Michael's Spec

Operate Contrail

An abstract on operating system state using Contrail k8s Operator

Author: Michael Henkel

Introduction

Contrail is a distributed system which is built around a set of other distributed systems. There are different kinds of dependencies between but also within the systems. These dependencies create challenges operating the whole system in a consistent manner.

Operate

The current tools used are mainly focused on deployment and provide little to no support for real lifecycle management of the full system. Lifecycle requirements are defined by CRUD (create, read, update and delete) operations.

C(reate)

It must be possible to add elements to the system with the least amount of impact. Adding dataplane elements is fairly easy as they are not clustered nor are other systems depending on them.

Adding control plane elements is more challenging as other systems depend on them. Ie. When the Cassandra cluster is scaled by one node, Contrail Config, Kubemanager, Analytics and Control must be aware of that and make use of the new node.

R(read)

It is important to have a reliable status of the health of the system. It must be easy and straightforward to see that the system is not only up and running but also functions within defined parameters.

U(pdate)

During the lifecycle of the system, parameters will be changed. A create operation for example on one layer causes parameter updates on a dependent layer. However, there is a distinction between mutable and immutable parameters. These are different from system to system.

D(elete)

Deletion of an element from the system can have two causes: a) intentionally, b) accidentally. Both cases will have to be handled differently based on the characteristics of the system.

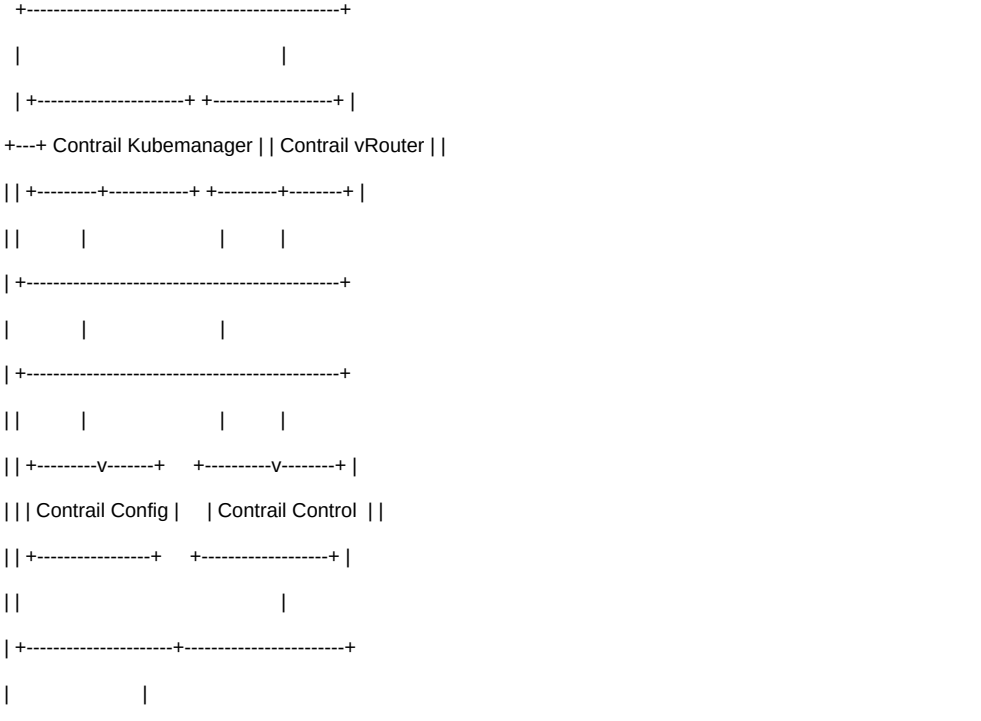
Anatomy of a system

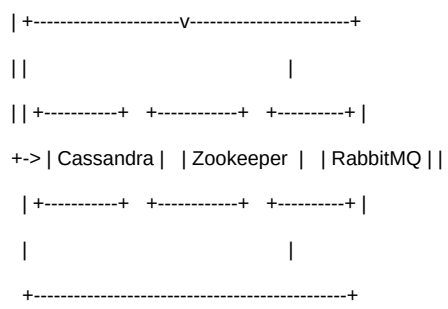
Each system in Contrail handles CRUD operations differently. In order to provide a tool which can operate the entire system consistently it is important to know

1. the dependency between the systems
2. how to operate each system

Dependency

The distributed system of Contrail can be roughly separated into three layers of dependencies, where higher layers depend on lower layers.





Certain operations on a layer requires changes on the dependent layer. The way these changes become active depends on the particular system.

System operation

Each system has to be operated differently. In order to operate the system as a whole the operation tool must know all that differences.

Zookeeper

Prior to version 3.5 Zookeeper didn't support runtime configuration changes at all. 3.5 introduced the dynamic addition/removal of nodes. Therefore the configuration is split into an immutable static part and a mutable dynamic part.

In order to be able to add and remove nodes, the tool must be able to change the dynamic configuration. The dynamic configuration part is the same for all nodes in the cluster and as such can be shared between the nodes. Furthermore Zookeeper does not allow to scale a cluster which started with a single node. The minimum number of nodes to start with is three. From there on new nodes can be added. If an environment starts with a single node and must scale to three nodes, Zookeeper has to be restarted.

Cassandra

A Cassandra cluster is created by bootstrapping one or more seeds nodes. The seeds node count should be smaller than the total node count but also greater than one if there are more than one node in the cluster. Whenever the node count changes, the seed count can change as well. This change must be reflected in a per node configuration file (in contrast to Zookeeper where one dynamic configuration file can be shared between all nodes).

When a node is removed from the cluster it must be drained from the database.

RabbitMQ

A RabbitMQ cluster starts with an initial, first node. Adding new nodes does not require configuration changes. New nodes de-/register themselves with the initial node.

Contrail Config/Analytics Plane (Config/Analytics/Kubemanager)

Not all of the Contrail Control plane systems support dynamic reconfiguration. If the configuration changes, the particular system has to be restarted in order to load the new configuration. That means before the systems can be started initially, the final system configuration must be known. This includes its own configuration (ie. all ip addresses of Contrail Config nodes) and the configuration of systems it depends on.

Contrail Control

Contrail Control supports dynamic reload of Cassandra and Collector node configuration. However, Contrail Control does not actively watch configuration changes but must be triggered by sending SIGHUP in order to reload the configuration.

Contrail vRouter

Contrail vRouter agent supports dynamic reload of Contrail Control node configuration. However, vRouter agent does not actively watch configuration changes but must be triggered by sending SIGHUP in order to reload the configuration.

The Operator

As a consequence a tool is required which not only understands the semantics of each system but also the dependency between the systems. The tool needs to have a holistic view of the entire system. It must be able to change configurations of individual systems and reflect the impact of that change to dependenting systems.

This is where the Operator Framework comes into play. The Operator Framework utilizes Kubernetes sophisticated way of scheduling containers whilst allowing to interact with the services at any stage of the lifecycle in a programmatic and system specific way. Prior to Operator Framework most tools (Ansible, BOSCH, Puppet, Chef etc) mainly focused on deployment with very little to no knowledge of system semantics.

Each system is defined as a controller, the controller is orchestrated by the operator. Controllers can exchange state information between each other. Each controller has system specific knowledge and understands the semantics of the system it is controlling.

Cassandra Controller

Cluster Creation/Node addition

The Cassandra controller understands that the seeds list must be changed whenever a new node is added. Because it knows the current and the intended state, it can calculate the required number of seeds. This information must be provided to existing nodes and to the new node. Therefore the configuration file of all existing nodes must be changed by the controller. The way Kubernetes allows to change files from the outside within a container is through ConfigMaps. A ConfigMap is a list of key/value pairs. The configuration of each node is stored as a key/value pair in a ConfigMap where the key is a node identifier and the value the configuration. The configuration is mounted inside the container and is consumed by the service. When the configuration changes, the controller identifies the key as the node identifier and adjusts the value which is the configuration. As a result the service will read the change configuration.

Status Monitoring

Kubernetes allows you to define commands which run inside the container and evaluate the return code. These Readiness probes can be used to report the health of the system. Incase of Cassandra, the Readiness probe executes a ‘nodetool status’ command and evaluates the output. Based on the result of the evaluation it either returns 0 or -1. This return code is then evaluated by the operator to set the status of the system to a defined value.

Deletion

When a node is removed from the cluster, it must be deregistered. Kubernetes provides a hook for PreStop events. This hook executes a script inside the container which drains the node from the cluster.

Zookeeper Controller

Cluster Creation/Node addition

As mentioned above, the Zookeeper dynamic configuration is the same across all cluster nodes. As such the ConfigMap is a single key/value pair which is consumed by all cluster nodes. Whenever the node count changes, the value is adjusted, which automatically updates the configuration file. All cluster nodes will read the updated configuration automatically.

The static and node specific part of the configuration is stored in a node specific key/value pair, where the key is the node identifier and value the node specific configuration.

Status Monitoring

The status is retrieved via a Readiness probe running a command inside each container which checks that each node in the cluster is serving requests. If the Readiness probe succeeds the status is set to true, if not, to false.

Deletion

Deletion of a node changes the update of the shared configuration.

RabbitMQ Controller

Cluster Creation/Node addition

A RabbitMQ cluster must be created in stages. The first node must be running, subsequent nodes must first register themselves with the first node and then start the application. The RabbitMQ controller uses a combination of techniques to provide this order. First a common configuration file is created as a ConfigMap key/value pair which contains and ordered list of nodes. When a node starts, it evaluates this file and if it is the first node, it simply starts the application. If it is not the first node, it starts a status check on the first node in a loop until the check succeeds. Once succeeded the node registers with the first node and starts the application.

Status Monitoring

The Readiness probe runs the ‘rabbitmqctl cluster_status’ command which gives back the cluster members. This information is compared against the intended state of the controller.

Deletion

The PreStop hook calls the ‘rabbitmqctl forget_node’ command which removes the node from the cluster.

Contrail Control Plane Controllers

There is a controller for each Contrail Control plane element. They all follow the same semantics. In contrast to the systems described above, some Contrail control plane services do not support dynamic reload of configuration. Whenever the configuration changes, the system must be restarted.

Cluster Creation/Node addition

For the initial cluster creation all nodes in the system need to know all configuration parameters before they start. The following considerations are important:

- The dependent layer (Cassandra, Zookeeper, Rabbitmq) might be up already or not
- The Kubernetes scheduler assigns a host based on certain parameters. As such the target host is not known until the POD is created

Based on the two considerations the creation of the cluster has to be staged. First the POD must be scheduled in order to know the host the node will be running on. Second, theSecond the deployment must be paused until the dependent layer information isare available. Once all node information of the system isare known and the dependent layer is in operation, the deployment process can craft the final configuration and start the actual service which will eventually form the system.

This staging is achieved by the combination of different methods:

A Contrail POD consists of one init container and one or more service container. When the POD is created it has an empty status label. This label is mounted as a file inside the init container. When the init container becomes active, it reads the label file in a loop. If the label file contains the string “ready”, it returns 0 and the deployment of the remaining services continues. If not the loop is continuing.

The Contrail controllers monitor the dependent services and the individual nodes of the Contrail system. The Contrail controller knows the intended state of the system. It knows how many nodes the Contrail system will have to consist of. As a consequence it monitors the instantiation of the PODs and particularly the init containers of the respective PODs. Once the amount of PODs for a system has reached the intended amount of instances and all have an IP address assigned, the controller knows all hosts the individual nodes will run on. With the information about all Pod IP addresses the controller crafts the system specific configuration and adds it to a ConfigMap using a node specific key with the configuration as the value.

Furthermore the controller checks for the status of the dependent systems.

If both conditions, dependent systems operational and all init containers have an ip address assigned, the controller changes the status label content from empty to “ready”. As a result the loop which runs in the init container will exit with 0 and the remaining deployment continues.

Node addition or update of parameters require a re-creation of the system.

Status Monitoring

The Readiness probe reads the status by sending a curl request to the services inside the container. If the return is a http 200 code, the service is marked as active.

Deletion

Deletion of the node requires a restart of the system

Contrail vRouter

Cluster Creation/Node addition

The Contrail vRouter is implemented as a Kubernetes Daemonset. A Daemonset instantiates a POD on every node which matches certain filters. The Contrail vRouter depends on the Contrail Control nodes. However, the part of the configuration which specifies the Control nodes can be dynamically updated and the Contrail vRouter will not read the updated information without a restart. This re-read is not done automatically. A SIGHUP signal has to be sent to the vRouter process.

The vRouter POD has a sidecar container which shares the PID namespace with the vRouter container. The sidecar container has a label mounted as a file and it watches that file for changes. Whenever the node count of the Contrail Control nodes changes, the operator updates the vRouter configuration file and changes the content of the label which is being watched by the sidecar container. As a result, the sidecar container sends a SIGHUP to the vRouter process which reloads the changed configuration.

Kubernetes Operator Basics

An operator consists of one or multiple controller/api pairs. The api contains the type definitions and how they are exposed to the user. The controller implements the business logic dealing with the information provided by the user. The operator SDK will auto-generate code skeletons for the api and the controller.

Custom Controller Architecture

SharedInformer

SharedInformer provides eventually consistent linkage of its clients, the Resource Event Handlers, to the authoritative state of a given collection of objects. An object is identified by its API group, kind/resource, namespace, and name. One SharedInformer provides linkage to objects of a particular API group and kind/resource. Whenever a custom controller needs to watch for events generated by a particular object type, a SharedInformer is created.

1. The Reflector watches add/update/delete events for the object type.
2. Objects which triggered an event will be placed in the Delta FIFO queue along with the event type.

3. The Informer pops objects from the queue.
4. The Indexer indexes the object using name/namespace as a key
5. and stores the object with the key in a thread safe cache.

CustomController

The CustomController implements business logic dealing with objects it is watching.

1. Resource Event Handlers are invoked by the Informer(s). The Resource Event Handlers define, based on the event type and a set of customizable functions, what key is placed on the Work Queue. There is one event handler function per event type. Resource Event Handlers can be used to filter events and modify the key.
2. Resource Event Handlers enqueue the key (not the object) on the Work Queue.
3. The Reconciler pops the key from the Work Queue.
4. The Reconciler retrieves the object using the key from the cache using the key.

Now the business logic dealing with the object is executed.

An implementation of a barebone controller (not using any SDK), watching pod and deployment events, can be seen here:

https://github.com/michaelhenkel/app-operator/blob/master/pkg/main/mycontroller.go

Operator SDK

The Operator SDK provides convenient wrappers around the client-go libraries which makes it easy to listen for certain events of certain objects. This allow s us toallows to focus on the implementation of the business logic, which basically defines how a custom controller reacts on a change of state.

Operator SDK Development Workflow

Define type data structure for the API

The initial step is to define the data structure of the resource. This data structure defines the fields and the data types represented to a user.

```
type AppASpec struct {
    Name string `json:"name,omitempty"`
    Size int    `json:"size,omitempty"`
}
type AppAStatus struct {
    Status string `json:"status,omitempty"`
}
```

This data structure defines a name and a size field as user inputs and a status field which will contain the status information.

Generate Custom Resource Definition

The operator SDK allows us to generate anthe OpenAPIv3 representation of the data structure automatically.

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
 name: appas.app.example.com
spec:
 group: app.example.com
 names:
   kind: AppA
   listKind: AppAList
   plural: appas
   singular: appa
```

```
scope: Namespaced
subresources:
  status: {}
validation:
  openAPIV3Schema:
    properties:
      apiVersion:
        description: 'APIVersion defines the versioned schema of this representation
          of an object. Servers should convert recognized schemas to the latest
          internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/contributors/devel/api-conventions.md#resources'
        type: string
      kind:
        description: 'Kind is a string value representing the REST resource this
          object represents. Servers may infer this from the endpoint the client
          submits requests to. Cannot be updated. In CamelCase. More info: https://git.k8s.io/community/contributors/devel/api-conventions.md#types-kinds'
        type: string
      metadata:
        type: object
      spec:
        properties:
          name:
            description: 'INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
              Important: Run "operator-sdk generate k8s" to regenerate code after
              modifying this file Add custom validation using kubebuilder tags:
              https://book.kubebuilder.io/beyond_basics/generating_crd.html'
            type: string
          size:
            format: int64
            type: integer
        type: object
      status:
        properties:
          status:
            description: 'INSERT ADDITIONAL STATUS FIELD - define observed state
              of cluster Important: Run "operator-sdk generate k8s" to regenerate
              code after modifying this file Add custom validation using kubebuilder
              tags: https://book.kubebuilder.io/beyond_basics/generating_crd.html'
            type: string
        type: object
version: v1alpha1
```

```
versions:
- name: v1alpha1
  served: true
  storage: true
```

This CRD (Custom Resource Definition) above can be applied to the KubeAPI server which will then expose anthe endpoint to the resource. KubeAPI will perform syntax validation based on thise definition.

Define the business logic in the controller

The business logic defines what the controller does with certain events.

Application Custom Controller Workflows

Generic Reconciliation Flow

Manager Controller

The Manager Controller is a convenience wrapper around all subsequent controllers. It allows, using a single manifest, to

- start the application controller
- create the application resource

Once the application resources are created, the Manager Controller watches for changes on the resources. The Manager Controller status provides a holistic view on the state of the entire system.

Manifest format

```
apiVersion: contrail.juniper.net/v1alpha1
kind: Manager
metadata:
 name: cluster-1
spec:
 size: 1
 hostNetwork: true
 contrailStatusImage: hub.juniper.net/contrail-nightly/contrail-status:5.2.0-0.740
 imagePullSecrets:
 - contrail-nightly
 config:
  activate: true
  create: true
  configuration:
    cloudOrchestrator: kubernetes
  images:
    api: hub.juniper.net/contrail-nightly/contrail-controller-config-api:5.2.0-0.740
    devicemanager: hub.juniper.net/contrail-nightly/contrail-controller-config-devicemgr:5.2.0-0.740
    schematransformer: hub.juniper.net/contrail-nightly/contrail-controller-config-schema:5.2.0-0.740
    servicemonitor: hub.juniper.net/contrail-nightly/contrail-controller-config-svcmonitor:5.2.0-0.740
```

```
    analyticsapi: hub.juniper.net/contrail-nightly/contrail-analytics-api:5.2.0-0.740
    collector: hub.juniper.net/contrail-nightly/contrail-analytics-collector:5.2.0-0.740
    redis: hub.juniper.net/contrail-nightly/contrail-external-redis:5.2.0-0.740
    nodemanagerconfig: hub.juniper.net/contrail-nightly/contrail-nodemgr:5.2.0-0.740
    nodemanageranalytics: hub.juniper.net/contrail-nightly/contrail-nodemgr:5.2.0-0.740
    nodeinit: hub.juniper.net/contrail-nightly/contrail-node-init:5.2.0-0.740
    init: busybox
control:
  activate: true
  create: true
  images:
    control: hub.juniper.net/contrail-nightly/contrail-controller-control-control:5.2.0-0.740
    dns: hub.juniper.net/contrail-nightly/contrail-controller-control-dns:5.2.0-0.740
    named: hub.juniper.net/contrail-nightly/contrail-controller-control-named:5.2.0-0.740
    nodemanager: hub.juniper.net/contrail-nightly/contrail-nodemgr:5.2.0-0.740
    nodeinit: hub.juniper.net/contrail-nightly/contrail-node-init:5.2.0-0.740
    init: busybox
kubemanager:
  activate: true
  create: true
  images:
    kubemanager: hub.juniper.net/contrail-nightly/contrail-kubernetes-kube-manager:5.2.0-0.740
    nodeinit: hub.juniper.net/contrail-nightly/contrail-node-init:5.2.0-0.740
    init: busybox
  configuration:
    serviceAccount: contrail-service-account
    clusterRoleBinding: contrail-cluster-role-binding
    clusterRole: contrail-cluster-role
    cloudOrchestrator: kubernetes
    #useKubeadmConfig: true
    kubernetesApiServer: "10.96.0.1"
    kubernetesApiSecurePort: 443
    kubernetesPodSubnets: 10.32.0.0/12
    kubernetesServiceSubnets: 10.96.0.0/12
    kubernetesClusterName: kubernetes
    kubernetesIpFabricForwarding: true
    kubernetesIpFabricSnat: true
    k8sTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
webui:
  activate: true
```

```
  create: true
  images:
    webuiweb: hub.juniper.net/contrail-nightly/contrail-controller-webui-web:5.2.0-0.740
    webuijob: hub.juniper.net/contrail-nightly/contrail-controller-webui-job:5.2.0-0.740
    nodeinit: hub.juniper.net/contrail-nightly/contrail-node-init:5.2.0-0.740
vrouter:
  activate: true
  create: true
  images:
    vrouteragent: hub.juniper.net/contrail-nightly/contrail-vrouter-agent:5.2.0-0.740
    vrouterkernelinit: hub.juniper.net/contrail-nightly/contrail-vrouter-kernel-init:5.2.0-0.740
    vroutercni: hub.juniper.net/contrail-nightly/contrail-kubernetes-cni-init:5.2.0-0.740
    nodemanager: hub.juniper.net/contrail-nightly/contrail-nodemgr:5.2.0-0.740
    nodeinit: hub.juniper.net/contrail-nightly/contrail-node-init:5.2.0-0.740
cassandra:
  activate: true
  create: true
  images:
    cassandra: gcr.io/google-samples/cassandra:v13
    init: busybox
  configuration:
    cassandraListenAddress: auto
    cassandraPort: 9160
    cassandraCqlPort: 9042
    cassandraSslStoragePort: 7001
    cassandraStoragePort: 7000
    cassandraJmxPort: 7199
    cassandraStartRpc: true
    cassandraClusterName: ContrailConfigDB
    maxHeapSize: 512M
    heapNewSize: 100M
    nodeType: config-database
zookeeper:
  activate: true
  create: true
  images:
    zookeeper: hub.juniper.net/contrail-nightly/contrail-external-zookeeper:5.2.0-0.740
    init: busybox
  configuration:
    zookeeperPort: 2181
```

```
    zookeeperPorts: 2888:3888
    nodeType: config-database
rabbitmq:
  activate: true
  create: true
  images:
    rabbitmq: hub.juniper.net/contrail-nightly/contrail-external-rabbitmq:5.2.0-0.740
    init: busybox
  configuration:
    erlangCookie: 47EFF3BB-4786-46E0-A5BB-58455B3C2CB4
    nodePort: 5673
    nodeType: config-database
```

Workflow diagrams

Cassandra Controller

The Cassandra Controller operates the Cassandra cluster. It allows to

- start a single node cluster
- start a multi node cluster
- scale up
- replace
- upgrade

The Cassandra Pods are deployed as a Kubernetes StatefulSet.

Workflow diagram

Configuration Crafting

The Cassandra Pods require a configuration file and each Pod's configuration file contains Pod specific and cluster wide configuration information, as such, each Pod requires its own configuration file. In a Deployment, there is no concept of a per Pod configuration. All Pods share the same configuration. In order to overcome this limitation, the configuration is written as a value in a ConfigMap. The key for the value is a Pod identifier (Pod Name or IP). Each ConfigMap has a key per Pod and the key's value is the Pod configuration.

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: cassandra-cluster1
 namespace: default
data:
 "172.16.0.1": |
    listen_address: "172.16.0.1"
    seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
      - seeds: 172.16.0.1,172.16.0.2
 "172.16.0.2": |
    listen_address: "172.16.0.2"
    seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
      - seeds: 172.16.0.1,172.16.0.2
 "172.16.0.3": |
    listen_address: "172.16.0.3"
    seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
      - seeds: 172.16.0.1,172.16.0.2
```

The ConfigMap keys are mounted as VolumeMounts inside the Pods, where the key is the filename and the value the content of the file. Each Pod will have all configuration files:

```
  ~ cat configs/172.16.0.1.yaml
listen_address: "172.16.0.1"
seed_provider:
- class_name: org.apache.cassandra.locator.SimpleSeedProvider
  parameters:
  - seeds: 172.16.0.1,172.16.0.2
  ~ cat configs/172.16.0.2.yaml
listen_address: "172.16.0.2"
seed_provider:
- class_name: org.apache.cassandra.locator.SimpleSeedProvider
  parameters:
  - seeds: 172.16.0.1,172.16.0.2
  ~ cat configs/172.16.0.3.yaml
listen_address: "172.16.0.3"
seed_provider:
- class_name: org.apache.cassandra.locator.SimpleSeedProvider
  parameters:
  - seeds: 172.16.0.1,172.16.0.2
```

Each Pod is configured with a command parameter which starts Cassandra using the Pod specific configuration file:

```
command := []string{"bash", "-c", "/docker-entrypoint.sh cassandra -f -Dcassandra.config=file:///configs/${POD_IP}.yaml"}
```

This allows the Cassandra Controller to update the configuration at runtime. I.e. in case two nodes are added to the cluster, one node should be added to the seeds list. The Controller updates the ConfigMap values and changes

- seeds: 172.16.0.1,172.16.0.2

to

- seeds: 172.16.0.1,172.16.0.2,172.16.0.2

This change goes into the configuration files and all Cassandra nodes dynamically read that change.

Readiness Probe

In order to signalize to other controllers the readiness of the Cassandra cluster, Readiness Probes are used. Readiness Probes run in a loop and execute a command inside the container and return 0 or -1. If all Pods of the cluster return 0, the cluster is considered to be ready.

```
    readinessProbe:
      exec:
        command:
        - /bin/bash
        - -c
        - "seeds=$(grep -r '  - seeds:' /mydata/${POD_IP}.yaml |awk -F'  - seeds: ' '{print $2}'|tr  ',' ' ') &&  for seed in $(echo $seeds); do if [[ $(nodetool status | grep $seed |awk '{print $1}') != 'UN' ]]; then exit -1; fi; done"
```

// Needs to be adjusted as it only checks for the seeds but not all nodes

The readiness is used by the depending services to evaluate if they are good to start or have to wait.

Node Draining

When a Cassandra node leaves the cluster, it needs to be de-registered first. When the Pod is stopped, a life-cycle hook executes a command inside the container before the Pod is terminated.

```
lifecycle:
  preStop:
    exec:
      command:
      - /bin/sh
      - -c
      - nodetool drain
```

Zookeeper Controller

The Zookeeper Controller operates the Zookeeper cluster. It allows to

- start a single node cluster
- start a multi node cluster
- scale up
- replace
- upgrade

Scale up from single node to multi node (min. 3) requires a restart of the Zookeeper cluster. Scaling up from 3 nodes onwards is seamless.

The Zookeeper pods are deployed as a Kubernetes StatefulSet.

Workflow diagram

Same as for Cassandra

Configuration Crafting

The Zookeeper cluster has to start with three nodes in order to scale further out. If started with a single node, the Deployment has to be re-created with the amount of required nodes (minimum 3). Zookeeper uses two separate configuration files, one containing the per Pod static configuration and one containing the cluster wide dynamic configuration.

Crafting the static configuration is a bit complex as Zookeeper process cannot be started with a configuration file as an argument, it only takes a directory. Mounting a ConfigMap as a volume does not allow for per Pod directory names, the directory name is the same across all Pods and has no node identifier. As a consequence the controller can only craft a configuration which is common to all Pods. However, the configuration file requires Pod specific information. This is where the container startup command comes into play:

```
command := []string{"bash", "-c", "myid=$(cat /mydata/${POD_IP}) && echo ${myid} > /data/myid && cp /conf-1/* /conf/ && sed -i \"s/clientPortAddress=.* /clientPortAddress=${POD_IP}/g\" /conf/zoo.cfg && zkServer.sh --config /conf start-foreground"}
```

It takes the generic static configuration file which was crafted by the controller, adds the Pod specific information and copies it to a separate directory. Zookeeper is started by pointing to that directory. The static configuration is mutable and cannot be changed at runtime. Changes to static configuration require a re-creation of the Pod.

Handling the dynamic configuration is straightforward as it is the same across all Pods. It is a single key/value pair in the ConfigMap which is mounted into all Pods. A change in scale changes the dynamic configuration file which is then consumed by each Zookeeper Pod.

Static configuration file (per Pod specific)

```
root@zookeeper-cluster-1-7c9467656b-hzkg2:/apache-zookeeper-3.5.5-bin# cat /conf/zoo.cfg
clientPort=2181
clientPortAddress=172.17.0.7
dataDir=/data
dataLogDir=/datalog
tickTime=2000
initLimit=5
syncLimit=2
maxClientCnxns=60
admin.enableServer=true
standaloneEnabled=false
4lw.commands.whitelist=stat,ruok,conf,isro
reconfigEnabled=true
dynamicConfigFile=/mydata/zoo.cfg.dynamic.100000000
```

Dynamic configuration file (same across all Pods)

```
root@zookeeper-cluster-1-7c9467656b-hzkg2:/apache-zookeeper-3.5.5-bin# cat /mydata/zoo.cfg.dynamic.100000000
server.1=172.17.0.6:2888:3888:participant
server.2=172.17.0.7:2888:3888:participant
server.3=172.17.0.8:2888:3888:participant
```

Readiness Probe

```
    readinessProbe:

      exec:

        command:

        - /bin/bash

        - -c

        - "OK=$(echo ruok | nc ${POD_IP} 2181); if [[ ${OK} == \"imok\" ]]; then exit 0; else exit 1;fi"
```

Rabbitmq Controller

The Rabbitmq Controller operates the Rabbitmq cluster. It allows to

- start a single node cluster
- start a multi node cluster
- scale up
- replace
- upgrade

The Rabbitmq Pods are deployed as a Kubernetes StatefulSet.

Workflow diagram

Same as for Cassandra

Configuration Crafting

The Rabbitmq configuration does not need cluster wide configuration but only per Pod configuration. A Rabbitmq cluster is bootstrapped with an initial Pod and all subsequent Pods register themselves with the initial Pod. As such there is no dynamic configuration required.

The per Pod configuration is stored in a ConfigMap where the Pod identifier is the key and the value the configuration. This configuration is mounted as a file inside the container.

Rabbitmq cannot be started using a configuration file location as an argument. The location of the configuration file is defined by environment variables. Each Pod requires its own RABBITMQ_NODENAME, as such the environment variable cannot be defined as part of the Pod definition. Therefore the export of the variable and the startup sequence are part of the startup command.

```
            runner := `#!/bin/bash
echo $RABBITMQ_ERLANG_COOKIE > /var/lib/rabbitmq/.erlang.cookie
chmod 0600 /var/lib/rabbitmq/.erlang.cookie
export RABBITMQ_NODENAME=rabbit@${POD_IP}
if [[ $(grep $POD_IP /etc/rabbitmq/0) ]] ; then
 rabbitmq-server
else
 rabbitmqctl --node rabbit@$(cat /etc/rabbitmq/0) ping
 while [[ $? -ne 0 ]]; do
   rabbitmqctl --node rabbit@$(cat /etc/rabbitmq/0) ping
 done
 rabbitmq-server -detached
 rabbitmqctl --node rabbit@$(cat /etc/rabbitmq/0) node_health_check
 while [[ $? -ne 0 ]]; do
   rabbitmqctl --node rabbit@$(cat /etc/rabbitmq/0) node_health_check
 done
 rabbitmqctl stop_app
 sleep 2
 rabbitmqctl join_cluster rabbit@$(cat /etc/rabbitmq/0)
 rabbitmqctl shutdown
 rabbitmq-server
fi
command := []string{"bash", "/runner/run.sh"}
```

Readiness Probe

The Controller creates a file in each Pod which contains the intended amount of total Pods. The Readiness Probe script compares that with the output of the rabbitmqctl cluster_status command.

```
    readinessProbe:
      exec:
        command:
        c- /bin/bash
        - -c
        - "export RABBITMQ_NODENAME=rabbit@$POD_IP; cluster_status=$(rabbitmqctl cluster_status);nodes=$(echo $cluster_status | sed -e 's/.*disc,\\[\\(.*\\)]}]}, {.*/\\1/' | grep -oP \"(?<=rabbit@).*?(?='\)\"); for node in $(cat /etc/rabbitmq/rabbitmq.nodes); do echo ${nodes} |grep ${node}; if [[ $? -ne 0 ]]; then exit -1; fi; done"
      initialDelaySeconds: 15
      timeoutSeconds: 5
```

Node Drain

When a Pod is (gracefully) stopped, the command rabbitmqctl reset is executed.

ContrailConfig Controller

The ContrailConfig Controller operates the ContrailConfig cluster. It allows to

- start a single node cluster
- start a multi node cluster
- scale up
- replace
- upgrade

It watches Zookeeper, Cassandra and Rabbitmq Controller. Any change in the dependent controller or the ContrailConfig controller itself requires a re-creation of the ContrailConfig Deployment.

The ContrailConfig Pods are deployed as a Kubernetes Deployment.

Workflow Diagram

ContrailRegistration Controller

In Contrail the Contrail Control, Analytics and vRouter nodes must be registered with the Contrail Config database (Cassandra). The registration is a REST call to the ContrailConfig service. Previously this registration was done by the respective Pods. This introduced a dependency of those Pods on ContrailConfig services. In order to break that dependency, the registration will now be performed centrally by the ContrailRegistration Controller.

The ContrailRegistration Controller listens for ContrailConfig, Control and vRouter Pod creation events. When the ContrailRegistration Controller is started initially, it creates a queue of all Pods it must register. When ContrailConfig becomes available, it will pop the Pods from the queue and register them. Likewise, when the Pods are removed, the ContrailRegistration Controller will deregister them.

ContrailKubemanager Controller

The ContrailKubemamager Controller operates the ContrailKubemamager cluster. It allows to

- start a single node cluster
- start a multi node cluster
- scale up
- replace
- upgrade

It watches Zookeeper, Cassandra, Rabbitmq and ContrailConfig Controller. Any change in the dependent controller or the ContrailKubemamager controller itself requires a restart of the ContrailKubemanager cluster.

The ContrailKubemanager Pods are deployed as a Kubernetes StatefulSet.

Workflow diagram

ContrailControl Controller

The ContrailControl Controller operates the ContrailControl cluster. It allows to

- start a single node cluster
- start a multi node cluster
- scale up
- replace
- upgrade

The ContrailControl pods are deployed as a Kubernetes StatefulSet.

Workflow diagram

Same as ContrailConfig

ContrailVrouter Controller

The ContrailVrouter Controller operates Kubernetes Daemonsets. The Contrail vRouter service consists of a forwarding and an agent component. The forwarding component can be either a kernel module, a dpdk poll mode driver (PMD) or sriov. Sriov can be combined with kernel mode or dpdk forwarding. As in Deployments, Daemonsets have no concept of Pod specific configuration. Each configuration in the Daemonset is applied to all Pods managed by that Daemonset. As vRouters run on all nodes in the cluster and their configuration will have dependency on the hardware and network configuration of a particular node (NIC types, PCI addresses, QoS configuration, vRouter gateway) a single Daemonset will not be sufficient.

Groups and Profiles

To overcome the described limitation the notion of profiles is introduced. Multiple instances of the ContrailVrouter Controller can be started. Each instance of the resource specifies a group of nodes which share the same hardware configuration parameters.

Each instance can include one or more profiles.

Nodes will be labeled with a nodeselector based on which the correct Group/Daemonset will be deployed.

Data structure

```
type Vrouter struct {
    Spec Spec
}
type Spec struct {
    Groups []*Group
    Profiles []*Profile
}
type Group struct {
    Name           string
    Profiles       []*Profile
    VrouterGateway net.IPAddr
    NodeSelector   NodeSelector
    Tolerations    []Toleration
}
type Profile struct {
    Name                   string
    DpdkConfiguration      DpdkConfiguration
    KernelModeConfiguration KernelModeConfiguration
    SriovConfiguration     SriovConfiguration
    OtherConfig            OtherConfig
}
type DpdkConfiguration struct {
    CoreMask   string
    MoreConfig map[string]string
}
type KernelModeConfiguration struct {
    MoreConfig map[string]string
}
type SriovConfiguration struct {
    NumberOfVfs            int
    VirtualFunctionMappings []string
    MoreConfig             map[string]string
}
type OtherConfig struct {
    MoreConfig map[string]string
}
```

Custom Resource

```
vrouterProfileTemplates:
 - metadata:
     name: dpdk-profile1
```

```
    labels:
      contrailcluster: cluster-1
  spec:
    dpdkConfiguration:
    coreMask: "0xF"
    2MBHugePages: 1024
    1GBHugePages: 10
    cpuPinning:
    moreConfig:
      key1: value1
      key2: value2
- metadata:
    name: sriov-profile1
    labels:
      contrailcluster: cluster-1
  spec:
    sriovConfiguration:
      numberOfVfs: 7
      virtualFunctionMappings:
      - vf1
      - vf2
    moreConfig:
      key1: value1
      key2: value2
- metadata:
    name: kernelmode-profile1
    labels:
      contrailcluster: cluster-1
  spec:
    kernelModeConfiguration:
    moreConfig:
      key1: value1
      key2: value2
vrouterTemplates:
- metadata:
    name: vrouter-dpdk-group1
    kind: ContrailVrouter
    labels:
      contrailcluster: cluster-1
  spec:
```

```
    activate: true
    nodeSelector:
      node-role.kubernetes.io/infra: ""
      nicType: x710
    tolerations:
    - operator: Exists
      effect: NoSchedule
    override: false
    upgradeStrategy: rolling
    configuration:
      vRouterGateway: 1.1.1.1
      profiles:
      - dpdk-profile1
      - other-profile1
- metadata:
    name: vrouter-sriov-group 1
    kind: ContrailVrouter
    labels:
      contrailcluster: cluster-1
  spec:
    activate: true
    nodeSelector:
      node-role.kubernetes.io/infra: ""
      nodeType: sriov
    tolerations:
    - operator: Exists
      effect: NoSchedule
    override: false
    upgradeStrategy: rolling
    configuration:
      vRouterGateway: 1.1.1.2
      profiles:
      - sriov-profile1
      - other-profile1
- metadata:
    name: vrouter-kernelmode-group1
    kind: ContrailVrouter
    labels:
      contrailcluster: cluster-1
  spec:
```

```
activate: true
nodeSelector:
  node-role.kubernetes.io/infra: ""
  nodeType: sriov
tolerations:
- operator: Exists
  effect: NoSchedule
override: false
upgradeStrategy: rolling
configuration:
  vRouterGateway: 1.1.1.3
  profiles:
  - kernelmode-profile1
  - other-profile1
```

Action Items:

- [x] Unittests
- [ ] Unittests
- [ ] Unittests
- [ ] libsandesh to support dynamic reload of collectors (https://contrail-jws.atlassian.net/browse/CEM-7473)`
- [ ] Contrail python services to support dynamic reload of cassandra/rabbitmq/zookeeper (https://contrail-jws.atlassian.net/browse/CEM-7473)
- [ ] Add controllers for fabric Pods (swift, ironic, keystone, mysql)
- [ ] Description of rolling/in-place upgrade per Controller
- [ ] Ansible playbook (...due to the lack of any real alternatives) for kubernetes deployment (we should consider KubeSpray) - tested with KubeSpray
- [ ] KubeAPI HA strategy
- [ ] reverse proxy per node?
- [ ] can it be integrated into kubeadm init phase (https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init-phase/)?
- [ ] kubespray seems to support it (https://github.com/kubernetes-sigs/kubespray/blob/master/docs/ha-mode.md)
- [ ] Contrail-status must be replaced. Status of the components must be shown in the status field of the resource
- [ ] Add DPDK/SRIOV agent roles
- [ ] Add TLS