

Sdewan CRD Controller

- [Goal](#)
- [Sdwan Design Principle](#)
- [Architecture](#)
- [CNF Deployment](#)
- [Sdewan rule CRs](#)
- [CNF Service CR](#)
- [Sdewan rule CRD Reconcile Logic](#)
- [Unusual Cases](#)
- [Admission Webhook Usage](#)
- [Sdewan rule CR type level Permission Implementation](#)
- [ServiceRule controller \(For next release\)](#)
- [References](#)

Goal

Sdewan CRD Controller (config agent) is the controller of Sdewan CRDs. With the CRD Controller, we are able to deploy Sdewan CRs to configure CNF rules. In this page, we have the following terms, let's define them here.

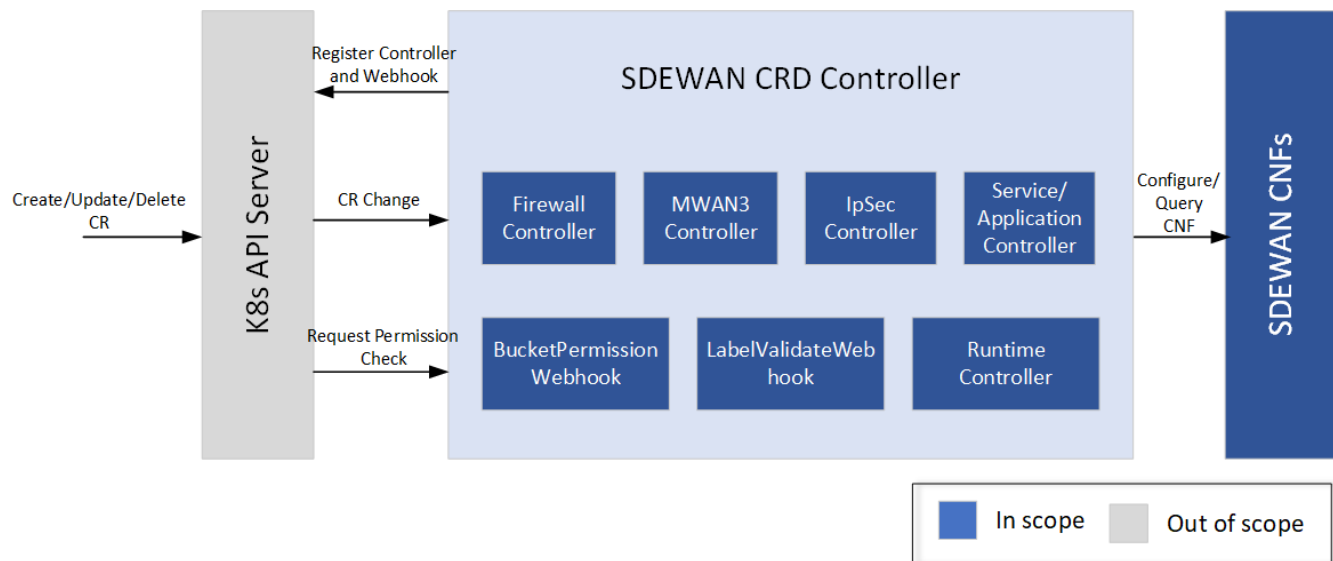
- **CNF Deployment:** A deployment running network function process(openWRT)
- **Sdewan rule:** The rule defines the CNF behaves. We have 3 classes of rules: mwan3, firewall, ipsec. Each class includes several kinds of rules. For example, mwan3 has 2 kinds: mwan3_policy and mwan3_rule. Firewall has 5 kinds: firewall_zone, firewall_snat, firewall_dnat, firewall_forwarding, firewall_rule. Ipsec has xx(ruoyu) kinds: xx, xx.
- **Sdewan rule CRD:** The CRD defines each kind of sdewan rule. For each kind of Sdewan rule, we have a Sdewan rule CRD. Sdewan rule CRD is namespaced resource.
- **Sdewan rule CR:** Instance of Sdewan rule CRD.
- **Sdewan controller:** The controller watching Sdewan rule CRs.
- **CNF:** A network function running in container.

To deploy a CNF, user needs to create one CNF deployment and some Sdewan rule CRs. In a Kubernetes namespace, there could be more than one CNF deployment and many Sdewan rule CRs. We use label to correlate one CNF with some Sdewan rule CRs. The Sdewan controller watches Sdewan rule CRs and applies them onto the correlated CNF by calling CNF REST api.

Sdwan Design Principle

- There could be multiple tenants/namespaces in a Kubernetes cluster. User may deploy multiple CNFs in any one or more tenants.
- The replica of CNF deployment could be more than one for active/backup purpose. We should apply rules for all the pods under CNF deployment. (This release doesn't implement VRRP between pods)
- CNF deployment and Sdewan rule CRs can be created/updated/deleted in any order
- The Sdewan controller and CNF process could be crash/restart at anytime for some reasons. We need to handle these scenarios
- Each Sdewan rule CR has labels to identify the type it belongs to. 3 types are available at this time: basic, app-intent and k8s-service. We extend k8s user role permission so that we can set user permission at type level of Sdewan rule CR
- Sdewan rule CR dependencies are checked on creating/updating/deleting. For example, if we create a mwan3_rule CR which uses policy policy-x, but no mwan3_policy CR named policy-x exists. Then we block the request

Architecture



SDEWAN CRD Controller internally calls SDEWAN Restful API to do CNF configuration. And a remote client (e.g. SDEWAN Overlay Controller) can manage SDEWAN CNF configuration through creating/updating/deleting SDEWAN CRs. It includes below components:

- MWAN3 Controller: monitor mwan3 related CR change then do mwan3 configuration in SDEWAN CNF
- Firewall Controller: monitor firewall related CR change then do firewall configuration in SDEWAN CNF
- IpSec Controller: monitor ipsec related CR change then do ipsec configuration in SDEWAN CNF
- Service/Application Controller: configure firewall/NAT rule for in-cluster service and application
- Runtime controller: collect runtime information of CNF include IPsec, IKE, firewall/NAT connections, DHCP leases, DNS entries, ARP entries etc..
- BucketPerssion/LabelValidateWebhook: do sdewan CR request permission check based on CR label and user

CNF Deployment

In this section we describe what the CNF deployment should be like, as well as the pod under the deployment.

- CNF pod should has multiple network interfaces attached. We use multus and ovn4nfv CNIs to enable multiple interfaces. So in the CNF pod yaml, we set annotations: `k8s.v1.cni.cncf.io/networks`, `k8s.plugin.opnfv.org/nfn-network`.
- When user deploys a CNF, she/he most likely want to deploy the CNF on a specified node instead of a random node. Because some nodes may don't have provider network connected. So we set `spec.nodeSelector` for pod
- CNF pod runs Sdewan CNF (based on openWRT in ICN). We use image `integratedcloudnative/openwrt:dev`
- CNF pod should setup with rediness probe. Sdewan controller would check pod readiness before calling CNF REST api.

CNF pod

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: cnf-1
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  replicas: 1
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      annotations:
        k8s.plugin.opnfv.org/nfn-network: |-
          { "type": "ovn4nfv", "interface": [
            {
              "defaultGateway": "false",
              "interface": "net0",
              "name": "ovn-priv-net"
            },
            {
              "defaultGateway": "false",
              "interface": "net1",
              "name": "ovn-provider-net1"
            },
            {
              "defaultGateway": "false",
              "interface": "net2",
              "name": "ovn-provider-net2"
            }
          ]
        }
        k8s.v1.cni.cncf.io/networks: ' [{ "name": "ovn-networkobj"} ]'
    spec:
      containers:
      - command:
        - /bin/sh
        - /tmp/sdewan/entrypoint.sh
        image: integratedcloudnative/openwrt:dev
        name: sdewan
        readinessProbe:
          failureThreshold: 5
          httpGet:
            path: /
            port: 80
            scheme: HTTP
          initialDelaySeconds: 5
          periodSeconds: 5
          successThreshold: 1
          timeoutSeconds: 1
        securityContext:
          privileged: true
          procMount: Default
        volumeMounts:
        - mountPath: /tmp/sdewan
          name: example-sdewan
          readOnly: true
      nodeSelector:
        kubernetes.io/hostname: ubuntu18
```

Sdewan rule CRs

CRD defines all properties of a resource, but it's not human friendly. So we paste Sdewan rule CR samples instead of CRDs.

- Each Sdewan rule CR has a label named `sdewanPurpose` to indicate which CNF should the rule be applied onto
- Each Sdewan rule CR has the `status` field which indicates if the latest rule is applied and when it's applied
- `Mwan3Policy.spec.members[].network` should match the networks defined in CNF pod annotation `k8s.plugin.opnfv.org/nfn-network`. As well as `FirewallZone.spec[].network`

CR samples of Mwan3 type:

Mwan3Policy CR

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: Mwan3Policy
metadata:
  name: balancel
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  members:
    - network: ovn-net1
      weight: 2
      metric: 2
    - network: ovn-net2
      weight: 3
      metric: 3
status:
  appliedVersion: "2"
  appliedTime: "2020-03-29T04:21:48Z"
  inSync: True
```

Mwan3Rule CR

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: Mwan3Rule
metadata:
  name: http_rule
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  policy: balancel
  src_ip: 192.168.1.2
  dest_ip: 0.0.0.0/0
  dest_port: 80
  proto: tcp
status:
  appliedVersion: "2"
  appliedTime: "2020-03-29T04:21:48Z"
  inSync: True
```

CR samples of Firewall type:

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: FirewallZone
metadata:
  name: lan1
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  newtork:
    - ovn-net1
  input: ACCEPT
  output: ACCEPT
status:
  appliedVersion: "2"
  appliedTime: "2020-03-29T04:21:48Z"
  inSync: True
```

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: FirewallRule
metadata:
  name: reject_80
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  src: lan1
  src_ip: 192.168.1.2
  src_port: 80
  proto: tcp
  target: REJECT
status:
  appliedVersion: "2"
  appliedTime: "2020-03-29T04:21:48Z"
  inSync: True
```

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: FirewallSNAT
metadata:
  name: snat_lan1
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  src: lan1
  src_ip: 192.168.1.2
  src_dip: 1.2.3.4
  dest: wan1
  proto: icmp
status:
  appliedVersion: "2"
  appliedTime: "2020-03-29T04:21:48Z"
  inSync: True
```

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: FirewallDNAT
metadata:
  name: dnat_wan1
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  src: wan1
  src_dport: 19900
  dest: lan1
  dest_ip: 192.168.1.1
  dest_port: 22
  proto: tcp
status:
  appliedVersion: "2"
  appliedTime: "2020-03-29T04:21:48Z"
  inSync: True
```

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: FirewallForwarding
metadata:
  name: forwarding_lan_to_wan
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  src: lan1
  dest: wan1
status:
  appliedVersion: "2"
  appliedTime: "2020-03-29T04:21:48Z"
  inSync: True
```

CR samples of IPSec type(ruoyu):

IPSec Proposal CR

```
apiVersion: sdewan.akraino.org/v1alpha1
kind: IpsecProposal
metadata:
  name: test_proposal_1
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  encryption_algorithm: aes128
  hash_algorithm: sha256
  dh_group: modp3072
status:
  appliedVersion: "1"
  appliedTime: "2020-04-12T09:28:38Z"
  inSync: True
```

IPSec Site CR

```
apiVersion: sdewan.akraino.org/v1alpha1
kind: IpsecSite
metadata:
  name: ipsecsite-sample
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  remote: xx.xx.xx.xx
  authentication_method: psk
  pre_shared_key: xxx
  local_public_cert:
  local_private_cert:
  shared_ca:
  local_identifier:
  remote_identifier:
  crypto_proposal:
    - test_proposal_1
  connections:
    - connection_name: connection_A
      type: tunnel
      mode: start
      local_subnet: 172.12.0.0/24, 10.239.160.22
      remote_sourceip: 172.12.0.30-172.12.0.45
      remote_subnet:
      crypto_proposal:
        - test_proposal_1
status:
  appliedVersion: "1"
  appliedTime: "2020-04-12T09:28:38Z"
  inSync: True
```

IPSec Host CR

```
apiVersion: sdewan.akraino.org/v1alpha1
kind: IpsecHost
metadata:
  name: ipsechost-sample
  namespace: default
  labels:
    sdewanPurpose: cnf-1
spec:
  remote: xx.xx.xx.xx/%any
  authentication_method: psk
  pre_shared_key: xxx
  local_public_cert:
  local_private_cert:
  shared_ca:
  local_identifier:
  remote_identifier:
  crypto_proposal:
    - test_proposal_1
  connections:
    - connection_name: connection_A
      type: tunnel
      mode: start
      local_sourceip: %config
      remote_sourceip: xx.xx.xx.xx
      remote_subnet: xx.xx.xx.xx/xx
      crypto_proposal:
        - test_proposal_1
status:
  appliedVersion: "1"
  appliedTime: "2020-04-12T09:28:38Z"
  inSync: True
```

CNF Service CR

.spec.fullname - The full name of the target service, with which we can get the service IP

.spec.port - The port exposed by CNF, we will do DNAT for the requests accessing this port of CNF

.spec.dport - The port exposed by target service

CNF Service CR

```
apiVersion: batch.sdewan.akraino.org/v1alpha1
kind: CNFService
metadata:
  name: cnfservice-sample
  namespace: default
  labels:
    sdewanPurpose: cnf1
spec:
  fullname: httpd-svc.default.svc.cluster.local
  port: "2288"
  dport: "8080"
```

Sdewan rule CRD Reconcile Logic

As we have many kinds of CRDs, they have almost the same reconcile logic. So we only describe the Mwan3Rule logic.

Mwan3Rule Reconcile could be triggered by the following cases:

- Create/Update/Delete Mwan3Rule CR
- CNF deployment ready status change (With [predicate feature](#), we can only watch CNF deployment readiness status. With [enqueueRequestsFrom MapFunc](#), we can enqueue all Mwan3Rule CRs with specified `labels.sdewanPurpose`, if CNF deployment's ready status changes)
 - CNF becomes ready after creating
 - CNF becomes ready after restart
 - CNF becomes not-ready after crash

Mwan3Rule Reconcile flow:

```
def Mwan3RuleReconciler.Reconcile(req ctrl.Request):
    rule_cr = k8sClient.get(req.NamespacedName)
    cnf_deployment = k8sClient.get_deployment_with_label(rule_cr.labels.sdewanPurpose)
    if rule_cr.DeletionTimestamp.exists():
        # The CR is being deleted. finalizer on the CR
        if cnf_deployment.exists():
            if cnf_deployment.is_ready():
                for cnf_pod in cnf_deployment:
                    err = openwrt_client.delete_rule(cnf_pod_ip, rule_cr)
                    if err:
                        return "re-queue req"
                rule_cr.finalizer = nil
                return "ok"
            else:
                return "re-queue req"
        else:
            # Just remove finalizer, because no CNF pod exists
            rule_cr.finalizer = nil
            return "ok"
    else:
        # The CR is not being deleted
        if cnf_deployment not exist:
            return "ok"
        else:
            if cnf_deployment not ready:
                # set appliedVersion = nil if cnf_deployment get into not_ready status
                rule_cr.status.appliedVersion = nil
                return "re-queue req"
            else:
                for cnf_pod in cnf_deployment:
                    runtime_cr = openwrt_client.get_rule(cnf_pod_ip)
                    if runtime_cr != rule_cr:
                        err = openwrt_client.add_or_update_rule(cnf_pod_ip, rule_cr)
                        if err:
                            # err could be caused by dependencies not-applied or other reason
                            return "re-queue req"
                # set appliedVersion only when it's applied for all the cnf pods
                rule_cr.finalizer = cnf_finalizer
                rule_cr.status.appliedVersion = rule_cr.resourceVersion
                rule_cr.status.inSync = True
                return "ok"
```

Unusual Cases

In the following cases, when we say "call CNF api to create/update/delete rule", it means the logic below:

```
def create_or_update_rule(rule):
    runtime_rule = openwrt_client.get_rule(rule.name)
    if runtime_rule exist:
        if runtime_rule equal rule:
            return
        else:
            openwrt_client.update_rule(rule)
    else:
        openwrt_client.add_rule(rule)

def delete_rule(rule):
    runtime_rule = openwrt_client.get_rule(rule.name)
    if runtime_rule exist:
        openwrt_client.del_rule(rule)
```

Case 1:

- A deployment(CNF) for a given purpose has two pod replicas (CNF-pod-1 and CNF-pod-2)
- Controller is also brought yup.
- CNF-pod-1 and CNF-pod-2 are both running with no/default configuration.
- MWAN3 policy 1 is added
- MWAN3 rule 1 and Rule 2 are added to use MWAN3 Policy1.
- Since all controller, CNF-pod-1 and CNF-pod-2 are running, CNF-pod-1 and CNF-pod-2 has configuration MWAN3 Policy1, rule1 and rule2.
- Now CNF-pod-1 is stopped.

Mwan3Policy controller and Mwan3Rule controller receives a CNF event. Mwan3Policy adds all the related mwan3Policy CRs to reconcile queue. Mwan3Rule adds all the related mwan3Rule CRs to reconcile queue. In the reconcile, it finds that the CNF is not ready, so CR status.appliedVersion is set nil. The CRs are re-queued with time delay.


- MWAN3 rule 1 is deleted.

As every CR has finalizer, rule 1 CR is not deleted from etcd directly. Instead, deleteTimestamp field is added to the rule 1 CR. The mwan3Rule controller receives an event. In the reconcile, controller detects the CNF is not ready, so it re-queues the CR with delay.

- MWAN3 rule 3 added

Mwan3Rule controller receives an event. In the reconcile, controller detects the CNF is not ready, so it re-queues the CR with delay.

- MWAN3 rule 2 is updated.

 Mwan3Rule controller receives an event. In the reconcile, controller detects the CNF is not ready, so it re-queues the CR with delay.

- CNF-pod-1 is brought back up after 10 minutes (more than 5 minutes)

As pod restart, CNF-pod-1 is running with no/default configuration. In Mwan3Rule reconcile queue, there are 3 CRs: rule1, rule2, rule3. The controller reconcile them, and do the right things. For rule1, controller calls cnf api to delete rule1 from both CNF-pod-1 and CNF-pod-2. Then controller removes finalizer from the rule1 CR, then rule1 CR is deleted from etcd by k8s. For rule2, controller calls cnf api to update rule2 for both CNF-pod-1 and CNF-pod-2. Then set rule2 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true. For rule3, controller calls cnf api to add rule3 for both CNF-pod-1 and CNF-pod-2. Then set rule3 finalizer. Also set rule3 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true.


- Ensure that both CNF-pod-1 and CNF-pod-2 have latest configuration.

 Once the reconcile finish, both CNF-pod-1 and CNF-pod-2 have latest configuration.


Case 2:

- A deployment(CNF) for a given purpose has two pod replicas (CNF-pod-1 and CNF-pod-2)


- Controller is also brought up.
- CNF-pod-1 and CNF-pod-2 are both running with no/default configuration.
- MWAN3 policy 1 is added
- MWAN3 rule 1 and Rule 2 are added to use MWAN3 Policy1.
- Since all controller, CNF-pod-1 and CNF-pod-2 are running, CNF-pod-1 and CNF-pod-2 has configuration MWAN3 Policy1, rule1 and rule2.
- Now CNF-pod-1 is disconnected, but still running.

 We have the API readiness check for CNF pod, when it is disconnected. The CNF-pod-1 becomes not-ready. Mwan3Policy controller and Mwan3Rule controller receives a CNF event. Mwan3Policy adds all the related mwan3Policy CRs to reconcile queue. Mwan3Rule adds all the related mwan3Rule CRs to reconcile queue. In the reconcile, it finds that the CNF is not ready, so CR status.appliedVersion is set nil. The CRs are re-queued with time delay.

- MWAN3 rule 1 is deleted.

 As every CR has finalizer, rule 1 CR is not deleted from etcd directly. Instead, deleteTimestamp field is added to the rule 1 CR. The mwan3Rule controller receives an event. In the reconcile, controller detects the CNF is not ready, so it re-queues the CR with delay.


- MWAN3 rule 3 added

 Mwan3Rule controller receives an event. In the reconcile, controller detects the CNF is not ready, so it re-queues the CR with delay.

- MWAN3 rule 2 is updated.

 Mwan3Rule controller receives an event. In the reconcile, controller detects the CNF is not ready, so it re-queues the CR with delay.

- CNF-pod-1 is brought back up after 10 minutes (more than 5 minutes)


 As pod restart, CNF-pod-1 is running with no/default configuration. In Mwan3Rule reconcile queue, there are 3 CRs: rule1, rule2, rule3. The controller reconcile them, and do the right things. For rule1, controller calls cnf api to delete rule1 from both CNF-pod-1 and CNF-pod-2. Then controller removes finalizer from the rule1 CR, then rule1 CR is deleted from etcd by k8s. For rule2, controller calls cnf api to update rule2 for both CNF-pod-1 and CNF-pod-2. Then set rule2 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true. For rule3, controller calls cnf api to add rule3 for both CNF-pod-1 and CNF-pod-2. Then set rule3 finalizer. Also set rule3 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true.

- Ensure that both CNF-pod-1 and CNF-pod-2 have latest configuration.


 Once the reconcile finish, both CNF-pod-1 and CNF-pod-2 have latest configuration.

Case 3:


- A deployment(CNF) for a given purpose has two pod replicas (CNF-pod-1 and CNF-pod-2)
- Controller is also brought up.
- CNF-pod-1 and CNF-pod-2 are both running with no/default configuration.
- MWAN3 policy 1 is added
- MWAN3 rule 1 and Rule 2 are added to use MWAN3 Policy1.
- Since all controller, CNF-pod-1 and CNF-pod-2 are running, CNF-pod-1 and CNF-pod-2 has configuration MWAN3 Policy1, rule1 and rule2.
- Controller is down for 10 minutes.
- MWAN3 rule 1 is deleted.

 As controller is down, so no event, no reconcile. rule1 CR is not deleted from etcd because of finalizer. Instead, DeleteTimestamp is added to rule1 CR by k8s


- MWAN3 rule 3 added

 As controller is down, no event no reconcile. rule3 CR is added to etcd, but not applied onto CNF. rule3 status.appliedVersion and status.appliedTime and status.inSync are nil/default value.

- MWAN3 rule 2 is updated.

 As controller is down, no event no reconcile. rule2 CR is updated to etcd, but not applied onto CNF. rule2 status.appliedVersion and status.appliedTime and status.inSync are the value before controller goes down.

- Controller is up.


 Controller reconciles for all CRs. For rule1 CR, controller calls cnf api to delete rule1 from both CNF-pod-1 and CNF-pod-2. Then controller removes finalizer from the rule1 CR, then rule1 CR is deleted from etcd by k8s. For rule2, controller calls cnf api to update rule2 for both CNF-pod-1 and CNF-pod-2. Then set rule2 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true. For rule3, controller calls cnf api to add rule3 for both CNF-pod-1 and CNF-pod-2. Then set rule3 finalizer. Also set rule3 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true.

- Ensure that CNF-pod-1 and CNF-pod-2 have latest configuration and there is no duplicate information.


 Once the reconcile finish, both CNF-pod-1 and CNF-pod-2 have latest configuration.

Case 4:


- A deployment(CNF) for a given purpose has two pod replicas (CNF-pod-1 and CNF-pod-2)
- Controller is also brought up.
- CNF-pod-1 and CNF-pod-2 are both running with no/default configuration.
- MWAN3 policy 1 is added
- MWAN3 rule 1 and Rule 2 are added to use MWAN3 Policy1.
- Since all controller, CNF-pod-1 and CNF-pod-2 are running, CNF-pod-1 and CNF-pod-2 has configuration MWAN3 Policy1, rule1 and rule2.
- Controller is down for 10 minutes.
- After controller goes down, CNF-pod-1 is down

 As controller is down, so no event, no reconcile.

- MWAN3 rule 1 is deleted.

 As controller is down, so no event, no reconcile. rule1 CR is not deleted from etcd because of finalizer. Instead, DeleteTimestamp is added to rule1 CR by k8s


- MWAN3 rule 3 added

 As controller is down, no event no reconcile. rule3 CR is added to etcd, but not applied onto CNF. rule3 status.appliedVersion and status.appliedTime and status.inSync are nil/default value.

- For MWAN3 rule 2, we don't make any change
- CNF-pod-1 is up

 As controller is down, so no event, no reconcile. As pod restart, CNF-pod-1 is running with no/default configuration.

- Controller is up.

 Controller reconciles for all CRs. For rule1 CR, controller calls cnf api to delete rule1 from both CNF-pod-1 and CNF-pod-2. Then controller removes finalizer from the rule1 CR, then rule1 CR is deleted from etcd by k8s. For rule2, controller calls cnf api to update rule2 for both CNF-pod-1 and CNF-pod-2. Then set rule2 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true. For rule3, controller calls cnf api to add rule3 for both CNF-pod-1 and CNF-pod-2. Then set rule3 finalizer. Also set rule3 status.appliedVersion=<current-version> and status.appliedTime=<now-time> and status.inSync=true.

- Ensure that CNF-pod-1 and CNF-pod-2 have latest configuration and there is no duplicate information.

 Once the reconcile finish, both CNF-pod-1 and CNF-pod-2 have latest configuration.

Admission Webhook Usage

We use admission webhook to implement several features.

1. Prevent creating more than one CNF of the same label and the same namespace
2. Validate CR dependencies. For example, mwan3 rule depends on mwan3 policy
3. Extend user permission to control the operations on rule CRs. For example, we can control that ONAP can't update/delete rule CRs created by platform.

Sdewan rule CR type level Permission Implementation

k8s support permission control on namespace level. For example, user1 may be able to create/update/delete one kind of resource(e.g. pod) in namespace ns1, but not namespace ns2. For Sdewan, this can't fit our requirement. We want label level control of Sdewan rule CRs. For example, user_onap can create/update/delete Mwan3Rule CR of label sdewan-bucket-type=app-intent, but not label sdewan-bucket-type=basic.

Let me first describe the extended permission system and then explain how we implement it. In k8s, user or serviceAccount could be bonded to one or more roles. The roles defines the permissions, for example the following role defines that sdewan-test role can create/update Mwan3Rule CRs in default namespace. Also sdewan-testrole can get Mwan3Policy CRs.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  annotations:
    name: sdewan-test
    namespace: default
rules:
- apiGroups:
  - ""
  resources:
  - mwan3rules
  verbs:
  - create
  - update
- apiGroups:
  - ""
  resources:
  - mwan3policies
  verbs:
  - get
```

We extend the Role with annotations. In the annotation, we can define labeled based permissions. For example, the following role extends sdewan-test role permission: sdewan-test can only create/update Mwan3Rule CRs with label sdewan-bucket-type=app-intent or sdewan-bucket-type=k8s-service. Also it can only get Mwan3Policy CR with label sdewan-bucket-type=app-intent.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  annotations:
    sdewan-bucket-type-permission: |-
      { "mwan3rules": ["app-intent", "k8s-service"],
        "mwan3policies": ["app-intent"] }
    name: sdewan-test
    namespace: default
rules:
- apiGroups:
  - ""
  resources:
  - mwan3rules
  verbs:
  - create
  - update
- apiGroups:
  - ""
  resources:
  - mwan3policies
  verbs:
  - get
```

We use admission webhook to implement the type level permission control. Let me describe how admission webhook in simple words. When k8s api receives a request, kube-api call webhook API before save the object into etcd. If the webhook returns allowed=true, kube-api continues to persistent the object into etcd. Otherwise, kube-api reject the request. The webhook can optional tell kube-api to update the object together with allowed=true returned. Webhook request body has a field named [userInfo](#), it indicates who is making the k8s api request. With this field, we can implement the extended permission in webhook.

```
def mwan3rule_webhook_handle_permission(req admission.Request):
    userinfo = req["userInfo"]
    mwan3rule_cr = decode(req)
    roles = k8s_client.get_role_from_user(userinfo)
    for role in roles:
        if mwan3rule_cr.labels.sdewan-bucket-type in role.annotation.sdewan-bucket-type-permission.mwan3rules:
            return {"allowd": True}
    return {"allowd": False}
```

ServiceRule controller (For next release)

We create a controller watches the services created in the cluster. For each service, it creates a FirewallIDNAT CR. On controller startup, it makes a syncup to remove unused CRs.

References

- <https://github.com/kubernetes-sigs/controller-runtime/blob/master/pkg/doc.go>
- <https://book.kubebuilder.io/reference/using-finalizers.html>
- <https://godoc.org/sigs.k8s.io/controller-runtime/pkg/predicate#example-Funcs>
- <https://godoc.org/sigs.k8s.io/controller-runtime/pkg/handler#example-EnqueueRequestsFromMapFunc>