Kubernetes Default ServiceCIDR Reconfiguration

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

This document shares how to reconfigure the default Service IP range(s) assigned to a cluster.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.33.

To check the version, enter `kubectl version`.

Kubernetes Default ServiceCIDR Reconfiguration

This document explains how to manage the Service IP address range within a Kubernetes cluster, which also influences the cluster's supported IP families for Services.

The IP families available for Service ClusterIPs are determined by the `--service-cluster-ip-range` flag to kube-apiserver. For a better understanding of Service IP address allocation, refer to the Services IP address allocation tracking documentation.

Since Kubernetes 1.33, the Service IP families configured for the cluster are reflected by the ServiceCIDR object named `kubernetes`. The `kubernetes` ServiceCIDR object is created by the first kube-apiserver instance that starts, based on its configured `--service-cluster-ip-range` flag. To ensure consistent cluster behavior, all kube-apiserver instances must be configured with the same `--service-cluster-ip-range` values, which must match the default `kubernetes` ServiceCIDR object.

Kubernetes ServiceCIDR Reconfiguration Categories

ServiceCIDR reconfiguration typically falls into one of the following categories:

- **Extending the existing ServiceCIDRs:** This can be done dynamically by adding new ServiceCIDR objects without the need for reconfiguring the kube-apiserver. Please refer to the dedicated documentation on Extending Service IP Ranges.

- **Single-to-dual-stack conversion preserving the primary ServiceCIDR:** This involves introducing a secondary IP family (IPv6 to an IPv4-only cluster, or IPv4 to an IPv6-only cluster) while keeping the original IP family as primary. This requires an update to the kube-apiserver configuration and a corresponding modification of various cluster components that need to handle this additional IP family. These components include, but are not limited to, kube-proxy, the CNI or network plugin, service mesh implementations, and DNS services.

- **Dual-to-single conversion preserving the primary ServiceCIDR:** This involves removing the secondary IP family from a dual-stack cluster, reverting to a single IP family while retaining the original primary IP family. In addition to reconfiguring the components to match the new IP family, you might need to address Services that were explicitly configured to use the removed IP family.

- **Anything that results in changing the primary ServiceCIDR:** Completely replacing the default ServiceCIDR is a complex operation. If the new ServiceCIDR does not overlap with the existing one, it will require renumbering all existing Services and changing the `kubernetes.default` Service. The case where the primary IP family also changes is even more complicated, and may require changing multiple cluster components (kubelet, network plugins, etc.) to match the new primary IP family.

Manual Operations for Replacing the Default ServiceCIDR

Reconfiguring the default ServiceCIDR necessitates manual steps performed by the cluster operator, administrator, or the software managing the cluster lifecycle. These typically include:

1. **Updating** the kube-apiserver configuration: Modify the `--service-cluster-ip-range` flag with the new IP range(s).
2. **Reconfiguring** the network components: This is a critical step and the specific procedure depends on the different networking components in use. It might involve updating configuration files, restarting agent pods, or updating the components to manage the new ServiceCIDR(s) and the desired IP family configuration for Pods. Typical components can be the implementation of Kubernetes Services, such as kube-proxy, and the configured networking plugin, and potentially other networking components like service mesh controllers and DNS servers, to ensure they can correctly handle traffic and perform service discovery with the new IP family configuration.
3. **Managing existing Services:** Services with IPs from the old CIDR need to be addressed if they are not within the new configured ranges. Options include recreation (leading to downtime and new IP assignments) or potentially more complex reconfiguration strategies.
4. **Recreating internal Kubernetes services:** The `kubernetes.default` Service must be deleted and recreated to obtain an IP address from the new ServiceCIDR if the primary IP family is changed or replaced by a different network.

Illustrative Reconfiguration Steps

The following steps describe a controlled reconfiguration focusing on the complete replacement of the default ServiceCIDR and the recreation of the `kubernetes.default` Service:

1. Start the kube-apiserver with the initial `--service-cluster-ip-range`.

2. Create initial Services that obtain IPs from this range.
3. Introduce a new ServiceCIDR as a temporary target for reconfiguration.
4. Mark the `kubernetes` default ServiceCIDR for deletion (it will remain pending due to existing IPs and finalizers). This prevents new allocations from the old range.
5. Recreate existing Services. They should now be allocated IPs from the new, temporary ServiceCIDR.
6. Restart the kube-apiserver with the new ServiceCIDR(s) configured and shut down the old instance.
7. Delete the `kubernetes.default` Service. The new kube-apiserver will recreate it within the new ServiceCIDR.

What's next

- Kubernetes Networking Concepts
- Kubernetes Dual-Stack Services
- Extending Kubernetes Service IP Ranges

Extend Service IP Ranges

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

This document shares how to extend the existing Service IP range assigned to a cluster.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.29.

To check the version, enter `kubectl version`.

Note:

While you can use this feature with an earlier version, the feature is only GA and officially supported since v1.33.

Extend Service IP Ranges

Kubernetes clusters with kube-apiservers that have enabled the `MultiCIDRServiceAllocator` feature gate and have the `networking.k8s.io/v1` API group active, will create a ServiceCIDR object that takes the well-known name `kubernetes`, and that specifies an IP address range based on the value of the `--service-cluster-ip-range` command line argument to kube-apiserver.

```
kubectl get servicecidr
```

```
NAME          CIDRS           AGE
kubernetes    10.96.0.0/28    17d
```

The well-known `kubernetes` Service, that exposes the kube-apiserver endpoint to the Pods, calculates the first IP address from the default ServiceCIDR range and uses that IP address as its cluster IP address.

```
kubectl get service kubernetes
```

```
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
kubernetes    ClusterIP   10.96.0.1     <none>        443/TCP   17d
```

The default Service, in this case, uses the ClusterIP 10.96.0.1, that has the corresponding IPAddress object.

```
kubectl get ipaddress 10.96.0.1
```

```
NAME         PARENTREF
10.96.0.1    services/default/kubernetes
```

The ServiceCIDRs are protected with finalizers, to avoid leaving Service ClusterIPs orphans; the finalizer is only removed if there is another subnet that contains the existing IPAddresses or there are no IPAddresses belonging to the subnet.

Extend the number of available IPs for Services

There are cases that users will need to increase the number addresses available to Services, previously, increasing the Service range was a disruptive operation that could also cause data loss. With this new feature users only need to add a new ServiceCIDR to increase the number of available addresses.

Adding a new ServiceCIDR

On a cluster with a 10.96.0.0/28 range for Services, there is only 2^(32-28) - 2 = 14 IP addresses available. The `kubernetes.default` Service is always created; for this example, that leaves you with only 13 possible Services.

```
for i in $(seq 1 13); do kubectl create service clusterip "test-$i" --tcp 80 -o json | jq -r .spec.clusterIP; done
```

```
10.96.0.11
10.96.0.5
10.96.0.12
10.96.0.13
10.96.0.14
10.96.0.2
10.96.0.3
10.96.0.4
10.96.0.6
10.96.0.7
10.96.0.8
10.96.0.9
error: failed to create ClusterIP service: Internal error occurred: failed to allocate a serviceIP: range is full
```

You can increase the number of IP addresses available for Services, by creating a new ServiceCIDR that extends or adds new IP address ranges.

```
cat <EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: ServiceCIDR
metadata:
  name: newcidr1
spec:
  cidrs:
  - 10.96.0.0/24
EOF
```

```
servicecidr.networking.k8s.io/newcidr1 created
```

and this will allow you to create new Services with ClusterIPs that will be picked from this new range.

```
for i in $(seq 13 16); do kubectl create service clusterip "test-$i" --tcp 80 -o json | jq -r .spec.clusterIP; done
```

```
10.96.0.48
10.96.0.200
10.96.0.121
10.96.0.144
```

Deleting a ServiceCIDR

You cannot delete a ServiceCIDR if there are IPAddresses that depend on the ServiceCIDR.

```
kubectl delete servicecidr newcidr1
```

```
servicecidr.networking.k8s.io "newcidr1" deleted
```

Kubernetes uses a finalizer on the ServiceCIDR to track this dependent relationship.

```
kubectl get servicecidr newcidr1 -o yaml
```

```
apiVersion: networking.k8s.io/v1
kind: ServiceCIDRmetadata:  creationTimestamp: "2023-10-12T15:11:07Z"  deletionGracePeriodSeconds: 0  deletionTimestamp: "2023-10-
```

By removing the Services containing the IP addresses that are blocking the deletion of the ServiceCIDR

```
for i in $(seq 13 16); do kubectl delete service "test-$i" ; done
```

```
service "test-13" deleted
service "test-14" deleted
service "test-15" deleted
service "test-16" deleted
```

the control plane notices the removal. The control plane then removes its finalizer, so that the ServiceCIDR that was pending deletion will actually be removed.

```
kubectl get servicecidr newcidr1
```

```
Error from server (NotFound): servicecidrs.networking.k8s.io "newcidr1" not found
```

Kubernetes Service CIDR Policies

Cluster administrators can implement policies to control the creation and modification of ServiceCIDR resources within the cluster. This allows for centralized management of the IP address ranges used for Services and helps prevent unintended or conflicting configurations. Kubernetes provides mechanisms like Validating Admission Policies to enforce these rules.

Preventing Unauthorized ServiceCIDR Creation/Update using Validating Admission Policy

There can be situations that the cluster administrators want to restrict the ranges that can be allowed or to completely deny any changes to the cluster Service IP ranges.

Note:

The default "kubernetes" ServiceCIDR is created by the kube-apiserver to provide consistency in the cluster and is required for the cluster to work, so it always must be allowed. You can ensure your `ValidatingAdmissionPolicy` doesn't restrict the default ServiceCIDR by adding the clause:

```
  matchConditions:
  - name: 'exclude-default-servicecidr'
    expression: "object.metadata.name != 'kubernetes'"
```

as in the examples below.

Restrict Service CIDR ranges to some specific ranges

The following is an example of a `ValidatingAdmissionPolicy` that only allows ServiceCIDRs to be created if they are subranges of the given `allowed` ranges. (So the example policy would allow a ServiceCIDR with `cidrs: ['10.96.1.0/24']` or `cidrs: ['2001:db8:0:0:ffff::/80', '10.96.0.0/20']` but would not allow a ServiceCIDR with `cidrs: ['172.20.0.0/16']`.) You can copy this policy and change the value of `allowed` to something appropriate for you cluster.

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "servicecidrs.default"spec:  failurePolicy: Fail  matchConstraints:    resourceRul
```

Consult the CEL documentation to learn more about CEL if you want to write your own validation `expression`.

Restrict any usage of the ServiceCIDR API

The following example demonstrates how to use a `ValidatingAdmissionPolicy` and its binding to restrict the creation of any new Service CIDR ranges, excluding the default "kubernetes" ServiceCIDR:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "servicecidrs.deny"spec:  failurePolicy: Fail  matchConstraints:    resourceRules:
```

Adding entries to Pod /etc/hosts with HostAliases

Adding entries to a Pod's `/etc/hosts` file provides Pod-level override of hostname resolution when DNS and other options are not applicable. You can add these custom entries with the HostAliases field in PodSpec.

The Kubernetes project recommends modifying DNS configuration using the `hostAliases` field (part of the `.spec` for a Pod), and not by using an init container or other means to edit `/etc/hosts` directly. Change made in other ways may be overwritten by the kubelet during Pod creation or restart.

Default hosts file content

Start an Nginx Pod which is assigned a Pod IP:

```
kubectl run nginx --image nginx
```

```
pod/nginx created
```

Examine a Pod IP:

```
kubectl get pods --output=wide
```

```
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4   worker0
```

The hosts file content would look like this:

```
kubectl exec nginx -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.
127.0.0.1	localhost
::1	localhost ip6-localhost ip6-loopback
fe00::0	ip6-localnet
fe00::0	ip6-mcastprefix
fe00::1	ip6-allnodes
fe00::2	ip6-allrouters
10.200.0.4	nginx
```

By default, the `hosts` file only includes IPv4 and IPv6 boilerplates like `localhost` and its own hostname.

Adding additional entries with hostAliases

In addition to the default boilerplate, you can add additional entries to the `hosts` file. For example: to resolve `foo.local`, `bar.local` to `127.0.0.1` and `foo.remote`, `bar.remote` to `10.1.2.3`, you can configure HostAliases for a Pod under `.spec.hostAliases`:

`service/networking/hostaliases-pod.yaml` Copy service/networking/hostaliases-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: hostaliases-podspec:  restartPolicy: Never  hostAliases:  - ip: "127.0.0.1"    hostnames:    - "foo.loca
```

You can start a Pod with that configuration by running:

```
kubectl apply -f https://k8s.io/examples/service/networking/hostaliases-pod.yaml
```

```
pod/hostaliases-pod created
```

Examine a Pod's details to see its IPv4 address and its status:

```
kubectl get pod --output=wide
```

```
NAME                           READY     STATUS      RESTARTS   AGE       IP              NODE
hostaliases-pod                0/1       Completed   0          6s        10.200.0.5      worker0
```

The `hosts` file content looks like this:

```
kubectl logs hostaliases-pod
```

```
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.200.0.5      hostaliases-pod

# Entries added by HostAliases.
127.0.0.1       foo.local       bar.local
10.1.2.3        foo.remote      bar.remote
```

with the additional entries specified at the bottom.

Why does the kubelet manage the hosts file?

The kubelet manages the `hosts` file for each container of the Pod to prevent the container runtime from modifying the file after the containers have already been started. Historically, Kubernetes always used Docker Engine as its container runtime, and Docker Engine would then modify the `/etc/hosts` file after each container had started.

Current Kubernetes can use a variety of container runtimes; even so, the kubelet manages the hosts file within each container so that the outcome is as intended regardless of which container runtime you use.

Caution:

Avoid making manual changes to the hosts file inside a container.

If you make manual changes to the hosts file, those changes are lost when the container exits.

Networking

Learn how to configure networking for your cluster.

Adding entries to Pod /etc/hosts with HostAliases

Extend Service IP Ranges

Kubernetes Default ServiceCIDR Reconfiguration

Validate IPv4/IPv6 dual-stack

Validate IPv4/IPv6 dual-stack

This document shares how to validate IPv4/IPv6 dual-stack enabled Kubernetes clusters.

Before you begin

- Provider support for dual-stack networking (Cloud provider or otherwise must be able to provide Kubernetes nodes with routable IPv4/IPv6 network interfaces)
- A network plugin that supports dual-stack networking.
- Dual-stack enabled cluster

Your Kubernetes server must be at or later than version v1.23.

To check the version, enter `kubectl version`.

Note:

While you can validate with an earlier version, the feature is only GA and officially supported since v1.23.

Validate addressing

Validate node addressing

Each dual-stack Node should have a single IPv4 block and a single IPv6 block allocated. Validate that IPv4/IPv6 Pod address ranges are configured by running the following command. Replace the sample node name with a valid dual-stack Node from your cluster. In this example, the Node's name is `k8s-linuxpool1-34450317-0`:

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template='{{range .spec.podCIDRs}}{{printf "%s\n" .}}{{end}}'
```

```
10.244.1.0/24
2001:db8::/64
```

There should be one IPv4 block and one IPv6 block allocated.

Validate that the node has an IPv4 and IPv6 interface detected. Replace node name with a valid node from the cluster. In this example the node name is `k8s-linuxpool1-34450317-0`:

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template='{{range .status.addresses}}{{printf "%s: %s\n" .type .addre
```

```
Hostname: k8s-linuxpool1-34450317-0
InternalIP: 10.0.0.5
InternalIP: 2001:db8:10::5
```

Validate Pod addressing

Validate that a Pod has an IPv4 and IPv6 address assigned. Replace the Pod name with a valid Pod in your cluster. In this example the Pod name is `pod01`:

```
kubectl get pods pod01 -o go-template --template='{{range .status.podIPs}}{{printf "%s\n" .ip}}{{end}}'
```

```
10.244.1.4
2001:db8::4
```

You can also validate Pod IPs using the Downward API via the `status.podIPs` fieldPath. The following snippet demonstrates how you can expose the Pod IPs via an environment variable called `MY_POD_IPS` within a container.

```
        env:
        - name: MY_POD_IPS
          valueFrom:
            fieldRef:
              fieldPath: status.podIPs
```

The following command prints the value of the `MY_POD_IPS` environment variable from within a container. The value is a comma separated list that corresponds to the Pod's IPv4 and IPv6 addresses.

```
kubectl exec -it pod01 -- set | grep MY_POD_IPS
```

```
MY_POD_IPS=10.244.1.4,2001:db8::4
```

The Pod's IP addresses will also be written to `/etc/hosts` within a container. The following command executes a cat on `/etc/hosts` on a dual stack Pod. From the output you can verify both the IPv4 and IPv6 IP address for the Pod.

```
kubectl exec -it pod01 -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
fe00::0    ip6-mcastprefix
fe00::1    ip6-allnodes
fe00::2    ip6-allrouters
10.244.1.4    pod01
2001:db8::4    pod01
```

Validate Services

Create the following Service that does not explicitly define `.spec.ipFamilyPolicy`. Kubernetes will assign a cluster IP for the Service from the first configured `service-cluster-ip-range` and set the `.spec.ipFamilyPolicy` to `SingleStack`.

service/networking/dual-stack-default-svc.yaml Copy service/networking/dual-stack-default-svc.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  selector:    app.kubernetes.io/name: MyAp
```

Use `kubectl` to view the YAML for the Service.

```
kubectl get svc my-service -o yaml
```

The Service has `.spec.ipFamilyPolicy` set to `SingleStack` and `.spec.clusterIP` set to an IPv4 address from the first configured range set via `--service-cluster-ip-range` flag on kube-controller-manager.

```
apiVersion: v1
kind: Servicemetadata:  name: my-service  namespace: defaultspec:  clusterIP: 10.0.217.164  clusterIPs:  - 10.0.217.164  ipFamilie
```

Create the following Service that explicitly defines `IPv6` as the first array element in `.spec.ipFamilies`. Kubernetes will assign a cluster IP for the Service from the IPv6 range configured `service-cluster-ip-range` and set the `.spec.ipFamilyPolicy` to `SingleStack`.

service/networking/dual-stack-ipfamilies-ipv6.yaml Copy service/networking/dual-stack-ipfamilies-ipv6.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  ipFamilies:  - IPv6  selector:    app.kub
```

Use `kubectl` to view the YAML for the Service.

```
kubectl get svc my-service -o yaml
```

The Service has `.spec.ipFamilyPolicy` set to `SingleStack` and `.spec.clusterIP` set to an IPv6 address from the IPv6 range set via `--service-cluster-ip-range` flag on kube-controller-manager.

```
apiVersion: v1
kind: Servicemetadata:  labels:    app.kubernetes.io/name: MyApp  name: my-servicespec:  clusterIP: 2001:db8:fd00::5118  clusterIP
```

Create the following Service that explicitly defines `PreferDualStack` in `.spec.ipFamilyPolicy`. Kubernetes will assign both IPv4 and IPv6 addresses (as this cluster has dual-stack enabled) and select the `.spec.ClusterIP` from the list of `.spec.ClusterIPs` based on the address family of the first element in the `.spec.ipFamilies` array.

service/networking/dual-stack-preferred-svc.yaml Copy service/networking/dual-stack-preferred-svc.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  ipFamilyPolicy: PreferDualStack  selector
```

Note:

The `kubectl get svc` command will only show the primary IP in the `CLUSTER-IP` field.

```
kubectl get svc -l app.kubernetes.io/name=MyApp
```

```
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
my-service   ClusterIP   10.0.216.242   <none>        80/TCP    5s
```

Validate that the Service gets cluster IPs from the IPv4 and IPv6 address blocks using `kubectl describe`. You may then validate access to the service via the IPs and ports.

```
kubectl describe svc -l app.kubernetes.io/name=MyApp
```

```
Name:              my-service
Namespace:         default
Labels:            app.kubernetes.io/name=MyApp
Annotations:       <none>
Selector:          app.kubernetes.io/name=MyApp
Type:              ClusterIP
IP Family Policy:  PreferDualStack
IP Families:       IPv4,IPv6
IP:                10.0.216.242
IPs:               10.0.216.242,2001:db8:fd00::af55
Port:              <unset>  80/TCP
TargetPort:        9376/TCP
Endpoints:         <none>
Session Affinity:  None
Events:            <none>
```

Create a dual-stack load balanced Service

If the cloud provider supports the provisioning of IPv6 enabled external load balancers, create the following Service with `PreferDualStack` in `.spec.ipFamilyPolicy`, `IPv6` as the first element of the `.spec.ipFamilies` array and the `type` field set to `LoadBalancer`.

service/networking/dual-stack-prefer-ipv6-lb-svc.yaml Copy service/networking/dual-stack-prefer-ipv6-lb-svc.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  ipFamilyPolicy: PreferDualStack  ipFamili
```

Check the Service:

```
kubectl get svc -l app.kubernetes.io/name=MyApp
```

Validate that the Service receives a `CLUSTER-IP` address from the IPv6 address block along with an `EXTERNAL-IP`. You may then validate access to the service via the IP and port.

```
NAME         TYPE           CLUSTER-IP            EXTERNAL-IP        PORT(S)        AGE
my-service   LoadBalancer   2001:db8:fd00::7ebc   2603:1030:805::5   80:30790/TCP   35s
```