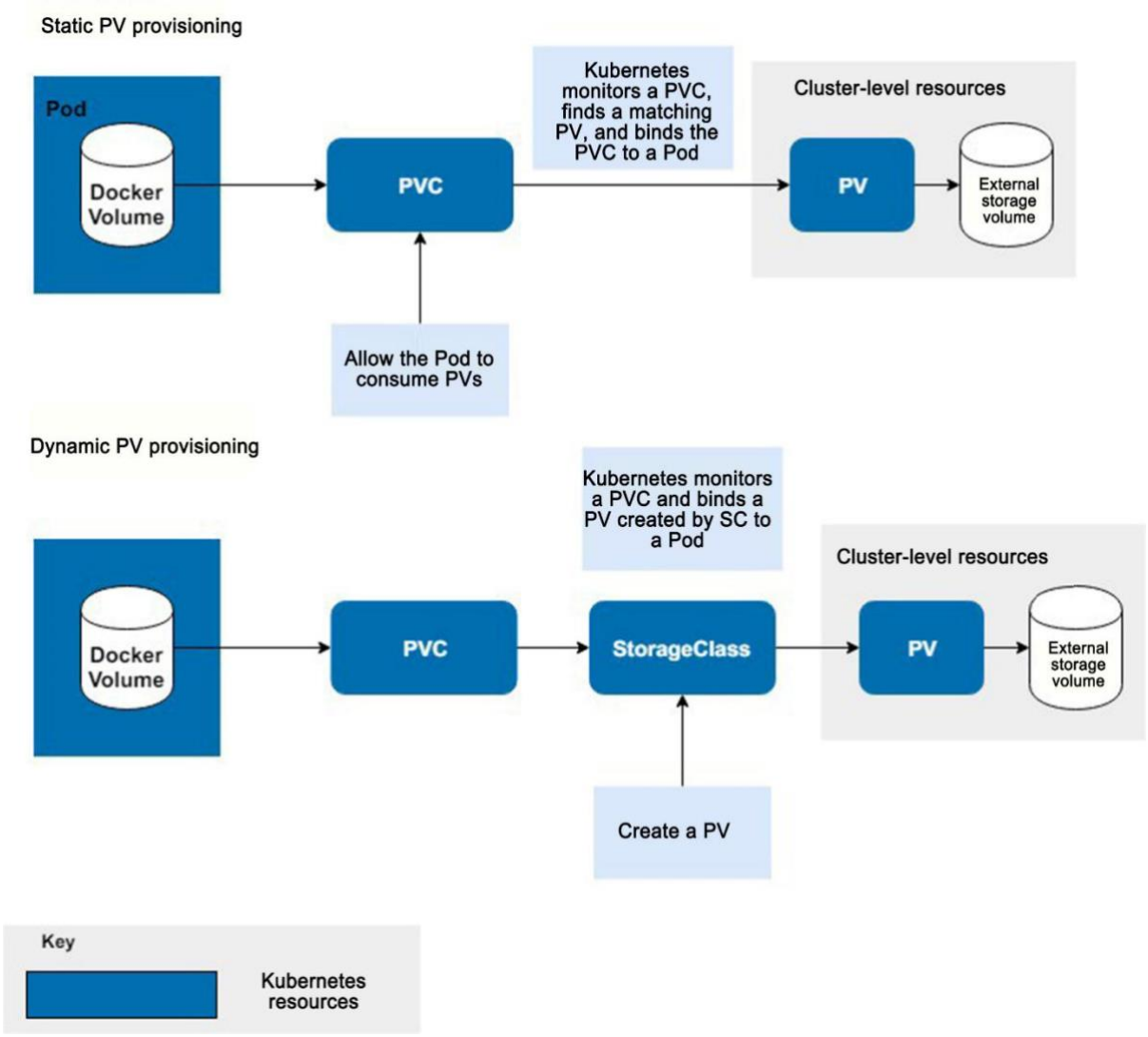

Figure 24: PV

For more information, please refer to "Official Documentation of Kubernetes Storage".

3.9.2.2 PVC Introduction

PVCs are objects that send requests for storage resources to the cluster. They are analogous to certificates for your deployed applications to access storage. PVCs are mounted as volumes in workloads so that these workloads can claim their specified quota persistent storage. To access persistent storage, a container should have a PVC mounted as a volume. The PVC allows you to store application data—which is deployed in the Pod—outside the Pod. In case of a Pod failure, you can enable a new Pod to access application data in external storage, avoiding application interruption. All projects in KubeManager contain the lists of PVCs you have created. To view these PVC lists, go to the navigation bar, click "Resource", expand the drop-down list, and click Workload to PVC. Created PVCs can be reused.

3.9.2.3 PVC Required by Both New and Existing Persistent Storage

PVC is the prerequisites for Pods to use the persistent storage. That is, PVC needs to be mounted, irrespective of whether the Pods will use existing storage or configure new storage. To set up a static PV for a workload, the workload should be mounted with a PVC that points to an existing PV and corresponds to the existing storage infrastructure. To configure new storage for a workload, the workload should be mounted with a PVC that refers to a storage class for creating a new PV. KubeManager allows you to create any number of PVCs in a project. You may mount a

PVC on a workload either during workload creation or after running the workload.

3.9.2.4 Setting up Existing Storage based on PVC and PV

A Pod can store data generated during its runtime in a container. However, the data may lose when the Pod fails. To address this problem, Kubernetes provisions PV, which are external storage disks accessible by Pods or Kubernetes resources corresponding to the file system. A Pod stores application data in an external location. If a running Pod crashes, users may use a new Pod to access data in the persistent storage, minimizing the influence of the breakdown.

The PVs can be either local physical disks or file systems, or storage resources hosted by cloud suppliers, such as Amazon EBS or Azure Disk. In KubeManager, PVs and storage volumes are created in two independent processes. Creating a PV does not create a storage volume. Instead, it creates a Kubernetes resource and maps it to an existing volume. Therefore, storage must be configured before a PV is created.

Note: PVs are created on the cluster level. A cluster may be stored in several projects and namespaces. In a multi-tenancy cluster, multiple users can access a same PV.

3.9.2.5 Binding PV to PVC

During the setup of persistent storage for a Pod, a PVC will be mounted in the same way as other Kubernetes volumes. While creating the PVC, Kubernetes Master regards it as a request for storage and binds it to a PV that matches with the minimum resource (capacity) requirements of the PVC. Not all PVCs can be bound to PVs. According to “Kubernetes Documentation”: If the matching volume does not exist, a PVC will remain in the unbound state. A PVC is in the bound state only when there is a matching volume. For example, a cluster has many 51Gi PVs, but none of them matches a 100Gi PVC. The PVC will be bound when a 100Gi PV is added into that cluster. You may create countless PVCs. But they will only be bound to PVs which Kubernetes finds to at least match disk capacity they require. To dynamically provision new storage, the PVC mounted in the container must correspond to a storage class rather than a PV.

3.9.2.6 Using PVCs and Storage Classes to Set up New Storage

Storage classes allow you to dynamically create PVs instead of creating persistent storage in the infrastructure. For example, if a workload is bound to a PVC which refers to a Sangfor aSAN storage class, the storage class can dynamically create a Sangfor aSAN volume and the corresponding PV. After that, Kubernetes Master will bind the created PV to the PVC of the workload, so that your workload can use persistent storage.

3.10 Monitoring and Alerting

3.10.1 Principle of KubeManager Monitoring

KubeManager applies prometheus-operator to deploy Prometheus for monitoring data collection. It deploys Grafana for data display and AlertManager for sending alarms. KubeManager integrates common monitoring parameters to the UI, making it convenient for users to view the parameters. The global view of the monitoring system is shown below:

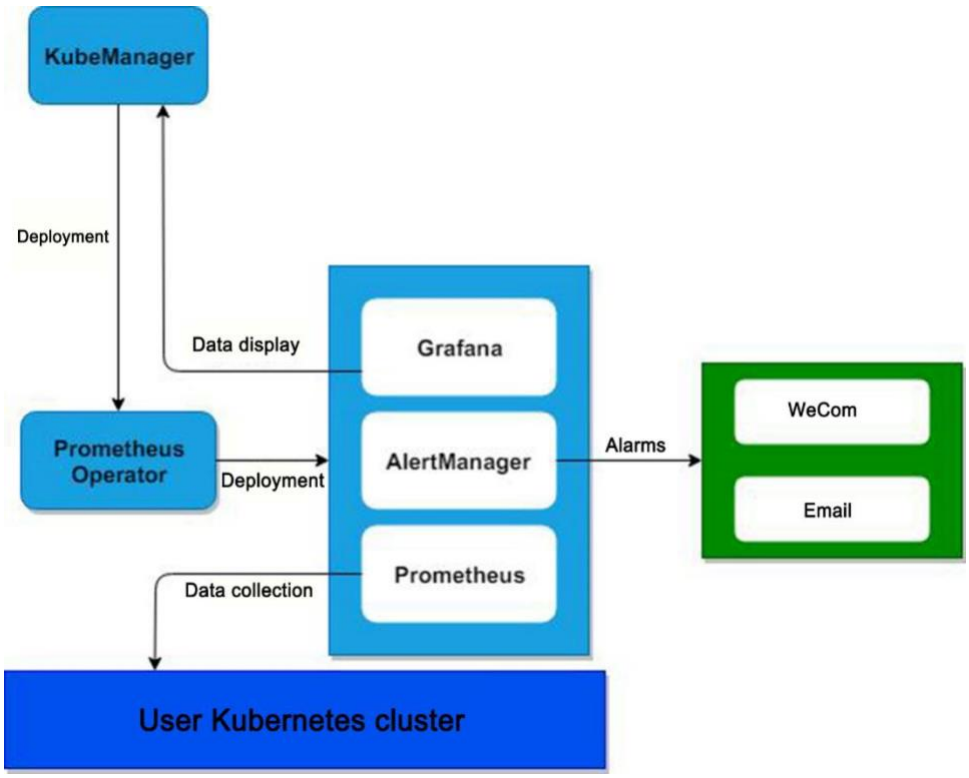

Figure 25: Global View of Monitoring System

When a user enables monitoring on the interface, KubeManager automatically deploys prometheus-operator and relevant monitoring configurations in the corresponding Kubernetes cluster. And the prometheus-operator automatically delivers components information to the Prometheus configuration. Prometheus collects data of relevant components as configured. Currently, data of the following components are collected:

- Kubelet.
- Kube-scheduler.
- Coredns.
- Etcd.
- Kube-controller-manager.
- Fluentd.
- Node (node information).
- Kubernetes' system information.

After collecting these data, Prometheus determines to place the monitoring information in either the memory or persistent storage, depending on the user configuration. If the monitoring system triggers the alarm rule configured by the user, the AlertManager will send the alarms to the user as configured. The user may view the monitored information on the KubeManager interface. To view more detailed or custom information, the user may go to the Grafana interface via the KubeManager.

3.10.2 Features of KubeManager Monitoring

The advantages of KubeManager monitoring and alerting system are as follows:

- KubeManager monitoring and alerting system employs an open-source combination including Prometheus and Grafana. It is integrated to the app store of KubeManager.

While facilitating one-click deployment by users, it integrates the most important monitoring data to KubeManager's UI, reducing operation costs for users. This not only addresses users' needs for custom indices by retaining Grafana's powerful custom monitoring indices, but also makes it convenient for users to view common monitoring data.

- Since Grafana's service is not exposed, to view Grafana's data, users need to login to the interface of KubeManager and access by KubeManager proxy.

3.11 Logs

3.11.1 KubeManager Logging Introduction

The log components of KubeManager fall into two categories:

- ElasticSearch, Kafka and etc. that connect to external systems and in which KubeManager stores application logs it only collects.
- The log collection and storage system provided by the KubeManager system out of the box.

If a user already has a log storage system and wishes to correlate application logs deployed and managed by KubeManager with existing logs, the user may utilize the log docking function. Currently, external systems supported by the KubeManager are as follows:

- ElasticSearch.
- Splunk.
- Kafka.
- Syslog.
- Fluentd.

If users have no externally deployed log systems, they may utilize Sangfor Logging, a log collection and storage function provided by KubeManager which significantly lowers the difficulty of deployment for users. KubeManager's log system can functions at either the cluster or project level. Cluster-level logs collect all of those at the cluster level, and only the cluster administrator has the permission to enable and disable the log collection module. Project-level logs collect those at the project level, and the project administrator can enable/disable the log collection module.

3.11.2 Log Collection Principle

3.11.2.1 Collection Principle of External Logs System

When docking with the log module of an external system, KubeManager uses fluentd to collect logs. After a user starts log collection and completes relevant configurations, KubeManager will automatically deploy fluentd in daemonSet mode in the cluster. And it can deliver relevant configurations to fluentd. The latter then collects all logs under directory /var/lib/docker and distributes the logs to corresponding storage backend. The workflow is shown in the figure below:

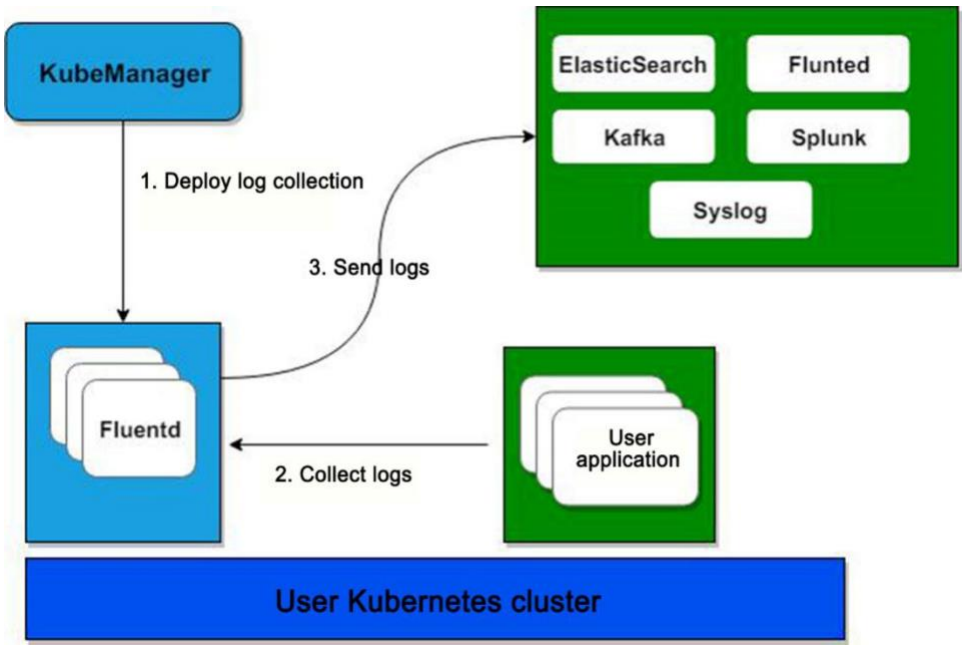

Figure 26: fluentd

3.11.2.2 Collection Principle of Sangfor Logging

The Sangfor Logging log collection system is realized through the open-source EFK solution. That is, the collection system is implemented by fluentd, the storage system by ElasticSearch, and log query by Kibana.

- Users can access the management interface of Kibana by using proxy on KubeManager. The web interface of Kibana is not directly exposed on public networks, which ensures data security. Meanwhile, KubeManager implements complete verification of user permissions. Only legal users can be directed to the management interface of Kibana.
- KubeManager determines the number of nodes of the ElasticSearch cluster, based on the user cluster scale and whether ElasticSearch uses shared storage.
- By transforming fluented and monitoring events on Docker, Sangfor Logging can collect log files in containers.

The overall architecture of Sangfor Logging is shown in the figure below:

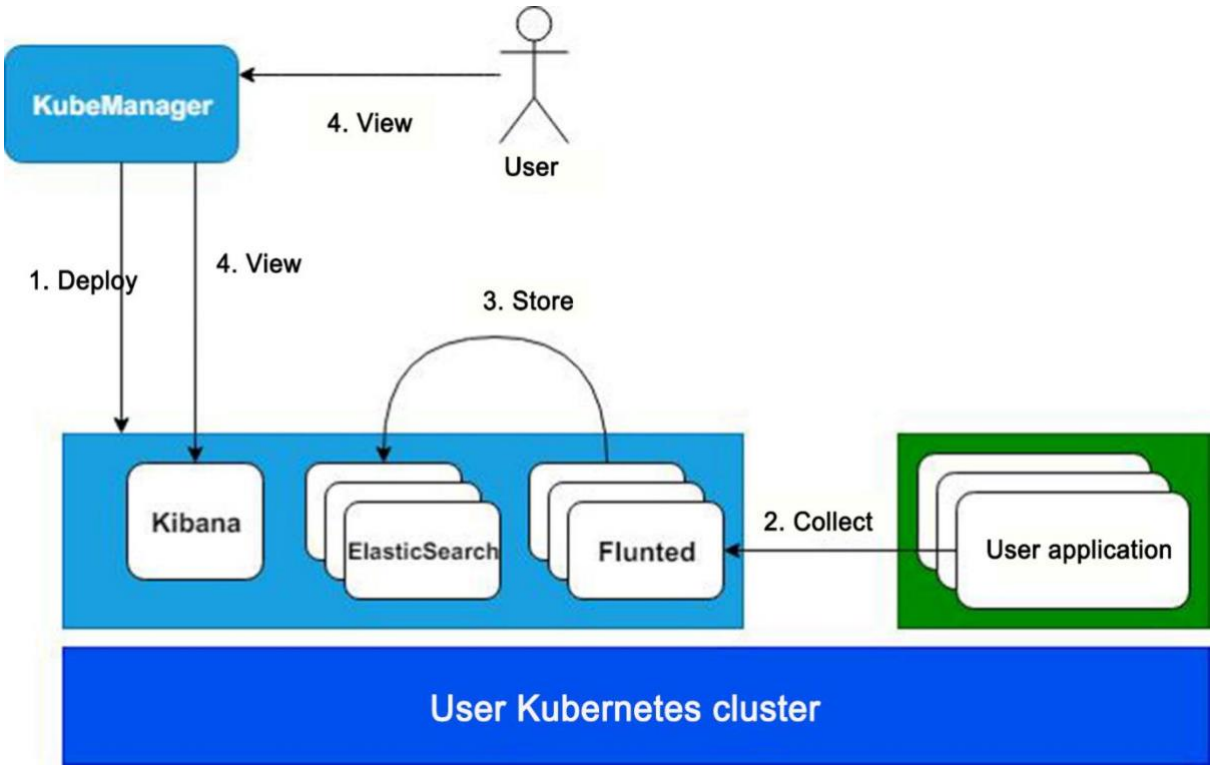

Figure 27: Sangfor Logging

3.11.3 Features of Sangfor Logging

The collection system of Sangfor Logging has the following features:

- Provides high availability for log storage. Sangfor Logging automatically determines the number of nodes to be deployed for ElasticSearch, based on user configuration. This means to provide high availability at the cost of minimal resources.
- Collects log files in containers, prevents users from changing service configurations or codes, and lowers the difficulty of containerization for users. This means that users only need to enable log collection and enter the path of the log file in a container when deploying a workload.
- Ensures data security. Services of Kibana and ElasticSearch are not exposed on public networks. To view log data, users need to login to KubeManager and be authenticated by it. Then users use its proxy to access Kibana and ElasticSearch, which ensures data security.