Services, Load Balancing, and Networking

Concepts and resources behind networking in Kubernetes.

- 1: Service
- 2: Ingress
- 3: Ingress Controllers
- 4: Gateway API
- 5: EndpointSlices
- 6: Network Policies
- 7: DNS for Services and Pods
- 8: IPv4/IPv6 dual-stack
- 9: Topology Aware Routing
- 10: Networking on Windows
- 11: Service ClusterIP allocation
- 12: Service Internal Traffic Policy

The Kubernetes network model

Every `Pod` in a cluster gets its own unique cluster-wide IP address. This means you do not need to explicitly create links between `Pods` and you almost never need to deal with mapping container ports to host ports.
This creates a clean, backwards-compatible model where `Pods` can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- pods can communicate with all other pods on any other node without NAT
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Note: For those platforms that support `Pods` running in the host network (e.g. Linux), when pods are attached to the host network of a node they can still communicate with all pods on all nodes without NAT.

This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

Kubernetes IP addresses exist at the `Pod` scope - containers within a `Pod` share their network namespaces - including their IP address and MAC address. This means that containers within a `Pod` can all reach each other's ports on `localhost`. This also means that containers within a `Pod` must coordinate port usage, but this is no different from processes in a VM. This is called the "IP-per-pod" model.
How this is implemented is a detail of the particular container runtime in use.

It is possible to request ports on the `Node` itself which forward to your `Pod` (called host ports), but this is a very niche operation. How that forwarding is implemented is also a detail of the container runtime. The `Pod` itself is blind to the existence or non-existence of host ports.

Kubernetes networking addresses four concerns:

- Containers within a Pod use networking to communicate via loopback.
- Cluster networking provides communication between different Pods.
- The Service API lets you expose an application running in Pods to be reachable from outside your cluster.
  - Ingress provides extra functionality specifically for exposing HTTP applications, websites and APIs.
  - Gateway API is an add-on that provides an expressive, extensible, and role-oriented family of API kinds for modeling service networking.
- You can also use Services to publish services only for consumption inside your cluster.

The Connecting Applications with Services tutorial lets you learn about Services and Kubernetes networking with a hands-on example.

Cluster Networking explains how to set up networking for your cluster, and also provides an overview of the technologies involved.

1 - Service

Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.

In Kubernetes, a Service is a method for exposing a network application that is running as one or more Pods in your cluster.

A key aim of Services in Kubernetes is that you don't need to modify your existing application to use an unfamiliar service discovery mechanism. You can run code in Pods, whether this is a code designed for a cloud-native world, or an older app you've containerized. You use a Service to make that set of Pods available on the network so that clients can interact with it.

If you use a Deployment to run your app, that Deployment can create and destroy Pods dynamically. From one moment to the next, you don't know how many of those Pods are working and healthy; you might not even know what those healthy Pods are named. Kubernetes Pods are created and destroyed to match the desired state of your cluster. Pods are ephemeral resources (you should not expect that an individual Pod is reliable and durable).

Each Pod gets its own IP address (Kubernetes expects network plugins to ensure this). For a given Deployment in your cluster, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Enter *Services*.

Services in Kubernetes

The Service API, part of Kubernetes, is an abstraction to help you expose groups of Pods over a network. Each Service object defines a logical set of endpoints (usually these endpoints are Pods) along with a policy about how to make those pods accessible.

For example, consider a stateless image-processing backend which is running with 3 replicas. Those replicas are fungible—frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves.

The Service abstraction enables this decoupling.

The set of Pods targeted by a Service is usually determined by a selector that you define. To learn about other ways to define Service endpoints, see Services *without* selectors.

If your workload speaks HTTP, you might choose to use an Ingress to control how web traffic reaches that workload. Ingress is not a Service type, but it acts as the entry point for your cluster. An Ingress lets you consolidate your routing rules into a single resource, so that you can expose multiple components of your workload, running separately in your cluster, behind a single listener.

The Gateway API for Kubernetes provides extra capabilities beyond Ingress and Service. You can add Gateway to your cluster - it is a family of extension APIs, implemented using CustomResourceDefinitions - and then use these to configure access to network services that are running in your cluster.

Cloud-native service discovery

If you're able to use Kubernetes APIs for service discovery in your application, you can query the API server for matching EndpointSlices. Kubernetes updates the EndpointSlices for a Service whenever the set of Pods in a Service changes.

For non-native applications, Kubernetes offers ways to place a network port or load balancer in between your application and the backend Pods.

Either way, your workload can use these service discovery mechanisms to find the target it wants to connect to.

Defining a Service

A Service is an object (the same way that a Pod or a ConfigMap is an object). You can create, view or modify Service definitions using the Kubernetes API. Usually you use a tool such as `kubectl` to make those API calls for you.

For example, suppose you have a set of Pods that each listen on TCP port 9376 and are labelled as `app.kubernetes.io/name=MyApp`. You can define a Service to publish that TCP listener:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Applying this manifest creates a new Service named "my-service" with the default ClusterIP service type. The Service targets TCP port 9376 on any Pod with the `app.kubernetes.io/name: MyApp` label.

Kubernetes assigns this Service an IP address (the *cluster IP*), that is used by the virtual IP address mechanism. For more details on that mechanism, read Virtual IPs and Service Proxies.

The controller for that Service continuously scans for Pods that match its selector, and then makes any necessary updates to the set of EndpointSlices for the Service.

The name of a Service object must be a valid RFC 1035 label name.

> **Note:** A Service can map *any* incoming `port` to a `targetPort`. By default and for convenience, the `targetPort` is set to the same value as the `port` field.

Port definitions

Port definitions in Pods have names, and you can reference these names in the `targetPort` attribute of a Service. For example, we can bind the `targetPort` of the Service to the Pod port in the following way:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/name: proxy
spec:
  containers:
  - name: nginx
    image: nginx:stable
    ports:
      - containerPort: 80
        name: http-web-svc

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app.kubernetes.io/name: proxy
  ports:
  - name: name-of-service-port
    protocol: TCP
    port: 80
    targetPort: http-web-svc
```

This works even if there is a mixture of Pods in the Service using a single configured name, with the same network protocol available via different port numbers. This offers a lot of flexibility for deploying and evolving your Services. For example, you can change the port numbers that Pods expose in the next version of your backend software, without breaking clients.

The default protocol for Services is TCP; you can also use any other supported protocol.

Because many Services need to expose more than one port, Kubernetes supports multiple port definitions for a single Service. Each port definition can have the same `protocol`, or a different one.

Services without selectors

Services most commonly abstract access to Kubernetes Pods thanks to the selector, but when used with a corresponding set of EndpointSlices objects and without a selector, the Service can abstract other kinds of backends, including ones that run outside the cluster.

For example:

- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different Namespace or on another cluster.
- You are migrating a workload to Kubernetes. While evaluating the approach, you run only a portion of your backends in Kubernetes.

In any of these scenarios you can define a Service *without* specifying a selector to match Pods. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Because this Service has no selector, the corresponding EndpointSlice (and legacy Endpoints) objects are not created automatically. You can map the Service to the network address and port where it's running, by adding an EndpointSlice object manually. For example:

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-1 # by convention, use the name of the Service
                     # as a prefix for the name of the EndpointSlice
  labels:
    # You should set the "kubernetes.io/service-name" label.
    # Set its value to match the name of the Service
    kubernetes.io/service-name: my-service
addressType: IPv4
ports:
  - name: '' # empty because port 9376 is not assigned as a well-known
             # port (by IANA)
    appProtocol: http
    protocol: TCP
    port: 9376
endpoints:
  - addresses:
      - "10.4.5.6"
  - addresses:
      - "10.1.2.3"
```

Custom EndpointSlices

When you create an EndpointSlice object for a Service, you can use any name for the EndpointSlice. Each EndpointSlice in a namespace must have a unique name. You link an EndpointSlice to a Service by setting the `kubernetes.io/service-name` label on that EndpointSlice.

> **Note:**
> The endpoint IPs *must not* be: loopback (127.0.0.0/8 for IPv4, ::1/128 for IPv6), or link-local (169.254.0.0/16 and 224.0.0.0/24 for IPv4, fe80::/64 for IPv6).
>
> The endpoint IP addresses cannot be the cluster IPs of other Kubernetes Services, because kube-proxy doesn't support virtual IPs as a destination.

For an EndpointSlice that you create yourself, or in your own code, you should also pick a value to use for the label `endpointslice.kubernetes.io/managed-by`. If you create your own controller code to manage EndpointSlices, consider using a value similar to `"my-domain.example/name-of-controller"`. If you are using a third party tool, use the name of the tool in all-lowercase and change spaces and other punctuation to dashes (`-`). If people are directly using a tool such as `kubectl` to manage EndpointSlices, use a name that describes this manual management, such as `"staff"` or `"cluster-admins"`. You should avoid using the reserved value `"controller"`, which identifies EndpointSlices managed by Kubernetes' own control plane.

Accessing a Service without a selector

Accessing a Service without a selector works the same as if it had a selector. In the example for a Service without a selector, traffic is routed to one of the two endpoints defined in the EndpointSlice manifest: a TCP connection to 10.1.2.3 or 10.4.5.6, on port 9376.

> **Note:** The Kubernetes API server does not allow proxying to endpoints that are not mapped to pods. Actions such as `kubectl proxy <service-name>` where the service has no selector will fail due to this constraint. This prevents the Kubernetes API server from being used as a proxy to endpoints the caller may not be authorized to access.

An `ExternalName` Service is a special case of Service that does not have selectors and uses DNS names instead. For more information, see the ExternalName section.

EndpointSlices

FEATURE STATE: `Kubernetes v1.21 [stable]`

EndpointSlices are objects that represent a subset (a *slice*) of the backing network endpoints for a Service.

Your Kubernetes cluster tracks how many endpoints each EndpointSlice represents. If there are so many endpoints for a Service that a threshold is reached, then Kubernetes adds another empty EndpointSlice and stores new endpoint information there. By default, Kubernetes makes a new EndpointSlice once the existing EndpointSlices all contain at least 100 endpoints. Kubernetes does not make the new EndpointSlice until an extra endpoint needs to be added.

See EndpointSlices for more information about this API.

Endpoints

In the Kubernetes API, an Endpoints (the resource kind is plural) defines a list of network endpoints, typically referenced by a Service to define which Pods the traffic can be sent to.

The EndpointSlice API is the recommended replacement for Endpoints.

Over-capacity endpoints

Kubernetes limits the number of endpoints that can fit in a single Endpoints object. When there are over 1000 backing endpoints for a Service, Kubernetes truncates the data in the Endpoints object. Because a Service can be linked with more than one EndpointSlice, the 1000 backing endpoint limit only affects the legacy Endpoints API.

In that case, Kubernetes selects at most 1000 possible backend endpoints to store into the Endpoints object, and sets an annotation on the Endpoints: `endpoints.kubernetes.io/over-capacity: truncated`. The control plane also removes that annotation if the number of backend Pods drops below 1000.

Traffic is still sent to backends, but any load balancing mechanism that relies on the legacy Endpoints API only sends traffic to at most 1000 of the available backing endpoints.

The same API limit means that you cannot manually update an Endpoints to have more than 1000 endpoints.

Application protocol

FEATURE STATE: `Kubernetes v1.20 [stable]`

The `appProtocol` field provides a way to specify an application protocol for each Service port. This is used as a hint for implementations to offer richer behavior for protocols that they understand. The value of this field is mirrored by the corresponding Endpoints and EndpointSlice objects.

This field follows standard Kubernetes label syntax. Valid values are one of:

- IANA standard service names.
- Implementation-defined prefixed names such as `mycompany.com/my-custom-protocol`.
- Kubernetes-defined prefixed names:

Protocol	Description
`kubernetes.io/h2c`	HTTP/2 over cleartext as described in RFC 7540

Multi-port Services

For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object. When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9377
```

Note:
As with Kubernetes names in general, names for ports must only contain lowercase alphanumeric characters and `-`. Port names must also start and end with an alphanumeric character.

For example, the names `123-abc` and `web` are valid, but `123_abc` and `-web` are not.

Service type

For some parts of your application (for example, frontends) you may want to expose a Service onto an external IP address, one that's accessible from outside of your cluster.

Kubernetes Service types allow you to specify what kind of Service you want.

The available `type` values and their behaviors are:

`ClusterIP`

Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default that is used if you don't explicitly specify a `type` for a Service. You can expose the Service to the public internet using an Ingress or a Gateway.

`NodePort`

Exposes the Service on each Node's IP at a static port (the `NodePort`). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of `type: ClusterIP`.

`LoadBalancer`

Exposes the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider.

`ExternalName`

Maps the Service to the contents of the `externalName` field (for example, to the hostname `api.foo.bar.example`). The mapping configures your cluster's DNS server to return a `CNAME` record with that external hostname value. No proxying of any kind is set up.

The `type` field in the Service API is designed as nested functionality - each level adds to the previous. However there is an exception to this nested design. You can define a `LoadBalancer` Service by disabling the load balancer `NodePort` allocation.

`type: ClusterIP`

This default Service type assigns an IP address from a pool of IP addresses that your cluster has reserved for that purpose.

Several of the other types for Service build on the `ClusterIP` type as a foundation.

If you define a Service that has the `.spec.clusterIP` set to `"None"` then Kubernetes does not assign an IP address. See headless Services for more information.

Choosing your own IP address

You can specify your own cluster IP address as part of a `Service` creation request. To do this, set the `.spec.clusterIP` field. For example, if you already have an existing DNS entry that you wish to reuse, or legacy systems that are configured for a specific IP address and difficult to re-configure.

The IP address that you choose must be a valid IPv4 or IPv6 address from within the `service-cluster-ip-range` CIDR range that is configured for the API server. If you try to create a Service with an invalid `clusterIP` address value, the API server will return a 422 HTTP status code to indicate that there's a problem.

Read avoiding collisions to learn how Kubernetes helps reduce the risk and impact of two different Services both trying to use the same IP address.

`type: NodePort`

If you set the `type` field to `NodePort`, the Kubernetes control plane allocates a port from a range specified by `--service-node-port-range` flag (default: 30000-32767). Each node proxies that port (the same port number on every Node) into your Service. Your Service reports the allocated port in its `.spec.ports[*].nodePort` field.

Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes' IP addresses directly.

For a node port Service, Kubernetes additionally allocates a port (TCP, UDP or SCTP to match the protocol of the Service). Every node in the cluster configures itself to listen on that assigned port and to forward traffic to one of the ready endpoints associated with that Service. You'll be able to contact the `type: NodePort` Service, from outside the cluster, by connecting to any node using the appropriate protocol (for example: TCP), and the appropriate port (as assigned to that Service).

Choosing your own port

If you want a specific port number, you can specify a value in the `nodePort` field. The control plane will either allocate you that port or report that the API transaction failed. This means that you need to take care of possible port collisions yourself. You also have to use a valid port number, one that's inside the range configured for NodePort use.

Here is an example manifest for a Service of `type: NodePort` that specifies a NodePort value (30007, in this example):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - port: 80
      # By default and for convenience, the `targetPort` is set to
      # the same value as the `port` field.
      targetPort: 80
      # Optional field
      # By default and for convenience, the Kubernetes control plane
      # will allocate a port from a range (default: 30000-32767)
      nodePort: 30007
```

Reserve Nodeport ranges to avoid collisions

FEATURE STATE: `Kubernetes v1.29 [stable]`

The policy for assigning ports to NodePort services applies to both the auto-assignment and the manual assignment scenarios. When a user wants to create a NodePort service that uses a specific port, the target port may conflict with another port that has already been assigned.

To avoid this problem, the port range for NodePort services is divided into two bands. Dynamic port assignment uses the upper band by default, and it may use the lower band once the upper band has been exhausted. Users can then allocate from the lower band with a lower risk of port collision.

Custom IP address configuration for `type: NodePort` Services

You can set up nodes in your cluster to use a particular IP address for serving node port services. You might want to do this if each node is connected to multiple networks (for example: one network for application traffic, and another network for traffic between nodes and the control plane).

If you want to specify particular IP address(es) to proxy the port, you can set the `--nodeport-addresses` flag for kube-proxy or the equivalent `nodePortAddresses` field of the kube-proxy configuration file to particular IP block(s).

This flag takes a comma-delimited list of IP blocks (e.g. `10.0.0.0/8`, `192.0.2.0/25`) to specify IP address ranges that kube-proxy should consider as local to this node.

For example, if you start kube-proxy with the `--nodeport-addresses=127.0.0.0/8` flag, kube-proxy only selects the loopback interface for NodePort Services. The default for `--nodeport-addresses` is an empty list. This means that kube-proxy should consider all available network interfaces for NodePort. (That's also compatible with earlier Kubernetes releases.)

> **Note:** This Service is visible as `<NodeIP>:spec.ports[*].nodePort` and `.spec.clusterIP:spec.ports[*].port`. If the `--nodeport-addresses` flag for kube-proxy or the equivalent field in the kube-proxy configuration file is set, `<NodeIP>` would be a filtered node IP address (or possibly IP addresses).

`type: LoadBalancer`

On cloud providers which support external load balancers, setting the `type` field to `LoadBalancer` provisions a load balancer for your Service. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's `.status.loadBalancer` field. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 192.0.2.127
```

Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.

To implement a Service of `type: LoadBalancer`, Kubernetes typically starts off by making the changes that are equivalent to you requesting a Service of `type: NodePort`. The cloud-controller-manager component then configures the external load balancer to forward traffic to that assigned node port.

You can configure a load balanced Service to omit assigning a node port, provided that the cloud provider implementation supports this.

Some cloud providers allow you to specify the `loadBalancerIP`. In those cases, the load-balancer is created with the user-specified `loadBalancerIP`. If the `loadBalancerIP` field is not specified, the load balancer is set up with an ephemeral IP address. If you specify a `loadBalancerIP` but your cloud provider does not support the feature, the `loadbalancerIP` field that you set is ignored.

> **Note:**
> The `.spec.loadBalancerIP` field for a Service was deprecated in Kubernetes v1.24.
>
> This field was under-specified and its meaning varies across implementations. It also cannot support dual-stack networking. This field may be removed in a future API version.
>
> If you're integrating with a provider that supports specifying the load balancer IP address(es) for a Service via a (provider specific) annotation, you should switch to doing that.
>
> If you are writing code for a load balancer integration with Kubernetes, avoid using this field. You can integrate with Gateway rather than Service, or you can define your own (provider specific) annotations on the Service that specify the equivalent detail.

Load balancers with mixed protocol types

FEATURE STATE: `Kubernetes v1.26 [stable]`

By default, for LoadBalancer type of Services, when there is more than one port defined, all ports must have the same protocol, and the protocol must be one which is supported by the cloud provider.

The feature gate `MixedProtocolLBService` (enabled by default for the kube-apiserver as of v1.24) allows the use of different protocols for LoadBalancer type of Services, when there is more than one port defined.

> **Note:** The set of protocols that can be used for load balanced Services is defined by your cloud provider; they may impose restrictions beyond what the Kubernetes API enforces.

Disabling load balancer NodePort allocation

FEATURE STATE: `Kubernetes v1.24 [stable]`

You can optionally disable node port allocation for a Service of `type: LoadBalancer`, by setting the field `spec.allocateLoadBalancerNodePorts` to `false`. This should only be used for load balancer implementations that route traffic directly to pods as opposed to using node ports. By default, `spec.allocateLoadBalancerNodePorts` is `true` and type LoadBalancer Services will

continue to allocate node ports. If `spec.allocateLoadBalancerNodePorts` is set to `false` on an existing Service with allocated node ports, those node ports will **not** be de-allocated automatically. You must explicitly remove the `nodePorts` entry in every Service port to de-allocate those node ports.

Specifying class of load balancer implementation

FEATURE STATE: `Kubernetes v1.24 [stable]`

For a Service with `type` set to `LoadBalancer`, the `.spec.loadBalancerClass` field enables you to use a load balancer implementation other than the cloud provider default.

By default, `.spec.loadBalancerClass` is not set and a `LoadBalancer` type of Service uses the cloud provider's default load balancer implementation if the cluster is configured with a cloud provider using the `--cloud-provider` component flag.

If you specify `.spec.loadBalancerClass`, it is assumed that a load balancer implementation that matches the specified class is watching for Services. Any default load balancer implementation (for example, the one provided by the cloud provider) will ignore Services that have this field set. `spec.loadBalancerClass` can be set on a Service of type `LoadBalancer` only. Once set, it cannot be changed. The value of `spec.loadBalancerClass` must be a label-style identifier, with an optional prefix such as "`internal-vip`" or "`example.com/internal-vip`". Unprefixed names are reserved for end-users.

Specifying IPMode of load balancer status

FEATURE STATE: `Kubernetes v1.29 [alpha]`

Starting as Alpha in Kubernetes 1.29, a feature gate named `LoadBalancerIPMode` allows you to set the `.status.loadBalancer.ingress.ipMode` for a Service with `type` set to `LoadBalancer`. The `.status.loadBalancer.ingress.ipMode` specifies how the load-balancer IP behaves. It may be specified only when the `.status.loadBalancer.ingress.ip` field is also specified.

There are two possible values for `.status.loadBalancer.ingress.ipMode`: "VIP" and "Proxy". The default value is "VIP" meaning that traffic is delivered to the node with the destination set to the load-balancer's IP and port. There are two cases when setting this to "Proxy", depending on how the load-balancer from the cloud provider delivers the traffics:

- If the traffic is delivered to the node then DNATed to the pod, the destination would be set to the node's IP and node port;
- If the traffic is delivered directly to the pod, the destination would be set to the pod's IP and port.

Service implementations may use this information to adjust traffic routing.

Internal load balancer

In a mixed environment it is sometimes necessary to route traffic from Services inside the same (virtual) network address block.

In a split-horizon DNS environment you would need two Services to be able to route both external and internal traffic to your endpoints.

To set an internal load balancer, add one of the following annotations to your Service depending on the cloud service provider you're using:

Default | GCP | AWS | Azure | IBM Cloud | OpenStack | Baidu Cloud | Tencent Cloud | Alibaba Cloud | OCI

Select one of the tabs.

`type: ExternalName`

Services of type ExternalName map a Service to a DNS name, not to a typical selector such as `my-service` or `cassandra`. You specify these Services with the `spec.externalName` parameter.

This Service definition, for example, maps the `my-service` Service in the `prod` namespace to `my.database.example.com`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

Note:
A Service of `type: ExternalName` accepts an IPv4 address string, but treats that string as a DNS name comprised of digits, not as an IP address (the internet does not however allow such names in DNS). Services with external names that resemble IPv4 addresses are not resolved by DNS servers.

If you want to map a Service directly to a specific IP address, consider using headless Services.

When looking up the host `my-service.prod.svc.cluster.local`, the cluster DNS Service returns a `CNAME` record with the value `my.database.example.com`. Accessing `my-service` works in the same way as other Services but with the crucial difference that redirection happens at the DNS level rather than via proxying or forwarding. Should you later decide to move your database into your cluster, you can start its Pods, add appropriate selectors or endpoints, and change the Service's `type`.

Caution:
You may have trouble using ExternalName for some common protocols, including HTTP and HTTPS. If you use ExternalName then the hostname used by clients inside your cluster is different from the name that the ExternalName references.

For protocols that use hostnames this difference may lead to errors or unexpected responses. HTTP requests will have a `Host:` header that the origin server does not recognize; TLS servers will not be able to provide a certificate matching the hostname that the client connected to.

Headless Services

Sometimes you don't need load-balancing and a single Service IP. In this case, you can create what are termed *headless Services*, by explicitly specifying `"None"` for the cluster IP address (`.spec.clusterIP`).

You can use a headless Service to interface with other service discovery mechanisms, without being tied to Kubernetes' implementation.

For headless Services, a cluster IP is not allocated, kube-proxy does not handle these Services, and there is no load balancing or proxying done by the platform for them. How DNS is automatically configured depends on whether the Service has selectors defined:

With selectors

For headless Services that define selectors, the endpoints controller creates EndpointSlices in the Kubernetes API, and modifies the DNS configuration to return A or AAAA records (IPv4 or IPv6 addresses) that point directly to the Pods backing the Service.

Without selectors

For headless Services that do not define selectors, the control plane does not create EndpointSlice objects. However, the DNS system looks for and configures either:

- DNS CNAME records for `type: ExternalName` Services.
- DNS A / AAAA records for all IP addresses of the Service's ready endpoints, for all Service types other than `ExternalName`.
  - For IPv4 endpoints, the DNS system creates A records.
  - For IPv6 endpoints, the DNS system creates AAAA records.

When you define a headless Service without a selector, the `port` must match the `targetPort`.

Discovering services

For clients running inside your cluster, Kubernetes supports two primary modes of finding a Service: environment variables and DNS.

Environment variables

When a Pod is run on a Node, the kubelet adds a set of environment variables for each active Service. It adds `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` variables, where the Service name is upper-cased and dashes are converted to underscores.

For example, the Service `redis-primary` which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11, produces the following environment variables:

```
REDIS_PRIMARY_SERVICE_HOST=10.0.0.11
REDIS_PRIMARY_SERVICE_PORT=6379
REDIS_PRIMARY_PORT=tcp://10.0.0.11:6379
REDIS_PRIMARY_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_PRIMARY_PORT_6379_TCP_PROTO=tcp
REDIS_PRIMARY_PORT_6379_TCP_PORT=6379
REDIS_PRIMARY_PORT_6379_TCP_ADDR=10.0.0.11
```

> **Note:**
> When you have a Pod that needs to access a Service, and you are using the environment variable method to publish the port and cluster IP to the client Pods, you must create the Service *before* the client Pods come into existence. Otherwise, those client Pods won't have their environment variables populated.
>
> If you only use DNS to discover the cluster IP for a Service, you don't need to worry about this ordering issue.

Kubernetes also supports and provides variables that are compatible with Docker Engine's "*legacy container links*" feature. You can read `makeLinkVariables` to see how this is implemented in Kubernetes.

DNS

You can (and almost always should) set up a DNS service for your Kubernetes cluster using an add-on.

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

For example, if you have a Service called `my-service` in a Kubernetes namespace `my-ns`, the control plane and the DNS Service acting together create a DNS record for `my-service.my-ns`. Pods in the `my-ns` namespace should be able to find the service by doing a name lookup for `my-service` (`my-service.my-ns` would also work).

Pods in other namespaces must qualify the name as `my-service.my-ns`. These names will resolve to the cluster IP assigned for the Service.

Kubernetes also supports DNS SRV (Service) records for named ports. If the `my-service.my-ns` Service has a port named `http` with the protocol set to `TCP`, you can do a DNS SRV query for `_http._tcp.my-service.my-ns` to discover the port number for `http`, as well as the IP address.

The Kubernetes DNS server is the only way to access `ExternalName` Services. You can find more information about `ExternalName` resolution in DNS for Services and Pods.

Virtual IP addressing mechanism

Read Virtual IPs and Service Proxies explains the mechanism Kubernetes provides to expose a Service with a virtual IP address.

Traffic policies

You can set the `.spec.internalTrafficPolicy` and `.spec.externalTrafficPolicy` fields to control how Kubernetes routes traffic to healthy (“ready”) backends.

See Traffic Policies for more details.

Session stickiness

If you want to make sure that connections from a particular client are passed to the same Pod each time, you can configure session affinity based on the client's IP address. Read session affinity to learn more.

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes Services can be exposed on those `externalIPs`. When network traffic arrives into the cluster, with the external IP (as destination IP) and the port matching that Service, rules and routes that Kubernetes has configured ensure that the traffic is routed to one of the endpoints for that Service.

When you define a Service, you can specify `externalIPs` for any service type. In the example below, the Service named `"my-service"` can be accessed by clients using TCP, on `"198.51.100.32:80"` (calculated from `.spec.externalIPs[]` and `.spec.ports[].port`).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 49152
  externalIPs:
    - 198.51.100.32
```

> **Note:** Kubernetes does not manage allocation of `externalIPs`; these are the responsibility of the cluster administrator.

API Object

Service is a top-level resource in the Kubernetes REST API. You can find more details about the Service API object.

What's next

Learn more about Services and how they fit into Kubernetes:

- Follow the Connecting Applications with Services tutorial.
- Read about Ingress, which exposes HTTP and HTTPS routes from outside the cluster to Services within your cluster.
- Read about Gateway, an extension to Kubernetes that provides more flexibility than Ingress.

For more context, read the following:

- Virtual IPs and Service Proxies
- EndpointSlices
- Service API reference
- EndpointSlice API reference
- Endpoint API reference (legacy)

2 - Ingress

Make your HTTP (or HTTPS) network service available using a protocol-aware configuration mechanism, that understands web concepts like URIs, hostnames, paths, and more. The Ingress concept lets you map traffic to different backends based on rules you define via the Kubernetes API.

FEATURE STATE: `Kubernetes v1.19 [stable]`

An API object that manages external access to the services in a cluster, typically HTTP.

Ingress may provide load balancing, SSL termination and name-based virtual hosting.

> **Note:** Ingress is frozen. New features are being added to the Gateway API.

Terminology

For clarity, this guide defines the following terms:

- Node: A worker machine in Kubernetes, part of a cluster.
- Cluster: A set of Nodes that run containerized applications managed by Kubernetes. For this example, and in most common Kubernetes deployments, nodes in the cluster are not part of the public internet.
- Edge router: A router that enforces the firewall policy for your cluster. This could be a gateway managed by a cloud provider or a physical piece of hardware.
- Cluster network: A set of links, logical or physical, that facilitate communication within a cluster according to the Kubernetes networking model.
- Service: A Kubernetes Service that identifies a set of Pods using label selectors. Unless mentioned otherwise, Services are assumed to have virtual IPs only routable within the cluster network.

What is Ingress?

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

Here is a simple example where an Ingress sends all its traffic to one Service:

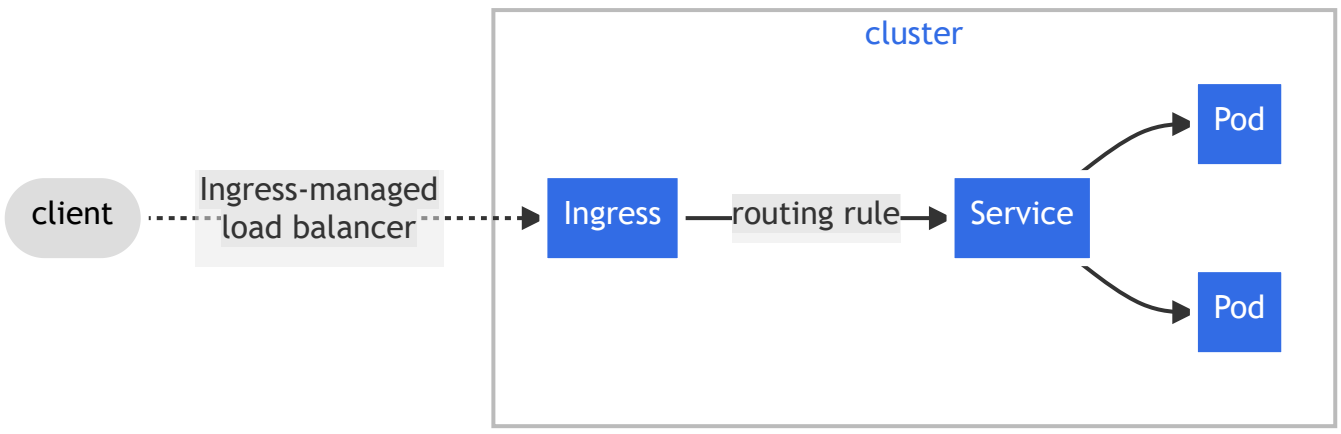


Figure. Ingress

An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An Ingress controller is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

Prerequisites

You must have an Ingress controller to satisfy an Ingress. Only creating an Ingress resource has no effect.

You may need to deploy an Ingress controller such as ingress-nginx. You can choose from a number of Ingress controllers.

Ideally, all Ingress controllers should fit the reference specification. In reality, the various Ingress controllers operate slightly differently.

> **Note:** Make sure you review your Ingress controller's documentation to understand the caveats of choosing it.

The Ingress resource

A minimal Ingress resource example:

service/networking/minimal-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

An Ingress needs `apiVersion`, `kind`, `metadata` and `spec` fields. The name of an Ingress object must be a valid DNS subdomain name. For general information about working with config files, see deploying applications, configuring containers, managing resources. Ingress frequently uses annotations to configure some options depending on the Ingress controller, an example of which is the rewrite-target annotation. Different Ingress controllers support different annotations. Review the documentation for your choice of Ingress controller to learn which annotations are supported.

The Ingress spec has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic.

If the `ingressClassName` is omitted, a default Ingress class should be defined.

There are some ingress controllers, that work without the definition of a default `IngressClass`. For example, the Ingress-NGINX controller can be configured with a flag `--watch-ingress-without-class`. It is recommended though, to specify the default `IngressClass` as shown below.

Ingress rules

Each HTTP rule contains the following information:

- An optional host. In this example, no host is specified, so the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, foo.bar.com), the rules apply to that host.
- A list of paths (for example, `/testpath`), each of which has an associated backend defined with a `service.name` and a `service.port.name` or `service.port.number`. Both the host and path must match the content of an incoming request before the load balancer directs traffic to the referenced Service.
- A backend is a combination of Service and port names as described in the Service doc or a custom resource backend by way of a CRD. HTTP (and HTTPS) requests to the Ingress that match the host and path of the rule are sent to the listed backend.

A `defaultBackend` is often configured in an Ingress controller to service any requests that do not match a path in the spec.

DefaultBackend

An Ingress with no rules sends all traffic to a single default backend and `.spec.defaultBackend` is the backend that should handle requests in that case. The `defaultBackend` is conventionally a configuration option of the Ingress controller and is not specified in your Ingress resources. If no `.spec.rules` are specified, `.spec.defaultBackend` must be specified. If `defaultBackend` is not set, the handling of requests that do not match any of the rules will be up to the ingress controller (consult the documentation for your ingress controller to find out how it handles this case).

If none of the hosts or paths match the HTTP request in the Ingress objects, the traffic is routed to your default backend.

Resource backends

A `Resource` backend is an ObjectRef to another Kubernetes resource within the same namespace as the Ingress object. A `Resource` is a mutually exclusive setting with Service, and will fail validation if both are specified. A common usage for a `Resource` backend is to ingress data to an object storage backend with static assets.

service/networking/ingress-resource-backend.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource-backend
spec:
  defaultBackend:
    resource:
      apiGroup: k8s.example.com
      kind: StorageBucket
      name: static-assets
  rules:
    - http:
        paths:
          - path: /icons
            pathType: ImplementationSpecific
            backend:
              resource:
                apiGroup: k8s.example.com
                kind: StorageBucket
                name: icon-assets
```

After creating the Ingress above, you can view it with the following command:

```
kubectl describe ingress ingress-resource-backend
```

```
Name:             ingress-resource-backend
Namespace:        default
Address:
Default backend:  APIGroup: k8s.example.com, Kind: StorageBucket, Name: static-assets
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /icons   APIGroup: k8s.example.com, Kind: StorageBucket, Name: icon-assets
Annotations:  <none>
Events:       <none>
```

Path types

Each path in an Ingress is required to have a corresponding path type. Paths that do not include an explicit `pathType` will fail validation. There are three supported path types:

- `ImplementationSpecific` : With this path type, matching is up to the IngressClass. Implementations can treat this as a separate `pathType` or treat it identically to `Prefix` or `Exact` path types.
- `Exact` : Matches the URL path exactly and with case sensitivity.
- `Prefix` : Matches based on a URL path prefix split by `/` . Matching is case sensitive and done on a path element by element basis. A path element refers to the list of labels in the path split by the `/` separator. A request is a match for path *p* if every *p* is an element-wise prefix of *p* of the request path.

> **Note:** If the last element of the path is a substring of the last element in request path, it is not a match (for example: `/foo/bar` matches `/foo/bar/baz`, but does not match `/foo/barbaz`).

Examples

Kind	Path(s)	Request path(s)	Matches?
Prefix	`/`	(all paths)	Yes
Exact	`/foo`	`/foo`	Yes
Exact	`/foo`	`/bar`	No
Exact	`/foo`	`/foo/`	No
Exact	`/foo/`	`/foo`	No
Prefix	`/foo`	`/foo` , `/foo/`	Yes
Prefix	`/foo/`	`/foo` , `/foo/`	Yes
Prefix	`/aaa/bb`	`/aaa/bbb`	No
Prefix	`/aaa/bbb`	`/aaa/bbb`	Yes
Prefix	`/aaa/bbb/`	`/aaa/bbb`	Yes, ignores trailing slash
Prefix	`/aaa/bbb`	`/aaa/bbb/`	Yes, matches trailing slash
Prefix	`/aaa/bbb`	`/aaa/bbb/ccc`	Yes, matches subpath
Prefix	`/aaa/bbb`	`/aaa/bbbxyz`	No, does not match string prefix
Prefix	`/` , `/aaa`	`/aaa/ccc`	Yes, matches `/aaa` prefix
Prefix	`/` , `/aaa` , `/aaa/bbb`	`/aaa/bbb`	Yes, matches `/aaa/bbb` prefix
Prefix	`/` , `/aaa` , `/aaa/bbb`	`/ccc`	Yes, matches `/` prefix
Prefix	`/aaa`	`/ccc`	No, uses default backend
Mixed	`/foo` (Prefix), `/foo` (Exact)	`/foo`	Yes, prefers Exact

Multiple matches

In some cases, multiple paths within an Ingress will match a request. In those cases precedence will be given first to the longest matching path. If two paths are still equally matched, precedence will be given to paths with an exact path type over prefix path type.

Hostname wildcards

Hosts can be precise matches (for example “ `foo.bar.com` ”) or a wildcard (for example “ `*.foo.com` ”). Precise matches require that the HTTP `host` header matches the `host` field. Wildcard matches require the HTTP `host` header is equal to the suffix of the wildcard rule.

Host	Host header	Match?
`*.foo.com`	`bar.foo.com`	Matches based on shared suffix
`*.foo.com`	`baz.bar.foo.com`	No match, wildcard only covers a single DNS label
`*.foo.com`	`foo.com`	No match, wildcard only covers a single DNS label

service/networking/ingress-wildcard-host.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "foo.bar.com"
    http:
      paths:
      - pathType: Prefix
        path: "/bar"
        backend:
          service:
            name: service1
            port:
              number: 80
  - host: "*.foo.com"
    http:
      paths:
      - pathType: Prefix
        path: "/foo"
        backend:
          service:
            name: service2
            port:
              number: 80
```

Ingress class

Ingresses can be implemented by different controllers, often with different configuration. Each Ingress should specify a class, a reference to an IngressClass resource that contains additional configuration including the name of the controller that should implement the class.

service/networking/external-lb.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: external-lb
spec:
  controller: example.com/ingress-controller
  parameters:
    apiGroup: k8s.example.com
    kind: IngressParameters
    name: external-lb
```

The `.spec.parameters` field of an IngressClass lets you reference another resource that provides configuration related to that IngressClass.

The specific type of parameters to use depends on the ingress controller that you specify in the `.spec.controller` field of the IngressClass.

IngressClass scope

Depending on your ingress controller, you may be able to use parameters that you set cluster-wide, or just for one namespace.

Cluster | Namespaced

The default scope for IngressClass parameters is cluster-wide.

If you set the `.spec.parameters` field and don't set `.spec.parameters.scope`, or if you set `.spec.parameters.scope` to `Cluster`, then the IngressClass refers to a cluster-scoped resource. The `kind` (in combination the `apiGroup`) of the parameters refers to a cluster-scoped API (possibly a custom resource), and the `name` of the parameters identifies a specific cluster scoped resource for that API.

For example:

```yaml
---
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: external-lb-1
spec:
  controller: example.com/ingress-controller
  parameters:
    # The parameters for this IngressClass are specified in a
    # ClusterIngressParameter (API group k8s.example.net) named
    # "external-config-1". This definition tells Kubernetes to
    # look for a cluster-scoped parameter resource.
    scope: Cluster
    apiGroup: k8s.example.net
    kind: ClusterIngressParameter
    name: external-config-1
```

Deprecated annotation

Before the IngressClass resource and `ingressClassName` field were added in Kubernetes 1.18, Ingress classes were specified with a `kubernetes.io/ingress.class` annotation on the Ingress. This annotation was never formally defined, but was widely supported by Ingress controllers.

The newer `ingressClassName` field on Ingresses is a replacement for that annotation, but is not a direct equivalent. While the annotation was generally used to reference the name of the Ingress controller that should implement the Ingress, the field is a reference to an IngressClass resource that contains additional Ingress configuration, including the name of the Ingress controller.

Default IngressClass

You can mark a particular IngressClass as default for your cluster. Setting the `ingressclass.kubernetes.io/is-default-class` annotation to `true` on an IngressClass resource will ensure that new Ingresses without an `ingressClassName` field specified will be assigned this default IngressClass.

> **Caution:** If you have more than one IngressClass marked as the default for your cluster, the admission controller prevents creating new Ingress objects that don't have an `ingressClassName` specified. You can resolve this by ensuring that at most 1 IngressClass is marked as default in your cluster.

There are some ingress controllers, that work without the definition of a default `IngressClass`. For example, the Ingress-NGINX controller can be configured with a flag `--watch-ingress-without-class`. It is recommended though, to specify the default `IngressClass`:

service/networking/default-ingressclass.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  labels:
    app.kubernetes.io/component: controller
  name: nginx-example
  annotations:
    ingressclass.kubernetes.io/is-default-class: "true"
spec:
  controller: k8s.io/ingress-nginx
```

Types of Ingress

Ingress backed by a single Service

There are existing Kubernetes concepts that allow you to expose a single Service (see alternatives). You can also do this with an Ingress by specifying a *default backend* with no rules.

service/networking/test-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  defaultBackend:
    service:
      name: test
      port:
        number: 80
```

If you create it using `kubectl apply -f` you should be able to view the state of the Ingress you added:

```
kubectl get ingress test-ingress
```

```
NAME           CLASS         HOSTS   ADDRESS         PORTS   AGE
test-ingress   external-lb   *       203.0.113.123   80      59s
```

Where `203.0.113.123` is the IP allocated by the Ingress controller to satisfy this Ingress.

> **Note:** Ingress controllers and load balancers may take a minute or two to allocate an IP address. Until that time, you often see the address listed as `<pending>`.

Simple fanout

A fanout configuration routes traffic from a single IP address to more than one Service, based on the HTTP URI being requested. An Ingress allows you to keep the number of load balancers down to a minimum. For example, a setup like:

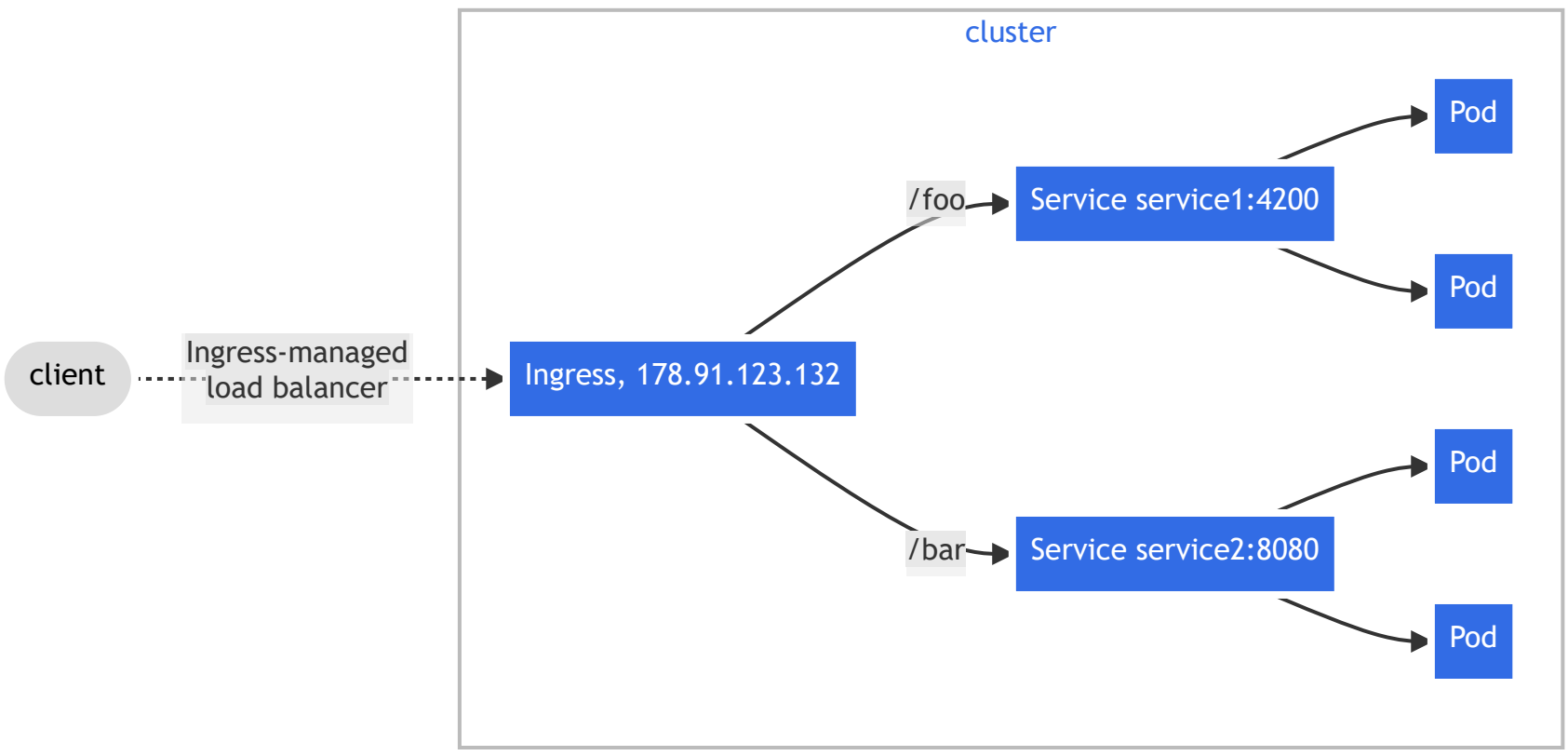


Figure. Ingress Fan Out

It would require an Ingress such as:

service/networking/simple-fanout-example.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 4200
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: service2
            port:
              number: 8080
```

When you create the Ingress with `kubectl apply -f`:

```
kubectl describe ingress simple-fanout-example
```

```
Name:             simple-fanout-example
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:4200 (10.8.0.90:4200)
               /bar   service2:8080 (10.8.0.91:8080)
Events:
  Type     Reason  Age                From                     Message
  ----     ------  ----               ----                     -------
  Normal   ADD     22s                loadbalancer-controller  default/test
```

The Ingress controller provisions an implementation-specific load balancer that satisfies the Ingress, as long as the Services (`service1`, `service2`) exist. When it has done so, you can see the address of the load balancer at the Address field.

Note: Depending on the Ingress controller you are using, you may need to create a default-http-backend Service.

Name based virtual hosting

Name-based virtual hosts support routing HTTP traffic to multiple host names at the same IP address.

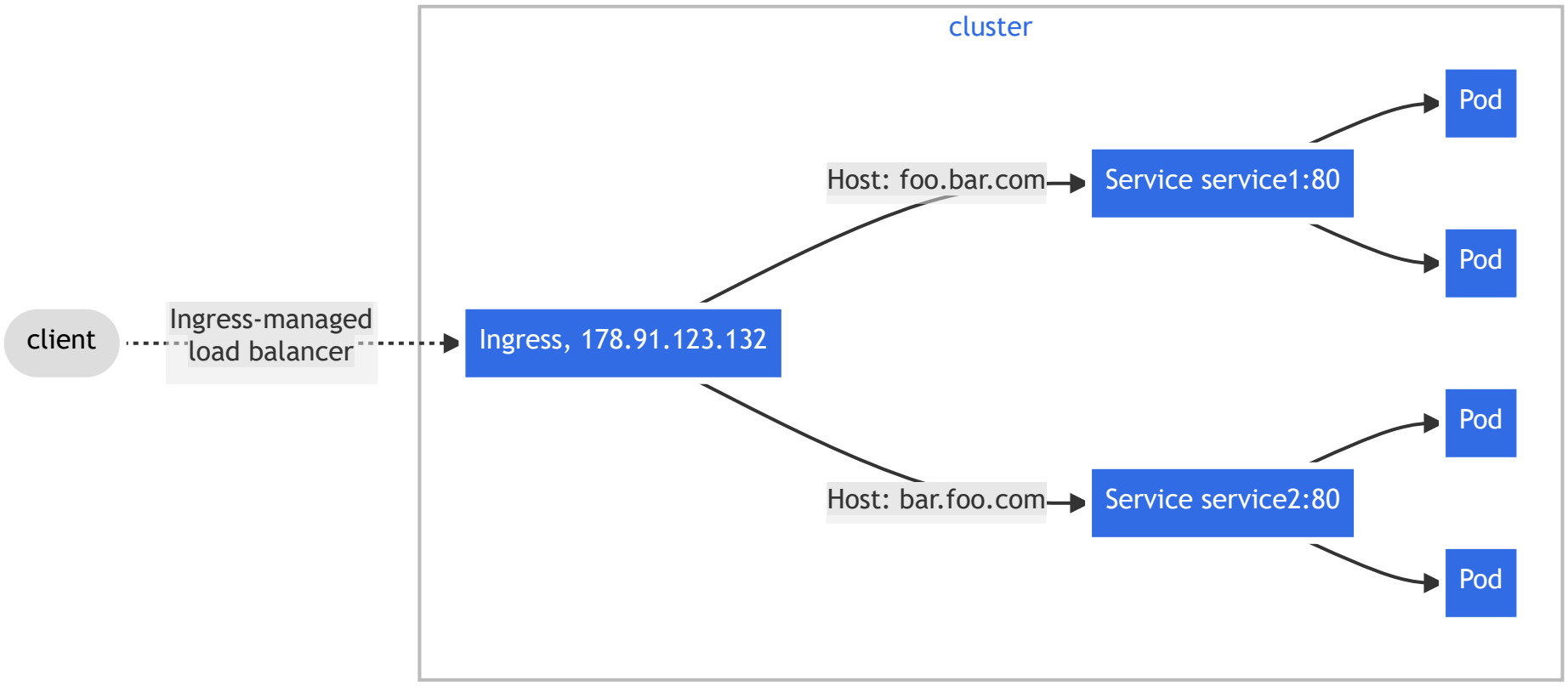


Figure. Ingress Name Based Virtual hosting

The following Ingress tells the backing load balancer to route requests based on the Host header.

service/networking/name-virtual-host-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service1
            port:
              number: 80
  - host: bar.foo.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service2
            port:
              number: 80
```

If you create an Ingress resource without any hosts defined in the rules, then any web traffic to the IP address of your Ingress controller can be matched without a name based virtual host being required.

For example, the following Ingress routes traffic requested for `first.bar.com` to `service1`, `second.bar.com` to `service2`, and any traffic whose request host header doesn't match `first.bar.com` and `second.bar.com` to `service3`.

service/networking/name-virtual-host-ingress-no-third-host.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress-no-third-host
spec:
  rules:
  - host: first.bar.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service1
            port:
              number: 80
  - host: second.bar.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service2
            port:
              number: 80
  - http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service3
            port:
              number: 80
```

TLS

You can secure an Ingress by specifying a Secret that contains a TLS private key and certificate. The Ingress resource only supports a single TLS port, 443, and assumes TLS termination at the ingress point (traffic to the Service and its Pods is in plaintext). If the TLS configuration section in an Ingress specifies different hosts, they are multiplexed on the same port according to the hostname specified through the SNI TLS extension (provided the Ingress controller supports SNI). The TLS secret must contain keys named `tls.crt` and `tls.key` that contain the certificate and private key to use for TLS. For example:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

Referencing this secret in an Ingress tells the Ingress controller to secure the channel from the client to the load balancer using TLS. You need to make sure the TLS secret you created came from a certificate that contains a Common Name (CN), also known as a Fully Qualified Domain Name (FQDN) for `https-example.foo.com` .

> **Note:** Keep in mind that TLS will not work on the default rule because the certificates would have to be issued for all the

possible sub-domains. Therefore, `hosts` in the `tls` section need to explicitly match the `host` in the `rules` section.

service/networking/tls-example-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
  - hosts:
      - https-example.foo.com
    secretName: testsecret-tls
  rules:
  - host: https-example.foo.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 80
```

Note: There is a gap between TLS features supported by various Ingress controllers. Please refer to documentation on nginx, GCE, or any other platform specific Ingress controller to understand how TLS works in your environment.

Load balancing

An Ingress controller is bootstrapped with some load balancing policy settings that it applies to all Ingress, such as the load balancing algorithm, backend weight scheme, and others. More advanced load balancing concepts (e.g. persistent sessions, dynamic weights) are not yet exposed through the Ingress. You can instead get these features through the load balancer used for a Service.

It's also worth noting that even though health checks are not exposed directly through the Ingress, there exist parallel concepts in Kubernetes such as readiness probes that allow you to achieve the same end result. Please review the controller specific documentation to see how they handle health checks (for example: nginx, or GCE).

Updating an Ingress

To update an existing Ingress to add a new Host, you can update it by editing the resource:

```
kubectl describe ingress test
```

```
Name:             test
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:80 (10.8.0.90:80)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age               From                     Message
  ----     ------  ----              ----                     -------
  Normal   ADD     35s               loadbalancer-controller  default/test
```

```
kubectl edit ingress test
```

This pops up an editor with the existing configuration in YAML format. Modify it to include the new Host:

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          service:
            name: service1
            port:
              number: 80
        path: /foo
        pathType: Prefix
  - host: bar.baz.com
    http:
      paths:
      - backend:
          service:
            name: service2
            port:
              number: 80
        path: /foo
        pathType: Prefix
..
```

After you save your changes, kubectl updates the resource in the API server, which tells the Ingress controller to reconfigure the load balancer.

Verify this:

```
kubectl describe ingress test
```

```
Name:             test
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:80 (10.8.0.90:80)
  bar.baz.com
               /foo   service2:80 (10.8.0.91:80)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age               From                     Message
  ----     ------  ----              ----                     -------
  Normal   ADD     45s               loadbalancer-controller  default/test
```

You can achieve the same outcome by invoking `kubectl replace -f` on a modified Ingress YAML file.

Failing across availability zones

Techniques for spreading traffic across failure domains differ between cloud providers. Please check the documentation of the relevant Ingress controller for details.

Alternatives

You can expose a Service in multiple ways that don't directly involve the Ingress resource:

- Use Service.Type=LoadBalancer
- Use Service.Type=NodePort

What's next

- Learn about the Ingress API
- Learn about Ingress controllers
- Set up Ingress on Minikube with the NGINX Controller

3 - Ingress Controllers

In order for an Ingress to work in your cluster, there must be an *ingress controller* running. You need to select at least one ingress controller and make sure it is set up in your cluster. This page lists common ingress controllers that you can deploy.

In order for the Ingress resource to work, the cluster must have an ingress controller running.

Unlike other types of controllers which run as part of the `kube-controller-manager` binary, Ingress controllers are not started automatically with a cluster. Use this page to choose the ingress controller implementation that best fits your cluster.

Kubernetes as a project supports and maintains AWS, GCE, and nginx ingress controllers.

Additional controllers

> **Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

- AKS Application Gateway Ingress Controller is an ingress controller that configures the Azure Application Gateway.
- Alibaba Cloud MSE Ingress is an ingress controller that configures the Alibaba Cloud Native Gateway, which is also the commercial version of Higress.
- Apache APISIX ingress controller is an Apache APISIX-based ingress controller.
- Avi Kubernetes Operator provides L4-L7 load-balancing using VMware NSX Advanced Load Balancer.
- BFE Ingress Controller is a BFE-based ingress controller.
- Cilium Ingress Controller is an ingress controller powered by Cilium.
- The Citrix ingress controller works with Citrix Application Delivery Controller.
- Contour is an Envoy based ingress controller.
- Emissary-Ingress API Gateway is an Envoy-based ingress controller.
- EnRoute is an Envoy based API gateway that can run as an ingress controller.
- Easegress IngressController is an Easegress based API gateway that can run as an ingress controller.
- F5 BIG-IP Container Ingress Services for Kubernetes lets you use an Ingress to configure F5 BIG-IP virtual servers.
- FortiADC Ingress Controller support the Kubernetes Ingress resources and allows you to manage FortiADC objects from Kubernetes
- Gloo is an open-source ingress controller based on Envoy, which offers API gateway functionality.
- HAProxy Ingress is an ingress controller for HAProxy.
- Higress is an Envoy based API gateway that can run as an ingress controller.
- The HAProxy Ingress Controller for Kubernetes is also an ingress controller for HAProxy.
- Istio Ingress is an Istio based ingress controller.
- The Kong Ingress Controller for Kubernetes is an ingress controller driving Kong Gateway.
- Kusk Gateway is an OpenAPI-driven ingress controller based on Envoy.
- The NGINX Ingress Controller for Kubernetes works with the NGINX webserver (as a proxy).
- The ngrok Kubernetes Ingress Controller is an open source controller for adding secure public access to your K8s services using the ngrok platform.
- The OCI Native Ingress Controller is an Ingress controller for Oracle Cloud Infrastructure which allows you to manage the OCI Load Balancer.
- The Pomerium Ingress Controller is based on Pomerium, which offers context-aware access policy.
- Skipper HTTP router and reverse proxy for service composition, including use cases like Kubernetes Ingress, designed as a library to build your custom proxy.
- The Traefik Kubernetes Ingress provider is an ingress controller for the Traefik proxy.
- Tyk Operator extends Ingress with Custom Resources to bring API Management capabilities to Ingress. Tyk Operator works with the Open Source Tyk Gateway & Tyk Cloud control plane.
- Voyager is an ingress controller for HAProxy.
- Wallarm Ingress Controller is an Ingress Controller that provides WAAP (WAF) and API Security capabilities.

Using multiple Ingress controllers

You may deploy any number of ingress controllers using ingress class within a cluster. Note the `.metadata.name` of your ingress class resource. When you create an ingress you would need that name to specify the `ingressClassName` field on your Ingress object (refer to IngressSpec v1 reference). `ingressClassName` is a replacement of the older annotation method.

If you do not specify an IngressClass for an Ingress, and your cluster has exactly one IngressClass marked as default, then Kubernetes applies the cluster's default IngressClass to the Ingress. You mark an IngressClass as default by setting the `ingressclass.kubernetes.io/is-default-class` annotation on that IngressClass, with the string value `"true"`.

Ideally, all ingress controllers should fulfill this specification, but the various ingress controllers operate slightly differently.

> **Note:** Make sure you review your ingress controller's documentation to understand the caveats of choosing it.

What's next

- Learn more about Ingress.
- Set up Ingress on Minikube with the NGINX Controller.

4 - Gateway API

Gateway API is a family of API kinds that provide dynamic infrastructure provisioning and advanced traffic routing.

Make network services available by using an extensible, role-oriented, protocol-aware configuration mechanism. Gateway API is an add-on containing API kinds that provide dynamic infrastructure provisioning and advanced traffic routing.

Design principles

The following principles shaped the design and architecture of Gateway API:

- **Role-oriented:** Gateway API kinds are modeled after organizational roles that are responsible for managing Kubernetes service networking:
  - **Infrastructure Provider:** Manages infrastructure that allows multiple isolated clusters to serve multiple tenants, e.g. a cloud provider.
  - **Cluster Operator:** Manages clusters and is typically concerned with policies, network access, application permissions, etc.
  - **Application Developer:** Manages an application running in a cluster and is typically concerned with application-level configuration and Service composition.
- **Portable:** Gateway API specifications are defined as custom resources and are supported by many implementations.
- **Expressive:** Gateway API kinds support functionality for common traffic routing use cases such as header-based matching, traffic weighting, and others that were only possible in Ingress by using custom annotations.
- **Extensible:** Gateway allows for custom resources to be linked at various layers of the API. This makes granular customization possible at the appropriate places within the API structure.

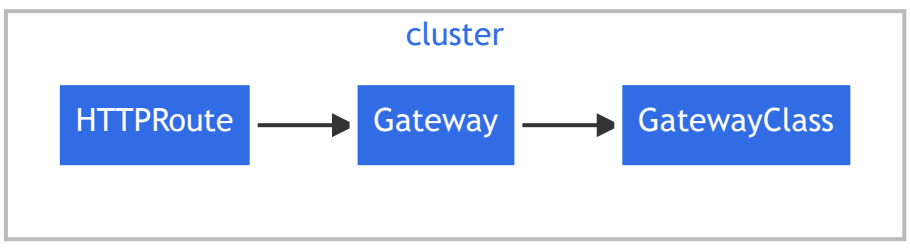
Resource model

Gateway API has three stable API kinds:

- **GatewayClass:** Defines a set of gateways with common configuration and managed by a controller that implements the class.
- **Gateway:** Defines an instance of traffic handling infrastructure, such as cloud load balancer.
- **HTTPRoute:** Defines HTTP-specific rules for mapping traffic from a Gateway listener to a representation of backend network endpoints. These endpoints are often represented as a Service.

Gateway API is organized into different API kinds that have interdependent relationships to support the role-oriented nature of organizations. A Gateway object is associated with exactly one GatewayClass; the GatewayClass describes the gateway controller responsible for managing Gateways of this class. One or more route kinds such as HTTPRoute, are then associated to Gateways. A Gateway can filter the routes that may be attached to its `listeners`, forming a bidirectional trust model with routes.

The following figure illustrates the relationships of the three stable Gateway API kinds:



GatewayClass

Gateways can be implemented by different controllers, often with different configurations. A Gateway must reference a GatewayClass that contains the name of the controller that implements the class.

A minimal GatewayClass example:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: example-class
spec:
  controllerName: example.com/gateway-controller
```

In this example, a controller that has implemented Gateway API is configured to manage GatewayClasses with the controller name `example.com/gateway-controller`. Gateways of this class will be managed by the implementation's controller.

See the GatewayClass reference for a full definition of this API kind.

Gateway

A Gateway describes an instance of traffic handling infrastructure. It defines a network endpoint that can be used for processing traffic, i.e. filtering, balancing, splitting, etc. for backends such as a Service. For example, a Gateway may represent a cloud load balancer or an in-cluster proxy server that is configured to accept HTTP traffic.

A minimal Gateway resource example:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: example-gateway
spec:
  gatewayClassName: example-class
  listeners:
  - name: http
    protocol: HTTP
    port: 80
```

In this example, an instance of traffic handling infrastructure is programmed to listen for HTTP traffic on port 80. Since the `addresses` field is unspecified, an address or hostname is assigned to the Gateway by the implementation's controller. This address is used as a network endpoint for processing traffic of backend network endpoints defined in routes.

See the Gateway reference for a full definition of this API kind.

HTTPRoute

The HTTPRoute kind specifies routing behavior of HTTP requests from a Gateway listener to backend network endpoints. For a Service backend, an implementation may represent the backend network endpoint as a Service IP or the backing Endpoints of the Service. An HTTPRoute represents configuration that is applied to the underlying Gateway implementation. For example, defining a new HTTPRoute may result in configuring additional traffic routes in a cloud load balancer or in-cluster proxy server.

A minimal HTTPRoute example:

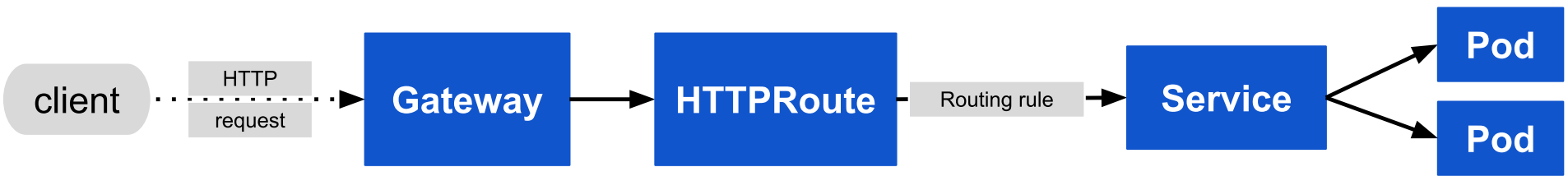
```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-httproute
spec:
  parentRefs:
  - name: example-gateway
  hostnames:
  - "www.example.com"
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /login
    backendRefs:
    - name: example-svc
      port: 8080
```

In this example, HTTP traffic from Gateway `example-gateway` with the Host: header set to `www.example.com` and the request path specified as `/login` will be routed to Service `example-svc` on port `8080`.

See the HTTPRoute reference for a full definition of this API kind.

Request flow

Here is a simple example of HTTP traffic being routed to a Service by using a Gateway and an HTTPRoute:



In this example, the request flow for a Gateway implemented as a reverse proxy is:

1. The client starts to prepare an HTTP request for the URL `http://www.example.com`
2. The client's DNS resolver queries for the destination name and learns a mapping to one or more IP addresses associated with the Gateway.
3. The client sends a request to the Gateway IP address; the reverse proxy receives the HTTP request and uses the Host: header to match a configuration that was derived from the Gateway and attached HTTPRoute.
4. Optionally, the reverse proxy can perform request header and/or path matching based on match rules of the HTTPRoute.
5. Optionally, the reverse proxy can modify the request; for example, to add or remove headers, based on filter rules of the HTTPRoute.
6. Lastly, the reverse proxy forwards the request to one or more backends.

Conformance

Gateway API covers a broad set of features and is widely implemented. This combination requires clear conformance definitions and tests to ensure that the API provides a consistent experience wherever it is used.

See the conformance documentation to understand details such as release channels, support levels, and running conformance tests.

Migrating from Ingress

Gateway API is the successor to the Ingress API. However, it does not include the Ingress kind. As a result, a one-time conversion from your existing Ingress resources to Gateway API resources is necessary.

Refer to the ingress migration guide for details on migrating Ingress resources to Gateway API resources.

What's next

Instead of Gateway API resources being natively implemented by Kubernetes, the specifications are defined as Custom Resources supported by a wide range of implementations. Install the Gateway API CRDs or follow the installation instructions of your selected implementation. After installing an implementation, use the Getting Started guide to help you quickly start working with Gateway API.

> **Note:** Make sure to review the documentation of your selected implementation to understand any caveats.

Refer to the API specification for additional details of all Gateway API kinds.

5 - EndpointSlices

The EndpointSlice API is the mechanism that Kubernetes uses to let your Service scale to handle large numbers of backends, and allows the cluster to update its list of healthy backends efficiently.

FEATURE STATE: `Kubernetes v1.21 [stable]`

Kubernetes' *EndpointSlice* API provides a way to track network endpoints within a Kubernetes cluster. EndpointSlices offer a more scalable and extensible alternative to Endpoints.

EndpointSlice API

In Kubernetes, an EndpointSlice contains references to a set of network endpoints. The control plane automatically creates EndpointSlices for any Kubernetes Service that has a selector specified. These EndpointSlices include references to all the Pods that match the Service selector. EndpointSlices group network endpoints together by unique combinations of protocol, port number, and Service name. The name of a EndpointSlice object must be a valid DNS subdomain name.

As an example, here's a sample EndpointSlice object, that's owned by the `example` Kubernetes Service.

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: example-abc
  labels:
    kubernetes.io/service-name: example
addressType: IPv4
ports:
  - name: http
    protocol: TCP
    port: 80
endpoints:
  - addresses:
      - "10.1.2.3"
    conditions:
      ready: true
    hostname: pod-1
    nodeName: node-1
    zone: us-west2-a
```

By default, the control plane creates and manages EndpointSlices to have no more than 100 endpoints each. You can configure this with the `--max-endpoints-per-slice` kube-controller-manager flag, up to a maximum of 1000.

EndpointSlices can act as the source of truth for kube-proxy when it comes to how to route internal traffic.

Address types

EndpointSlices support three address types:

- IPv4
- IPv6
- FQDN (Fully Qualified Domain Name)

Each `EndpointSlice` object represents a specific IP address type. If you have a Service that is available via IPv4 and IPv6, there will be at least two `EndpointSlice` objects (one for IPv4, and one for IPv6).

Conditions

The EndpointSlice API stores conditions about endpoints that may be useful for consumers. The three conditions are `ready`, `serving`, and `terminating`.

Ready

`ready` is a condition that maps to a Pod's `Ready` condition. A running Pod with the `Ready` condition set to `True` should have this EndpointSlice condition also set to `true`. For compatibility reasons, `ready` is NEVER `true` when a Pod is terminating. Consumers should refer to the `serving` condition to inspect the readiness of terminating Pods. The only exception to this rule is for Services with `spec.publishNotReadyAddresses` set to `true`. Endpoints for these Services will always have the `ready` condition set to `true`.

Serving

FEATURE STATE: `Kubernetes v1.26 [stable]`

The `serving` condition is almost identical to the `ready` condition. The difference is that consumers of the EndpointSlice API should check the `serving` condition if they care about pod readiness while the pod is also terminating.

> **Note:** Although `serving` is almost identical to `ready`, it was added to prevent breaking the existing meaning of `ready`. It may be unexpected for existing clients if `ready` could be `true` for terminating endpoints, since historically terminating endpoints were never included in the Endpoints or EndpointSlice API to begin with. For this reason, `ready` is *always* `false` for terminating endpoints, and a new condition `serving` was added in v1.20 so that clients can track readiness for terminating pods independent of the existing semantics for `ready`.

Terminating

FEATURE STATE: `Kubernetes v1.22 [beta]`

`Terminating` is a condition that indicates whether an endpoint is terminating. For pods, this is any pod that has a deletion timestamp set.

Topology information

Each endpoint within an EndpointSlice can contain relevant topology information. The topology information includes the location of the endpoint and information about the corresponding Node and zone. These are available in the following per endpoint fields on EndpointSlices:

- `nodeName` - The name of the Node this endpoint is on.
- `zone` - The zone this endpoint is in.

> **Note:**
> In the v1 API, the per endpoint `topology` was effectively removed in favor of the dedicated fields `nodeName` and `zone`.
>
> Setting arbitrary topology fields on the `endpoint` field of an `EndpointSlice` resource has been deprecated and is not supported in the v1 API. Instead, the v1 API supports setting individual `nodeName` and `zone` fields. These fields are automatically translated between API versions. For example, the value of the `"topology.kubernetes.io/zone"` key in the `topology` field in the v1beta1 API is accessible as the `zone` field in the v1 API.

Management

Most often, the control plane (specifically, the endpoint slice controller) creates and manages EndpointSlice objects. There are a variety of other use cases for EndpointSlices, such as service mesh implementations, that could result in other entities or controllers managing additional sets of EndpointSlices.

To ensure that multiple entities can manage EndpointSlices without interfering with each other, Kubernetes defines the label `endpointslice.kubernetes.io/managed-by`, which indicates the entity managing an EndpointSlice. The endpoint slice controller sets `endpointslice-controller.k8s.io` as the value for this label on all EndpointSlices it manages. Other entities managing EndpointSlices should also set a unique value for this label.

Ownership

In most use cases, EndpointSlices are owned by the Service that the endpoint slice object tracks endpoints for. This ownership is indicated by an owner reference on each EndpointSlice as well as a `kubernetes.io/service-name` label that enables simple lookups of all EndpointSlices belonging to a Service.

EndpointSlice mirroring

In some cases, applications create custom Endpoints resources. To ensure that these applications do not need to concurrently write to both Endpoints and EndpointSlice resources, the cluster's control plane mirrors most Endpoints resources to corresponding EndpointSlices.

The control plane mirrors Endpoints resources unless:

- the Endpoints resource has a `endpointslice.kubernetes.io/skip-mirror` label set to `true`.
- the Endpoints resource has a `control-plane.alpha.kubernetes.io/leader` annotation.
- the corresponding Service resource does not exist.
- the corresponding Service resource has a non-nil selector.

Individual Endpoints resources may translate into multiple EndpointSlices. This will occur if an Endpoints resource has multiple subsets or includes endpoints with multiple IP families (IPv4 and IPv6). A maximum of 1000 addresses per subset will be mirrored to EndpointSlices.

Distribution of EndpointSlices

Each EndpointSlice has a set of ports that applies to all endpoints within the resource. When named ports are used for a Service, Pods may end up with different target port numbers for the same named port, requiring different EndpointSlices. This is similar to the logic behind how subsets are grouped with Endpoints.

The control plane tries to fill EndpointSlices as full as possible, but does not actively rebalance them. The logic is fairly straightforward:

1. Iterate through existing EndpointSlices, remove endpoints that are no longer desired and update matching endpoints that have changed.
2. Iterate through EndpointSlices that have been modified in the first step and fill them up with any new endpoints needed.
3. If there's still new endpoints left to add, try to fit them into a previously unchanged slice and/or create new ones.

Importantly, the third step prioritizes limiting EndpointSlice updates over a perfectly full distribution of EndpointSlices. As an example, if there are 10 new endpoints to add and 2 EndpointSlices with room for 5 more endpoints each, this approach will create a new EndpointSlice instead of filling up the 2 existing EndpointSlices. In other words, a single EndpointSlice creation is preferable to multiple EndpointSlice updates.

With kube-proxy running on each Node and watching EndpointSlices, every change to an EndpointSlice becomes relatively expensive since it will be transmitted to every Node in the cluster. This approach is intended to limit the number of changes that need to be sent to every Node, even if it may result with multiple EndpointSlices that are not full.

In practice, this less than ideal distribution should be rare. Most changes processed by the EndpointSlice controller will be small enough to fit in an existing EndpointSlice, and if not, a new EndpointSlice is likely going to be necessary soon anyway. Rolling updates of Deployments also provide a natural repacking of EndpointSlices with all Pods and their corresponding endpoints getting replaced.

Duplicate endpoints

Due to the nature of EndpointSlice changes, endpoints may be represented in more than one EndpointSlice at the same time. This naturally occurs as changes to different EndpointSlice objects can arrive at the Kubernetes client watch / cache at different times.

> **Note:**
> Clients of the EndpointSlice API must iterate through all the existing EndpointSlices associated to a Service and build a complete list of unique network endpoints. It is important to mention that endpoints may be duplicated in different EndpointSlices.
>
> You can find a reference implementation for how to perform this endpoint aggregation and deduplication as part of the `EndpointSliceCache` code within `kube-proxy`.

Comparison with Endpoints

The original Endpoints API provided a simple and straightforward way of tracking network endpoints in Kubernetes. As Kubernetes clusters and Services grew to handle more traffic and to send more traffic to more backend Pods, the limitations of that original API became more visible. Most notably, those included challenges with scaling to larger numbers of network endpoints.

Since all network endpoints for a Service were stored in a single Endpoints object, those Endpoints objects could get quite large. For Services that stayed stable (the same set of endpoints over a long period of time) the impact was less noticeable; even then, some use cases of Kubernetes weren't well served.

When a Service had a lot of backend endpoints and the workload was either scaling frequently, or rolling out new changes frequently, each update to the single Endpoints object for that Service meant a lot of traffic between Kubernetes cluster components (within the control plane, and also between nodes and the API server). This extra traffic also had a cost in terms of CPU use.

With EndpointSlices, adding or removing a single Pod triggers the same *number* of updates to clients that are watching for changes, but the size of those update message is much smaller at large scale.

EndpointSlices also enabled innovation around new features such dual-stack networking and topology-aware routing.

What's next

- Follow the Connecting Applications with Services tutorial
- Read the API reference for the EndpointSlice API
- Read the API reference for the Endpoints API

6 - Network Policies

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), NetworkPolicies allow you to specify rules for traffic flow within your cluster, and also between Pods and the outside world. Your cluster must use a network plugin that supports NetworkPolicy enforcement.

If you want to control traffic flow at the IP address or port level for TCP, UDP, and SCTP protocols, then you might consider using Kubernetes NetworkPolicies for particular applications in your cluster. NetworkPolicies are an application-centric construct which allow you to specify how a pod is allowed to communicate with various network "entities" (we use the word "entity" here to avoid overloading the more common terms such as "endpoints" and "services", which have specific Kubernetes connotations) over the network. NetworkPolicies apply to a connection with a pod on one or both ends, and are not relevant to other connections.

The entities that a Pod can communicate with are identified through a combination of the following 3 identifiers:

1. Other pods that are allowed (exception: a pod cannot block access to itself)
2. Namespaces that are allowed
3. IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)

When defining a pod- or namespace- based NetworkPolicy, you use a selector to specify what traffic is allowed to and from the Pod(s) that match the selector.

Meanwhile, when IP based NetworkPolicies are created, we define policies based on IP blocks (CIDR ranges).

Prerequisites

Network policies are implemented by the network plugin. To use network policies, you must be using a networking solution which supports NetworkPolicy. Creating a NetworkPolicy resource without a controller that implements it will have no effect.

The Two Sorts of Pod Isolation

There are two sorts of isolation for a pod: isolation for egress, and isolation for ingress. They concern what connections may be established. "Isolation" here is not absolute, rather it means "some restrictions apply". The alternative, "non-isolated for $direction", means that no restrictions apply in the stated direction. The two sorts of isolation (or not) are declared independently, and are both relevant for a connection from one pod to another.

By default, a pod is non-isolated for egress; all outbound connections are allowed. A pod is isolated for egress if there is any NetworkPolicy that both selects the pod and has "Egress" in its `policyTypes`; we say that such a policy applies to the pod for egress. When a pod is isolated for egress, the only allowed connections from the pod are those allowed by the `egress` list of some NetworkPolicy that applies to the pod for egress. Reply traffic for those allowed connections will also be implicitly allowed. The effects of those `egress` lists combine additively.

By default, a pod is non-isolated for ingress; all inbound connections are allowed. A pod is isolated for ingress if there is any NetworkPolicy that both selects the pod and has "Ingress" in its `policyTypes`; we say that such a policy applies to the pod for ingress. When a pod is isolated for ingress, the only allowed connections into the pod are those from the pod's node and those allowed by the `ingress` list of some NetworkPolicy that applies to the pod for ingress. Reply traffic for those allowed connections will also be implicitly allowed. The effects of those `ingress` lists combine additively.

Network policies do not conflict; they are additive. If any policy or policies apply to a given pod for a given direction, the connections allowed in that direction from that pod is the union of what the applicable policies allow. Thus, order of evaluation does not affect the policy result.

For a connection from a source pod to a destination pod to be allowed, both the egress policy on the source pod and the ingress policy on the destination pod need to allow the connection. If either side does not allow the connection, it will not happen.

The NetworkPolicy resource

See the NetworkPolicy reference for a full definition of the resource.

An example NetworkPolicy might look like this:

service/networking/networkpolicy.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

Note: POSTing this to the API server for your cluster will have no effect unless your chosen networking solution supports network policy.

Mandatory Fields: As with all other Kubernetes config, a NetworkPolicy needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see Configure a Pod to Use a ConfigMap, and Object Management.

spec: NetworkPolicy spec has all the information needed to define a particular network policy in the given namespace.

podSelector: Each NetworkPolicy includes a `podSelector` which selects the grouping of pods to which the policy applies. The example policy selects pods with the label "role=db". An empty `podSelector` selects all pods in the namespace.

policyTypes: Each NetworkPolicy includes a `policyTypes` list which may include either `Ingress`, `Egress`, or both. The `policyTypes` field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no `policyTypes` are specified on a NetworkPolicy then by default `Ingress` will always be set and `Egress` will be set if the NetworkPolicy has any egress rules.

ingress: Each NetworkPolicy may include a list of allowed `ingress` rules. Each rule allows traffic which matches both the `from` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port, from one of three sources, the first specified via an `ipBlock`, the second via a `namespaceSelector` and the third via a `podSelector`.

egress: Each NetworkPolicy may include a list of allowed `egress` rules. Each rule allows traffic which matches both the `to` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port to any destination in `10.0.0.0/24`.

So, the example NetworkPolicy:

1. isolates `role=db` pods in the `default` namespace for both ingress and egress traffic (if they weren't already isolated)
2. (Ingress rules) allows connections to all pods in the `default` namespace with the label `role=db` on TCP port 6379 from:
    - any pod in the `default` namespace with the label `role=frontend`
    - any pod in a namespace with the label `project=myproject`
    - IP addresses in the ranges `172.17.0.0` – `172.17.0.255` and `172.17.2.0` – `172.17.255.255` (ie, all of `172.17.0.0/16` except `172.17.1.0/24`)
3. (Egress rules) allows connections from any pod in the `default` namespace with the label `role=db` to CIDR `10.0.0.0/24` on TCP port 5978

See the Declare Network Policy walkthrough for further examples.

Behavior of `to` and `from` selectors

There are four kinds of selectors that can be specified in an `ingress` `from` section or `egress` `to` section:

podSelector: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.

namespaceSelector: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations.

namespaceSelector *and* **podSelector**: A single `to` / `from` entry that specifies both `namespaceSelector` and `podSelector` selects particular Pods within particular namespaces. Be careful to use correct YAML syntax. For example:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
    podSelector:
      matchLabels:
        role: client
...
```

This policy contains a single `from` element allowing connections from Pods with the label `role=client` in namespaces with the label `user=alice`. But the following policy is different:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
  - podSelector:
      matchLabels:
        role: client
...
```

It contains two elements in the `from` array, and allows connections from Pods in the local Namespace with the label `role=client`, *or* from any Pod in any namespace with the label `user=alice`.

When in doubt, use `kubectl describe` to see how Kubernetes has interpreted the policy.

ipBlock: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. These should be cluster-external IPs, since Pod IPs are ephemeral and unpredictable.

Cluster ingress and egress mechanisms often require rewriting the source or destination IP of packets. In cases where this happens, it is not defined whether this happens before or after NetworkPolicy processing, and the behavior may be different for different combinations of network plugin, cloud provider, `Service` implementation, etc.

In the case of ingress, this means that in some cases you may be able to filter incoming packets based on the actual original source IP, while in other cases, the "source IP" that the NetworkPolicy acts on may be the IP of a `LoadBalancer` or of the Pod's node, etc.

For egress, this means that connections from pods to `Service` IPs that get rewritten to cluster-external IPs may or may not be subject to `ipBlock` -based policies.

Default policies

By default, if no policies exist in a namespace, then all ingress and egress traffic is allowed to and from pods in that namespace. The following examples let you change the default behavior in that namespace.

Default deny all ingress traffic

You can create a "default" ingress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any ingress traffic to those pods.

service/networking/network-policy-default-deny-ingress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will still be isolated for ingress. This policy does not affect isolation for egress from any pod.

Allow all ingress traffic

If you want to allow all incoming connections to all pods in a namespace, you can create a policy that explicitly allows that.

service/networking/network-policy-allow-all-ingress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

With this policy in place, no additional policy or policies can cause any incoming connection to those pods to be denied. This policy has no effect on isolation for egress from any pod.

Default deny all egress traffic

You can create a "default" egress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any egress traffic from those pods.

service/networking/network-policy-default-deny-egress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed egress traffic. This policy does not change the ingress isolation behavior of any pod.

Allow all egress traffic

If you want to allow all connections from all pods in a namespace, you can create a policy that explicitly allows all outgoing connections from pods in that namespace.

service/networking/network-policy-allow-all-egress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-egress
spec:
  podSelector: {}
  egress:
  - {}
  policyTypes:
  - Egress
```

With this policy in place, no additional policy or policies can cause any outgoing connection from those pods to be denied. This policy has no effect on isolation for ingress to any pod.

Default deny all ingress and all egress traffic

You can create a "default" policy for a namespace which prevents all ingress AND egress traffic by creating the following NetworkPolicy in that namespace.

service/networking/network-policy-default-deny-all.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed ingress or egress traffic.

Network traffic filtering

NetworkPolicy is defined for layer 4 connections (TCP, UDP, and optionally SCTP). For all the other protocols, the behaviour may vary across network plugins.

> **Note:** You must be using a CNI plugin that supports SCTP protocol NetworkPolicies.

When a `deny all` network policy is defined, it is only guaranteed to deny TCP, UDP and SCTP connections. For other protocols, such as ARP or ICMP, the behaviour is undefined. The same applies to allow rules: when a specific pod is allowed as ingress source or egress destination, it is undefined what happens with (for example) ICMP packets. Protocols such as ICMP may be allowed by some network plugins and denied by others.

Targeting a range of ports

FEATURE STATE: `Kubernetes v1.25 [stable]`

When writing a NetworkPolicy, you can target a range of ports instead of a single port.

This is achievable with the usage of the `endPort` field, as the following example:

service/networking/networkpolicy-multiport-egress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: multi-port-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Egress
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
      ports:
        - protocol: TCP
          port: 32000
          endPort: 32768
```

The above rule allows any Pod with label `role=db` on the namespace `default` to communicate with any IP within the range `10.0.0.0/24` over TCP, provided that the target port is between the range 32000 and 32768.

The following restrictions apply when using this field:

- The `endPort` field must be equal to or greater than the `port` field.
- `endPort` can only be defined if `port` is also defined.
- Both ports must be numeric.

> **Note:** Your cluster must be using a CNI plugin that supports the `endPort` field in NetworkPolicy specifications. If your network plugin does not support the `endPort` field and you specify a NetworkPolicy with that, the policy will be applied only for the single `port` field.

Targeting multiple namespaces by label

In this scenario, your `Egress` NetworkPolicy targets more than one namespace using their label names. For this to work, you need to label the target namespaces. For example:

```
kubectl label namespace frontend namespace=frontend
kubectl label namespace backend namespace=backend
```

Add the labels under `namespaceSelector` in your NetworkPolicy document. For example:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-namespaces
spec:
  podSelector:
    matchLabels:
      app: myapp
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchExpressions:
        - key: namespace
          operator: In
          values: ["frontend", "backend"]
```

> **Note:** It is not possible to directly specify the name of the namespaces in a NetworkPolicy. You must use a `namespaceSelector` with `matchLabels` or `matchExpressions` to select the namespaces based on their labels.

Targeting a Namespace by its name

The Kubernetes control plane sets an immutable label `kubernetes.io/metadata.name` on all namespaces, the value of the label is the namespace name.

While NetworkPolicy cannot target a namespace by its name with some object field, you can use the standardized label to target a specific namespace.

Pod lifecycle

> **Note:** The following applies to clusters with a conformant networking plugin and a conformant implementation of NetworkPolicy.

When a new NetworkPolicy object is created, it may take some time for a network plugin to handle the new object. If a pod that is affected by a NetworkPolicy is created before the network plugin has completed NetworkPolicy handling, that pod may be started unprotected, and isolation rules will be applied when the NetworkPolicy handling is completed.

Once the NetworkPolicy is handled by a network plugin,

1. All newly created pods affected by a given NetworkPolicy will be isolated before they are started. Implementations of NetworkPolicy must ensure that filtering is effective throughout the Pod lifecycle, even from the very first instant that any container in that Pod is started. Because they are applied at Pod level, NetworkPolicies apply equally to init containers, sidecar containers, and regular containers.

2. Allow rules will be applied eventually after the isolation rules (or may be applied at the same time). In the worst case, a newly created pod may have no network connectivity at all when it is first started, if isolation rules were already applied, but no allow rules were applied yet.

Every created NetworkPolicy will be handled by a network plugin eventually, but there is no way to tell from the Kubernetes API when exactly that happens.

Therefore, pods must be resilient against being started up with different network connectivity than expected. If you need to make sure the pod can reach certain destinations before being started, you can use an init container to wait for those destinations to be reachable before kubelet starts the app containers.

Every NetworkPolicy will be applied to all selected pods eventually. Because the network plugin may implement NetworkPolicy in a distributed manner, it is possible that pods may see a slightly inconsistent view of network policies when the pod is first created, or when pods or policies change. For example, a newly-created pod that is supposed to be able to reach both Pod A on Node 1 and Pod B on Node 2 may find that it can reach Pod A immediately, but cannot reach Pod B until a few seconds later.

NetworkPolicy and `hostNetwork` pods

NetworkPolicy behaviour for `hostNetwork` pods is undefined, but it should be limited to 2 possibilities:

- The network plugin can distinguish `hostNetwork` pod traffic from all other traffic (including being able to distinguish traffic from different `hostNetwork` pods on the same node), and will apply NetworkPolicy to `hostNetwork` pods just like it does to pod-network pods.
- The network plugin cannot properly distinguish `hostNetwork` pod traffic, and so it ignores `hostNetwork` pods when matching `podSelector` and `namespaceSelector`. Traffic to/from `hostNetwork` pods is treated the same as all other traffic to/from the node IP. (This is the most common implementation.)

This applies when

1. a `hostNetwork` pod is selected by `spec.podSelector`.

```
...
spec:
  podSelector:
    matchLabels:
      role: client
...
```

2. a `hostNetwork` pod is selected by a `podSelector` or `namespaceSelector` in an `ingress` or `egress` rule.

```
...
ingress:
  - from:
    - podSelector:
        matchLabels:
          role: client
...
```

At the same time, since `hostNetwork` pods have the same IP addresses as the nodes they reside on, their connections will be treated as node connections. For example, you can allow traffic from a `hostNetwork` Pod using an `ipBlock` rule.

What you can't do with network policies (at least, not yet)

As of Kubernetes 1.29, the following functionality does not exist in the NetworkPolicy API, but you might be able to implement workarounds using Operating System components (such as SELinux, OpenVSwitch, IPTables, and so on) or Layer 7 technologies (Ingress controllers, Service Mesh implementations) or admission controllers. In case you are new to network security in Kubernetes, its worth noting that the following User Stories cannot (yet) be implemented using the NetworkPolicy API.

- Forcing internal cluster traffic to go through a common gateway (this might be best served with a service mesh or other proxy).
- Anything TLS related (use a service mesh or ingress controller for this).
- Node specific policies (you can use CIDR notation for these, but you cannot target nodes by their Kubernetes identities specifically).
- Targeting of services by name (you can, however, target pods or namespaces by their labels, which is often a viable workaround).
- Creation or management of "Policy requests" that are fulfilled by a third party.
- Default policies which are applied to all namespaces or pods (there are some third party Kubernetes distributions and projects which can do this).
- Advanced policy querying and reachability tooling.
- The ability to log network security events (for example connections that are blocked or accepted).
- The ability to explicitly deny policies (currently the model for NetworkPolicies are deny by default, with only the ability to add allow rules).
- The ability to prevent loopback or incoming host traffic (Pods cannot currently block localhost access, nor do they have the ability to block access from their resident node).

NetworkPolicy's impact on existing connections

When the set of NetworkPolicies that applies to an existing connection changes - this could happen either due to a change in NetworkPolicies or if the relevant labels of the namespaces/pods selected by the policy (both subject and peers) are changed in the middle of an existing connection - it is implementation defined as to whether the change will take effect for that existing connection or not. Example: A policy is created that leads to denying a previously allowed connection, the underlying network plugin implementation is responsible for defining if that new policy will close the existing connections or not. It is recommended not to modify policies/pods/namespaces in ways that might affect existing connections.

What's next

- See the Declare Network Policy walkthrough for further examples.
- See more recipes for common scenarios enabled by the NetworkPolicy resource.

7 - DNS for Services and Pods

Your workload can discover Services within your cluster using DNS; this page explains how that works.

Kubernetes creates DNS records for Services and Pods. You can contact Services with consistent DNS names instead of IP addresses.

Kubernetes publishes information about Pods and Services which is used to program DNS. Kubelet configures Pods' DNS so that running containers can lookup Services by name rather than IP.

Services defined in the cluster are assigned DNS names. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain.

Namespaces of Services

A DNS query may return different results based on the namespace of the Pod making it. DNS queries that don't specify a namespace are limited to the Pod's namespace. Access Services in other namespaces by specifying it in the DNS query.

For example, consider a Pod in a `test` namespace. A `data` Service is in the `prod` namespace.

A query for `data` returns no results, because it uses the Pod's `test` namespace.

A query for `data.prod` returns the intended result, because it specifies the namespace.

DNS queries may be expanded using the Pod's `/etc/resolv.conf`. Kubelet configures this file for each Pod. For example, a query for just `data` may be expanded to `data.test.svc.cluster.local`. The values of the `search` option are used to expand queries. To learn more about DNS queries, see the `resolv.conf` manual page.

```
nameserver 10.32.0.10
search <namespace>.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

In summary, a Pod in the *test* namespace can successfully resolve either `data.prod` or `data.prod.svc.cluster.local`.

DNS Records

What objects get DNS records?

1. Services
2. Pods

The following sections detail the supported DNS record types and layout that is supported. Any other layout or names or queries that happen to work are considered implementation details and are subject to change without warning. For more up-to-date specification, see Kubernetes DNS-Based Service Discovery.

Services

A/AAAA records

"Normal" (not headless) Services are assigned DNS A and/or AAAA records, depending on the IP family or families of the Service, with a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. This resolves to the cluster IP of the Service.

Headless Services (without a cluster IP) Services are also assigned DNS A and/or AAAA records, with a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. Unlike normal Services, this resolves to the set of IPs of all of the Pods selected by the Service. Clients are expected to consume the set or else use standard round-robin selection from the set.

SRV records

SRV Records are created for named ports that are part of normal or headless services. For each named port, the SRV record has the form `_port-name._port-protocol.my-svc.my-namespace.svc.cluster-domain.example`. For a regular Service, this resolves to the port number and the domain name: `my-svc.my-namespace.svc.cluster-domain.example`. For a headless Service, this resolves to multiple answers, one for each Pod that is backing the Service, and contains the port number and the domain name of the Pod of the form `hostname.my-svc.my-namespace.svc.cluster-domain.example`.

Pods

A/AAAA records

Kube-DNS versions, prior to the implementation of the DNS specification, had the following DNS resolution:

`pod-ipv4-address.my-namespace.pod.cluster-domain.example` .

For example, if a Pod in the `default` namespace has the IP address 172.17.0.3, and the domain name for your cluster is `cluster.local` , then the Pod has a DNS name:

`172-17-0-3.default.pod.cluster.local` .

Any Pods exposed by a Service have the following DNS resolution available:

`pod-ipv4-address.service-name.my-namespace.svc.cluster-domain.example` .

Pod's hostname and subdomain fields

Currently when a Pod is created, its hostname (as observed from within the Pod) is the Pod's `metadata.name` value.

The Pod spec has an optional `hostname` field, which can be used to specify a different hostname. When specified, it takes precedence over the Pod's name to be the hostname of the Pod (again, as observed from within the Pod). For example, given a Pod with `spec.hostname` set to `"my-host"` , the Pod will have its hostname set to `"my-host"` .

The Pod spec also has an optional `subdomain` field which can be used to indicate that the pod is part of sub-group of the namespace. For example, a Pod with `spec.hostname` set to `"foo"` , and `spec.subdomain` set to `"bar"` , in namespace `"my-namespace"` , will have its hostname set to `"foo"` and its fully qualified domain name (FQDN) set to `"foo.bar.my-namespace.svc.cluster.local"` (once more, as observed from within the Pod).

If there exists a headless Service in the same namespace as the Pod, with the same name as the subdomain, the cluster's DNS Server also returns A and/or AAAA records for the Pod's fully qualified hostname.

Example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: busybox-subdomain
spec:
  selector:
    name: busybox
  clusterIP: None
  ports:
  - name: foo # name is not required for single-port Services
    port: 1234
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostname: busybox-1
  subdomain: busybox-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: busybox-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
```

Given the above Service `"busybox-subdomain"` and the Pods which set `spec.subdomain` to `"busybox-subdomain"`, the first Pod will see its own FQDN as `"busybox-1.busybox-subdomain.my-namespace.svc.cluster-domain.example"`. DNS serves A and/or AAAA records at that name, pointing to the Pod's IP. Both Pods " `busybox1` " and " `busybox2` " will have their own address records.

An EndpointSlice can specify the DNS hostname for any endpoint addresses, along with its IP.

> **Note:** Because A and AAAA records are not created for Pod names, `hostname` is required for the Pod's A or AAAA record to be created. A Pod with no `hostname` but with `subdomain` will only create the A or AAAA record for the headless Service (`busybox-subdomain.my-namespace.svc.cluster-domain.example`), pointing to the Pods' IP addresses. Also, the Pod needs to be ready in order to have a record unless `publishNotReadyAddresses=True` is set on the Service.

Pod's setHostnameAsFQDN field

FEATURE STATE: `Kubernetes v1.22 [stable]`

When a Pod is configured to have fully qualified domain name (FQDN), its hostname is the short hostname. For example, if you have a Pod with the fully qualified domain name `busybox-1.busybox-subdomain.my-namespace.svc.cluster-domain.example`, then by default the `hostname` command inside that Pod returns `busybox-1` and the `hostname --fqdn` command returns the FQDN.

When you set `setHostnameAsFQDN: true` in the Pod spec, the kubelet writes the Pod's FQDN into the hostname for that Pod's namespace. In this case, both `hostname` and `hostname --fqdn` return the Pod's FQDN.

> **Note:**
> In Linux, the hostname field of the kernel (the `nodename` field of `struct utsname`) is limited to 64 characters.
>
> If a Pod enables this feature and its FQDN is longer than 64 character, it will fail to start. The Pod will remain in `Pending` status (`ContainerCreating` as seen by `kubectl`) generating error events, such as Failed to construct FQDN from Pod hostname and cluster domain, FQDN `long-FQDN` is too long (64 characters is the max, 70 characters requested). One way of improving user experience for this scenario is to create an admission webhook controller to control FQDN size when users create top level objects, for example, Deployment.

Pod's DNS Policy

DNS policies can be set on a per-Pod basis. Currently Kubernetes supports the following Pod-specific DNS policies. These policies are specified in the `dnsPolicy` field of a Pod Spec.

- " `Default` ": The Pod inherits the name resolution configuration from the node that the Pods run on. See related discussion for more details.
- " `ClusterFirst` ": Any DNS query that does not match the configured cluster domain suffix, such as " `www.kubernetes.io` ", is forwarded to an upstream nameserver by the DNS server. Cluster administrators may have extra stub-domain and upstream DNS servers configured. See related discussion for details on how DNS queries are handled in those cases.
- " `ClusterFirstWithHostNet` ": For Pods running with hostNetwork, you should explicitly set its DNS policy to " `ClusterFirstWithHostNet` ". Otherwise, Pods running with hostNetwork and `"ClusterFirst"` will fallback to the behavior of the `"Default"` policy.
  - Note: This is not supported on Windows. See below for details
- " `None` ": It allows a Pod to ignore DNS settings from the Kubernetes environment. All DNS settings are supposed to be provided using the `dnsConfig` field in the Pod Spec. See Pod's DNS config subsection below.

> **Note:** "Default" is not the default DNS policy. If `dnsPolicy` is not explicitly specified, then "ClusterFirst" is used.

The example below shows a Pod with its DNS policy set to " `ClusterFirstWithHostNet` " because it has `hostNetwork` set to `true` .

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

Pod's DNS Config

FEATURE STATE: `Kubernetes v1.14 [stable]`

Pod's DNS Config allows users more control on the DNS settings for a Pod.

The `dnsConfig` field is optional and it can work with any `dnsPolicy` settings. However, when a Pod's `dnsPolicy` is set to " `None` ", the `dnsConfig` field has to be specified.

Below are the properties a user can specify in the `dnsConfig` field:

- `nameservers` : a list of IP addresses that will be used as DNS servers for the Pod. There can be at most 3 IP addresses specified. When the Pod's `dnsPolicy` is set to "`None`", the list must contain at least one IP address, otherwise this property is optional. The servers listed will be combined to the base nameservers generated from the specified DNS policy with duplicate addresses removed.
- `searches` : a list of DNS search domains for hostname lookup in the Pod. This property is optional. When specified, the provided list will be merged into the base search domain names generated from the chosen DNS policy. Duplicate domain names are removed. Kubernetes allows up to 32 search domains.
- `options` : an optional list of objects where each object may have a `name` property (required) and a `value` property (optional). The contents in this property will be merged to the options generated from the specified DNS policy. Duplicate entries are removed.

The following is an example Pod with custom DNS settings:

service/networking/custom-dns.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 192.0.2.1 # this is an example
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

When the Pod above is created, the container `test` gets the following contents in its `/etc/resolv.conf` file:

```
nameserver 192.0.2.1
search ns1.svc.cluster-domain.example my.dns.search.suffix
options ndots:2 edns0
```

For IPv6 setup, search path and name server should be set up like this:

```
kubectl exec -it dns-example -- cat /etc/resolv.conf
```

The output is similar to this:

```
nameserver 2001:db8:30::a
search default.svc.cluster-domain.example svc.cluster-domain.example cluster-domain.example
options ndots:5
```

DNS search domain list limits

FEATURE STATE: `Kubernetes 1.28 [stable]`

Kubernetes itself does not limit the DNS Config until the length of the search domain list exceeds 32 or the total length of all search domains exceeds 2048. This limit applies to the node's resolver configuration file, the Pod's DNS Config, and the merged DNS Config respectively.

> **Note:**
> Some container runtimes of earlier versions may have their own restrictions on the number of DNS search domains. Depending on the container runtime environment, the pods with a large number of DNS search domains may get stuck in the pending state.
>
> It is known that containerd v1.5.5 or earlier and CRI-O v1.21 or earlier have this problem.

DNS resolution on Windows nodes

- ClusterFirstWithHostNet is not supported for Pods that run on Windows nodes. Windows treats all names with a `.` as a FQDN and skips FQDN resolution.
- On Windows, there are multiple DNS resolvers that can be used. As these come with slightly different behaviors, using the `Resolve-DNSName` powershell cmdlet for name query resolutions is recommended.
- On Linux, you have a DNS suffix list, which is used after resolution of a name as fully qualified has failed. On Windows, you can only have 1 DNS suffix, which is the DNS suffix associated with that Pod's namespace (example: `mydns.svc.cluster.local`). Windows can resolve FQDNs, Services, or network name which can be resolved with this single suffix. For example, a Pod spawned in the `default` namespace, will have the DNS suffix `default.svc.cluster.local`. Inside a Windows Pod, you can resolve both `kubernetes.default.svc.cluster.local` and `kubernetes`, but not the partially qualified names (`kubernetes.default` or `kubernetes.default.svc`).

What's next

For guidance on administering DNS configurations, check Configure DNS Service

8 - IPv4/IPv6 dual-stack

Kubernetes lets you configure single-stack IPv4 networking, single-stack IPv6 networking, or dual stack networking with both network families active. This page explains how.

FEATURE STATE: `Kubernetes v1.23 [stable]`

IPv4/IPv6 dual-stack networking enables the allocation of both IPv4 and IPv6 addresses to Pods and Services.

IPv4/IPv6 dual-stack networking is enabled by default for your Kubernetes cluster starting in 1.21, allowing the simultaneous assignment of both IPv4 and IPv6 addresses.

Supported Features

IPv4/IPv6 dual-stack on your Kubernetes cluster provides the following features:

- Dual-stack Pod networking (a single IPv4 and IPv6 address assignment per Pod)
- IPv4 and IPv6 enabled Services
- Pod off-cluster egress routing (eg. the Internet) via both IPv4 and IPv6 interfaces

Prerequisites

The following prerequisites are needed in order to utilize IPv4/IPv6 dual-stack Kubernetes clusters:

- Kubernetes 1.20 or later

  For information about using dual-stack services with earlier Kubernetes versions, refer to the documentation for that version of Kubernetes.

- 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.

Configure IPv4/IPv6 dual-stack

To configure IPv4/IPv6 dual-stack, set dual-stack cluster network assignments:

- kube-apiserver:
  - `--service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR>`
- kube-controller-manager:
  - `--cluster-cidr=<IPv4 CIDR>,<IPv6 CIDR>`
  - `--service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR>`
  - `--node-cidr-mask-size-ipv4|--node-cidr-mask-size-ipv6` defaults to /24 for IPv4 and /64 for IPv6
- kube-proxy:
  - `--cluster-cidr=<IPv4 CIDR>,<IPv6 CIDR>`
- kubelet:
  - `--node-ip=<IPv4 IP>,<IPv6 IP>`
    - This option is required for bare metal dual-stack nodes (nodes that do not define a cloud provider with the `--cloud-provider` flag). If you are using a cloud provider and choose to override the node IPs chosen by the cloud provider, set the `--node-ip` option.
    - (The legacy built-in cloud providers do not support dual-stack `--node-ip`.)

> **Note:**
> An example of an IPv4 CIDR: `10.244.0.0/16` (though you would supply your own address range)
>
> An example of an IPv6 CIDR: `fdXY:IJKL:MNOP:15::/64` (this shows the format but is not a valid address - see RFC 4193)

Services

You can create Services which can use IPv4, IPv6, or both.

The address family of a Service defaults to the address family of the first service cluster IP range (configured via the `--service-cluster-ip-range` flag to the kube-apiserver).

When you define a Service you can optionally configure it as dual stack. To specify the behavior you want, you set the `.spec.ipFamilyPolicy` field to one of the following values:

- `SingleStack`: Single-stack service. The control plane allocates a cluster IP for the Service, using the first configured service cluster IP range.
- `PreferDualStack`:
  - Allocates IPv4 and IPv6 cluster IPs for the Service.
- `RequireDualStack`: Allocates Service `.spec.ClusterIPs` from both IPv4 and IPv6 address ranges.
  - Selects the `.spec.ClusterIP` from the list of `.spec.ClusterIPs` based on the address family of the first element in the `.spec.ipFamilies` array.

If you would like to define which IP family to use for single stack or define the order of IP families for dual-stack, you can choose the address families by setting an optional field, `.spec.ipFamilies`, on the Service.

> **Note:** The `.spec.ipFamilies` field is conditionally mutable: you can add or remove a secondary IP address family, but you cannot change the primary IP address family of an existing Service.

You can set `.spec.ipFamilies` to any of the following array values:

- `["IPv4"]`
- `["IPv6"]`
- `["IPv4","IPv6"]` (dual stack)
- `["IPv6","IPv4"]` (dual stack)

The first family you list is used for the legacy `.spec.ClusterIP` field.

Dual-stack Service configuration scenarios

These examples demonstrate the behavior of various dual-stack Service configuration scenarios.

Dual-stack options on new Services

1. This Service specification does not explicitly define `.spec.ipFamilyPolicy`. When you create this Service, Kubernetes assigns a cluster IP for the Service from the first configured `service-cluster-ip-range` and sets the `.spec.ipFamilyPolicy` to `SingleStack`. (Services without selectors and headless Services with selectors will behave in this same way.)

service/networking/dual-stack-default-svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
```

2. This Service specification explicitly defines `PreferDualStack` in `.spec.ipFamilyPolicy`. When you create this Service on a dual-stack cluster, Kubernetes assigns both IPv4 and IPv6 addresses for the service. The control plane updates the `.spec` for the Service to record the IP address assignments. The field `.spec.ClusterIPs` is the primary field, and contains both assigned IP addresses; `.spec.ClusterIP` is a secondary field with its value calculated from `.spec.ClusterIPs`.

    - For the `.spec.ClusterIP` field, the control plane records the IP address that is from the same address family as the first service cluster IP range.
    - On a single-stack cluster, the `.spec.ClusterIPs` and `.spec.ClusterIP` fields both only list one address.
    - On a cluster with dual-stack enabled, specifying `RequireDualStack` in `.spec.ipFamilyPolicy` behaves the same as `PreferDualStack`.

service/networking/dual-stack-preferred-svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
```

3. This Service specification explicitly defines `IPv6` and `IPv4` in `.spec.ipFamilies` as well as defining `PreferDualStack` in `.spec.ipFamilyPolicy`. When Kubernetes assigns an IPv6 and IPv4 address in `.spec.ClusterIPs`, `.spec.ClusterIP` is set to the IPv6 address because that is the first element in the `.spec.ClusterIPs` array, overriding the default.

service/networking/dual-stack-preferred-ipfamilies-svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  ipFamilies:
  - IPv6
  - IPv4
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
```

Dual-stack defaults on existing Services

These examples demonstrate the default behavior when dual-stack is newly enabled on a cluster where Services already exist. (Upgrading an existing cluster to 1.21 or beyond will enable dual-stack.)

1. When dual-stack is enabled on a cluster, existing Services (whether `IPv4` or `IPv6`) are configured by the control plane to set `.spec.ipFamilyPolicy` to `SingleStack` and set `.spec.ipFamilies` to the address family of the existing Service. The existing Service cluster IP will be stored in `.spec.ClusterIPs`.

service/networking/dual-stack-default-svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
```

You can validate this behavior by using kubectl to inspect an existing service.

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

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: MyApp
  name: my-service
spec:
  clusterIP: 10.0.197.123
  clusterIPs:
  - 10.0.197.123
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app.kubernetes.io/name: MyApp
  type: ClusterIP
status:
  loadBalancer: {}
```

2. When dual-stack is enabled on a cluster, existing [headless Services](#) with selectors are configured by the control plane to set `.spec.ipFamilyPolicy` to `SingleStack` and set `.spec.ipFamilies` to the address family of the first service cluster IP range (configured via the `--service-cluster-ip-range` flag to the kube-apiserver) even though `.spec.ClusterIP` is set to `None`.

service/networking/dual-stack-default-svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
```

You can validate this behavior by using kubectl to inspect an existing headless service with selectors.

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

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: MyApp
  name: my-service
spec:
  clusterIP: None
  clusterIPs:
  - None
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app.kubernetes.io/name: MyApp
```

Switching Services between single-stack and dual-stack

Services can be changed from single-stack to dual-stack and from dual-stack to single-stack.

1. To change a Service from single-stack to dual-stack, change `.spec.ipFamilyPolicy` from `SingleStack` to `PreferDualStack` or `RequireDualStack` as desired. When you change this Service from single-stack to dual-stack, Kubernetes assigns the missing address family so that the Service now has IPv4 and IPv6 addresses.

   Edit the Service specification updating the `.spec.ipFamilyPolicy` from `SingleStack` to `PreferDualStack`.

   Before:

   ```yaml
   spec:
     ipFamilyPolicy: SingleStack
   ```

   After:

```
spec:
  ipFamilyPolicy: PreferDualStack
```

2. To change a Service from dual-stack to single-stack, change `.spec.ipFamilyPolicy` from `PreferDualStack` or `RequireDualStack` to `SingleStack`. When you change this Service from dual-stack to single-stack, Kubernetes retains only the first element in the `.spec.ClusterIPs` array, and sets `.spec.ClusterIP` to that IP address and sets `.spec.ipFamilies` to the address family of `.spec.ClusterIPs`.

Headless Services without selector

For Headless Services without selectors and without `.spec.ipFamilyPolicy` explicitly set, the `.spec.ipFamilyPolicy` field defaults to `RequireDualStack`.

Service type LoadBalancer

To provision a dual-stack load balancer for your Service:

- Set the `.spec.type` field to `LoadBalancer`
- Set `.spec.ipFamilyPolicy` field to `PreferDualStack` or `RequireDualStack`

Note: To use a dual-stack `LoadBalancer` type Service, your cloud provider must support IPv4 and IPv6 load balancers.

Egress traffic

If you want to enable egress traffic in order to reach off-cluster destinations (eg. the public Internet) from a Pod that uses non-publicly routable IPv6 addresses, you need to enable the Pod to use a publicly routed IPv6 address via a mechanism such as transparent proxying or IP masquerading. The ip-masq-agent project supports IP masquerading on dual-stack clusters.

Note: Ensure your CNI provider supports IPv6.

Windows support

Kubernetes on Windows does not support single-stack "IPv6-only" networking. However, dual-stack IPv4/IPv6 networking for pods and nodes with single-family services is supported.

You can use IPv4/IPv6 dual-stack networking with `l2bridge` networks.

Note: Overlay (VXLAN) networks on Windows **do not** support dual-stack networking.

You can read more about the different network modes for Windows within the Networking on Windows topic.

What's next

- Validate IPv4/IPv6 dual-stack networking
- Enable dual-stack networking using kubeadm

9 - Topology Aware Routing

Topology Aware Routing provides a mechanism to help keep network traffic within the zone where it originated. Preferring same-zone traffic between Pods in your cluster can help with reliability, performance (network latency and throughput), or cost.

FEATURE STATE: `Kubernetes v1.23 [beta]`

> **Note:** Prior to Kubernetes 1.27, this feature was known as *Topology Aware Hints*.

Topology Aware Routing adjusts routing behavior to prefer keeping traffic in the zone it originated from. In some cases this can help reduce costs or improve network performance.

Motivation

Kubernetes clusters are increasingly deployed in multi-zone environments. *Topology Aware Routing* provides a mechanism to help keep traffic within the zone it originated from. When calculating the endpoints for a Service, the EndpointSlice controller considers the topology (region and zone) of each endpoint and populates the hints field to allocate it to a zone. Cluster components such as kube-proxy can then consume those hints, and use them to influence how the traffic is routed (favoring topologically closer endpoints).

Enabling Topology Aware Routing

> **Note:** Prior to Kubernetes 1.27, this behavior was controlled using the `service.kubernetes.io/topology-aware-hints` annotation.

You can enable Topology Aware Routing for a Service by setting the `service.kubernetes.io/topology-mode` annotation to `Auto`. When there are enough endpoints available in each zone, Topology Hints will be populated on EndpointSlices to allocate individual endpoints to specific zones, resulting in traffic being routed closer to where it originated from.

When it works best

This feature works best when:

1. Incoming traffic is evenly distributed

If a large proportion of traffic is originating from a single zone, that traffic could overload the subset of endpoints that have been allocated to that zone. This feature is not recommended when incoming traffic is expected to originate from a single zone.

2. The Service has 3 or more endpoints per zone

In a three zone cluster, this means 9 or more endpoints. If there are fewer than 3 endpoints per zone, there is a high (≈50%) probability that the EndpointSlice controller will not be able to allocate endpoints evenly and instead will fall back to the default cluster-wide routing approach.

How It Works

The "Auto" heuristic attempts to proportionally allocate a number of endpoints to each zone. Note that this heuristic works best for Services that have a significant number of endpoints.

EndpointSlice controller

The EndpointSlice controller is responsible for setting hints on EndpointSlices when this heuristic is enabled. The controller allocates a proportional amount of endpoints to each zone. This proportion is based on the allocatable CPU cores for nodes running in that zone. For example, if one zone had 2 CPU cores and another zone only had 1 CPU core, the controller would allocate twice as many endpoints to the zone with 2 CPU cores.

The following example shows what an EndpointSlice looks like when hints have been populated:

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: example-hints
  labels:
    kubernetes.io/service-name: example-svc
addressType: IPv4
ports:
  - name: http
    protocol: TCP
    port: 80
endpoints:
  - addresses:
      - "10.1.2.3"
    conditions:
      ready: true
    hostname: pod-1
    zone: zone-a
    hints:
      forZones:
        - name: "zone-a"
```

kube-proxy

The kube-proxy component filters the endpoints it routes to based on the hints set by the EndpointSlice controller. In most cases, this means that the kube-proxy is able to route traffic to endpoints in the same zone. Sometimes the controller allocates endpoints from a different zone to ensure more even distribution of endpoints between zones. This would result in some traffic being routed to other zones.

Safeguards

The Kubernetes control plane and the kube-proxy on each node apply some safeguard rules before using Topology Aware Hints. If these don't check out, the kube-proxy selects endpoints from anywhere in your cluster, regardless of the zone.

1. **Insufficient number of endpoints:** If there are less endpoints than zones in a cluster, the controller will not assign any hints.

2. **Impossible to achieve balanced allocation:** In some cases, it will be impossible to achieve a balanced allocation of endpoints among zones. For example, if zone-a is twice as large as zone-b, but there are only 2 endpoints, an endpoint allocated to zone-a may receive twice as much traffic as zone-b. The controller does not assign hints if it can't get this "expected overload" value below an acceptable threshold for each zone. Importantly this is not based on real-time feedback. It is still possible for individual endpoints to become overloaded.

3. **One or more Nodes has insufficient information:** If any node does not have a `topology.kubernetes.io/zone` label or is not reporting a value for allocatable CPU, the control plane does not set any topology-aware endpoint hints and so kube-proxy does not filter endpoints by zone.

4. **One or more endpoints does not have a zone hint:** When this happens, the kube-proxy assumes that a transition from or to Topology Aware Hints is underway. Filtering endpoints for a Service in this state would be dangerous so the kube-proxy falls back to using all endpoints.

5. **A zone is not represented in hints:** If the kube-proxy is unable to find at least one endpoint with a hint targeting the zone it is running in, it falls back to using endpoints from all zones. This is most likely to happen as you add a new zone into your existing cluster.

Constraints

- Topology Aware Hints are not used when `internalTrafficPolicy` is set to `Local` on a Service. It is possible to use both features in the same cluster on different Services, just not on the same Service.
- This approach will not work well for Services that have a large proportion of traffic originating from a subset of zones. Instead this assumes that incoming traffic will be roughly proportional to the capacity of the Nodes in each zone.
- The EndpointSlice controller ignores unready nodes as it calculates the proportions of each zone. This could have unintended consequences if a large portion of nodes are unready.
- The EndpointSlice controller ignores nodes with the `node-role.kubernetes.io/control-plane` or `node-role.kubernetes.io/master` label set. This could be problematic if workloads are also running on those nodes.
- The EndpointSlice controller does not take into account tolerations when deploying or calculating the proportions of each zone. If the Pods backing a Service are limited to a subset of Nodes in the cluster, this will not be taken into account.
- This may not work well with autoscaling. For example, if a lot of traffic is originating from a single zone, only the endpoints allocated to that zone will be handling that traffic. That could result in Horizontal Pod Autoscaler either not picking up on this event, or newly added pods starting in a different zone.

Custom heuristics

Kubernetes is deployed in many different ways, there is no single heuristic for allocating endpoints to zones will work for every use case. A key goal of this feature is to enable custom heuristics to be developed if the built in heuristic does not work for your use case. The first steps to enable custom heuristics were included in the 1.27 release. This is a limited implementation that may not yet cover some relevant and plausible situations.

What's next

- Follow the Connecting Applications with Services tutorial

10 - Networking on Windows

Kubernetes supports running nodes on either Linux or Windows. You can mix both kinds of node within a single cluster. This page provides an overview to networking specific to the Windows operating system.

Container networking on Windows

Networking for Windows containers is exposed through CNI plugins. Windows containers function similarly to virtual machines in regards to networking. Each container has a virtual network adapter (vNIC) which is connected to a Hyper-V virtual switch (vSwitch). The Host Networking Service (HNS) and the Host Compute Service (HCS) work together to create containers and attach container vNICs to networks. HCS is responsible for the management of containers whereas HNS is responsible for the management of networking resources such as:

- Virtual networks (including creation of vSwitches)
- Endpoints / vNICs
- Namespaces
- Policies including packet encapsulations, load-balancing rules, ACLs, and NAT rules.

The Windows HNS and vSwitch implement namespacing and can create virtual NICs as needed for a pod or container. However, many configurations such as DNS, routes, and metrics are stored in the Windows registry database rather than as files inside `/etc`, which is how Linux stores those configurations. The Windows registry for the container is separate from that of the host, so concepts like mapping `/etc/resolv.conf` from the host into a container don't have the same effect they would on Linux. These must be configured using Windows APIs run in the context of that container. Therefore CNI implementations need to call the HNS instead of relying on file mappings to pass network details into the pod or container.

Network modes

Windows supports five different networking drivers/modes: L2bridge, L2tunnel, Overlay (Beta), Transparent, and NAT. In a heterogeneous cluster with Windows and Linux worker nodes, you need to select a networking solution that is compatible on both Windows and Linux. The following table lists the out-of-tree plugins are supported on Windows, with recommendations on when to use each CNI:

Network Driver	Description	Container Packet Modifications	Network Plugins	Network Plugin Characteristics
L2bridge	Containers are attached to an external vSwitch. Containers are attached to the underlay network, although the physical network doesn't need to learn the container MACs because they are rewritten on ingress/egress.	MAC is rewritten to host MAC, IP may be rewritten to host IP using HNS OutboundNAT policy.	win-bridge, Azure-CNI, Flannel host-gateway uses win-bridge	win-bridge uses L2bridge network mode, connects containers to the underlay of hosts, offering best performance. Requires user-defined routes (UDR) for inter-node connectivity.
L2Tunnel	This is a special case of l2bridge, but only used on Azure. All packets are sent to the virtualization host where SDN policy is applied.	MAC rewritten, IP visible on the underlay network	Azure-CNI	Azure-CNI allows integration of containers with Azure vNET, and allows them to leverage the set of capabilities that Azure Virtual Network provides. For example, securely connect to Azure services or use Azure NSGs. See azure-cni for some examples

Network Driver	Description	Container Packet Modifications	Network Plugins	Network Plugin Characteristics
Overlay	Containers are given a vNIC connected to an external vSwitch. Each overlay network gets its own IP subnet, defined by a custom IP prefix.The overlay network driver uses VXLAN encapsulation.	Encapsulated with an outer header.	win-overlay, Flannel VXLAN (uses win-overlay)	win-overlay should be used when virtual container networks are desired to be isolated from underlay of hosts (e.g. for security reasons). Allows for IPs to be re-used for different overlay networks (which have different VNID tags) if you are restricted on IPs in your datacenter. This option requires KB4489899 on Windows Server 2019.
Transparent (special use case for ovn-kubernetes)	Requires an external vSwitch. Containers are attached to an external vSwitch which enables intra-pod communication via logical networks (logical switches and routers).	Packet is encapsulated either via GENEVE or STT tunneling to reach pods which are not on the same host. Packets are forwarded or dropped via the tunnel metadata information supplied by the ovn network controller. NAT is done for north-south communication.	ovn-kubernetes	Deploy via ansible. Distributed ACLs can be applied via Kubernetes policies. IPAM support. Load-balancing can be achieved without kube-proxy. NATing is done without using iptables/netsh.
NAT (*not used in Kubernetes*)	Containers are given a vNIC connected to an internal vSwitch. DNS/DHCP is provided using an internal component called WinNAT	MAC and IP is rewritten to host MAC/IP.	nat	Included here for completeness

As outlined above, the Flannel CNI plugin is also supported on Windows via the VXLAN network backend (**Beta support** ; delegates to win-overlay) and host-gateway network backend (stable support; delegates to win-bridge).

This plugin supports delegating to one of the reference CNI plugins (win-overlay, win-bridge), to work in conjunction with Flannel daemon on Windows (Flanneld) for automatic node subnet lease assignment and HNS network creation. This plugin reads in its own configuration file (cni.conf), and aggregates it with the environment variables from the FlannelD generated subnet.env file. It then delegates to one of the reference CNI plugins for network plumbing, and sends the correct configuration containing the node-assigned subnet to the IPAM plugin (for example: `host-local`).

For Node, Pod, and Service objects, the following network flows are supported for TCP/UDP traffic:

- Pod → Pod (IP)
- Pod → Pod (Name)
- Pod → Service (Cluster IP)
- Pod → Service (PQDN, but only if there are no ".")
- Pod → Service (FQDN)
- Pod → external (IP)
- Pod → external (DNS)
- Node → Pod
- Pod → Node

IP address management (IPAM)

The following IPAM options are supported on Windows:

- host-local
- azure-vnet-ipam (for azure-cni only)
- Windows Server IPAM (fallback option if no IPAM is set)

Load balancing and Services

A Kubernetes Service is an abstraction that defines a logical set of Pods and a means to access them over a network. In a cluster that includes Windows nodes, you can use the following types of Service:

- `NodePort`
- `ClusterIP`
- `LoadBalancer`
- `ExternalName`

Windows container networking differs in some important ways from Linux networking. The Microsoft documentation for Windows Container Networking provides additional details and background.

On Windows, you can use the following settings to configure Services and load balancing behavior:

Feature	Description	Minimum Supported Windows OS build	How to enable
Session affinity	Ensures that connections from a particular client are passed to the same Pod each time.	Windows Server 2022	Set `service.spec.sessionAffinity` to "ClientIP"
Direct Server Return (DSR)	Load balancing mode where the IP address fixups and the LBNAT occurs at the container vSwitch port directly; service traffic arrives with the source IP set as the originating pod IP.	Windows Server 2019	Set the following flags in kube-proxy: `--feature-gates="WinDSR=true" --enable-dsr=true`
Preserve-Destination	Skips DNAT of service traffic, thereby preserving the virtual IP of the target service in packets reaching the backend Pod. Also disables node-node forwarding.	Windows Server, version 1903	Set `"preserve-destination": "true"` in service annotations and enable DSR in kube-proxy.
IPv4/IPv6 dual-stack networking	Native IPv4-to-IPv4 in parallel with IPv6-to-IPv6 communications to, from, and within a cluster	Windows Server 2019	See IPv4/IPv6 dual-stack
Client IP preservation	Ensures that source IP of incoming ingress traffic gets preserved. Also disables node-node forwarding.	Windows Server 2019	Set `service.spec.externalTrafficPolicy` to "Local" and enable DSR in kube-proxy

Warning:
There are known issue with NodePort Services on overlay networking, if the destination node is running Windows Server 2022. To avoid the issue entirely, you can configure the service with `externalTrafficPolicy: Local`.

There are known issues with Pod to Pod connectivity on l2bridge network on Windows Server 2022 with KB5005619 or higher installed. To workaround the issue and restore Pod to Pod connectivity, you can disable the WinDSR feature in kube-proxy.

These issues require OS fixes. Please follow https://github.com/microsoft/Windows-Containers/issues/204 for updates.

Limitations

The following networking functionality is *not* supported on Windows nodes:

- Host networking mode
- Local NodePort access from the node itself (works for other nodes or external clients)
- More than 64 backend pods (or unique destination addresses) for a single Service
- IPv6 communication between Windows pods connected to overlay networks
- Local Traffic Policy in non-DSR mode
- Outbound communication using the ICMP protocol via the `win-overlay`, `win-bridge`, or using the Azure-CNI plugin. Specifically, the Windows data plane (VFP) doesn't support ICMP packet transpositions, and this means:
  - ICMP packets directed to destinations within the same network (such as pod to pod communication via ping) work as expected;
  - TCP/UDP packets work as expected;
  - ICMP packets directed to pass through a remote network (e.g. pod to external internet communication via ping) cannot be transposed and thus will not be routed back to their source;
  - Since TCP/UDP packets can still be transposed, you can substitute `ping <destination>` with `curl <destination>` when debugging connectivity with the outside world.

Other limitations:

- Windows reference network plugins win-bridge and win-overlay do not implement CNI spec v0.4.0, due to a missing `CHECK` implementation.
- The Flannel VXLAN CNI plugin has the following limitations on Windows:
  - Node-pod connectivity is only possible for local pods with Flannel v0.12.0 (or higher).
  - Flannel is restricted to using VNI 4096 and UDP port 4789. See the official Flannel VXLAN backend docs for more details on these parameters.

11 - Service ClusterIP allocation

In Kubernetes, Services are an abstract way to expose an application running on a set of Pods. Services can have a cluster-scoped virtual IP address (using a Service of `type: ClusterIP`). Clients can connect using that virtual IP address, and Kubernetes then load-balances traffic to that Service across the different backing Pods.

How Service ClusterIPs are allocated?

When Kubernetes needs to assign a virtual IP address for a Service, that assignment happens one of two ways:

dynamically

the cluster's control plane automatically picks a free IP address from within the configured IP range for `type: ClusterIP` Services.

statically

you specify an IP address of your choice, from within the configured IP range for Services.

Across your whole cluster, every Service `ClusterIP` must be unique. Trying to create a Service with a specific `ClusterIP` that has already been allocated will return an error.

Why do you need to reserve Service Cluster IPs?

Sometimes you may want to have Services running in well-known IP addresses, so other components and users in the cluster can use them.

The best example is the DNS Service for the cluster. As a soft convention, some Kubernetes installers assign the 10th IP address from the Service IP range to the DNS service. Assuming you configured your cluster with Service IP range 10.96.0.0/16 and you want your DNS Service IP to be 10.96.0.10, you'd have to create a Service like this:

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: CoreDNS
  name: kube-dns
  namespace: kube-system
spec:
  clusterIP: 10.96.0.10
  ports:
  - name: dns
    port: 53
    protocol: UDP
    targetPort: 53
  - name: dns-tcp
    port: 53
    protocol: TCP
    targetPort: 53
  selector:
    k8s-app: kube-dns
  type: ClusterIP
```

but as it was explained before, the IP address 10.96.0.10 has not been reserved; if other Services are created before or in parallel with dynamic allocation, there is a chance they can allocate this IP, hence, you will not be able to create the DNS Service because it will fail with a conflict error.

How can you avoid Service ClusterIP conflicts?

The allocation strategy implemented in Kubernetes to allocate ClusterIPs to Services reduces the risk of collision.

The `ClusterIP` range is divided, based on the formula `min(max(16, cidrSize / 16), 256)`, described as *never less than 16 or more than 256 with a graduated step between them*.

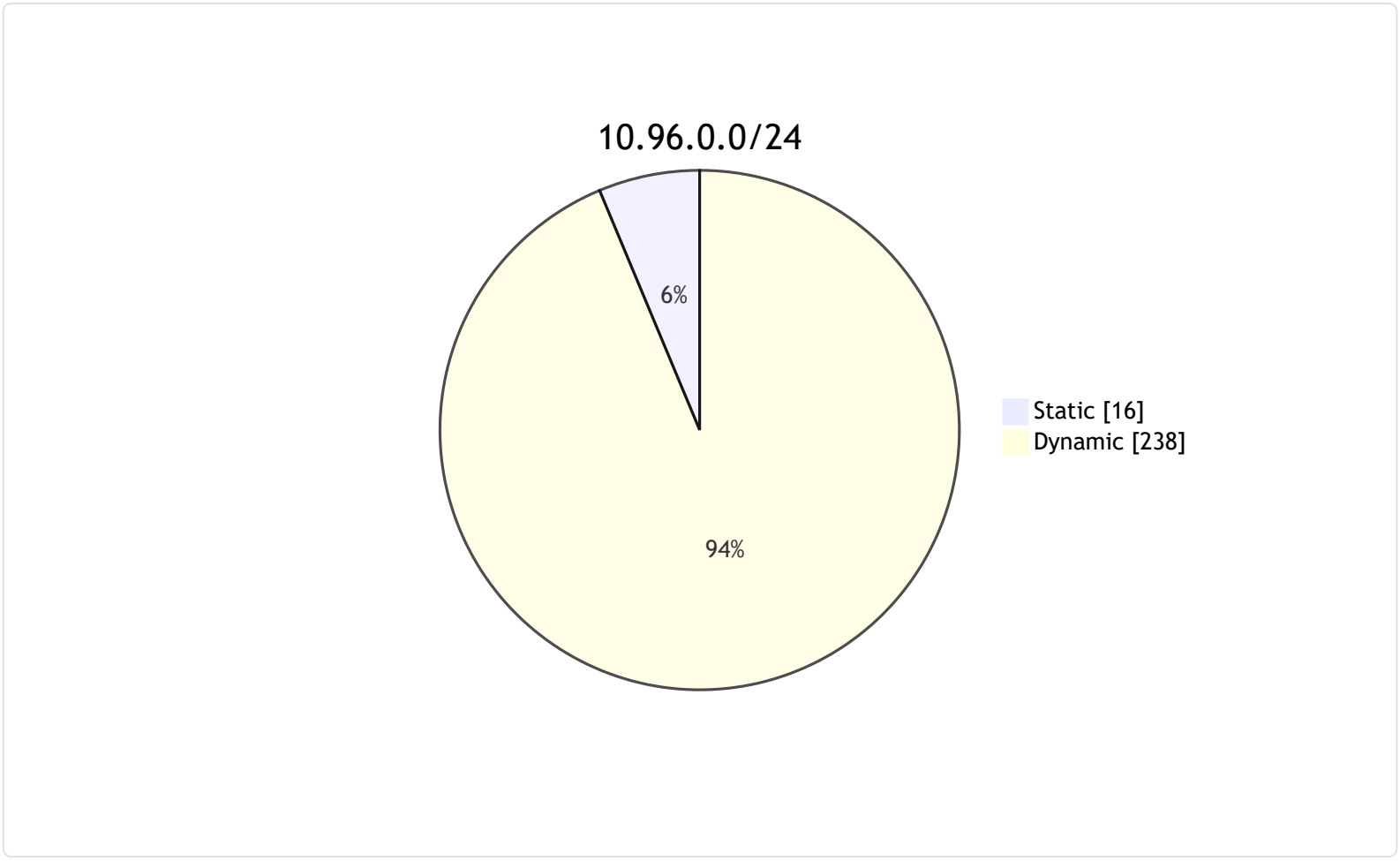
Dynamic IP assignment uses the upper band by default, once this has been exhausted it will use the lower range. This will allow users to use static allocations on the lower band with a low risk of collision.

Examples

Example 1

This example uses the IP address range: 10.96.0.0/24 (CIDR notation) for the IP addresses of Services.

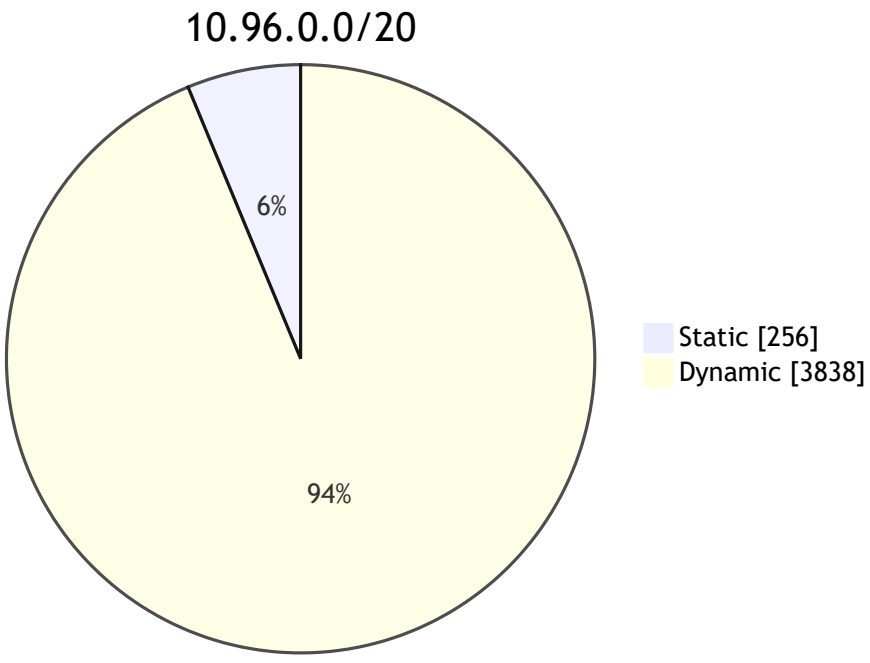
Range Size: 2^8 - 2 = 254
Band Offset: `min(max(16, 256/16), 256) = min(16, 256)` = 16
Static band start: 10.96.0.1
Static band end: 10.96.0.16
Range end: 10.96.0.254



Example 2

This example uses the IP address range: 10.96.0.0/20 (CIDR notation) for the IP addresses of Services.

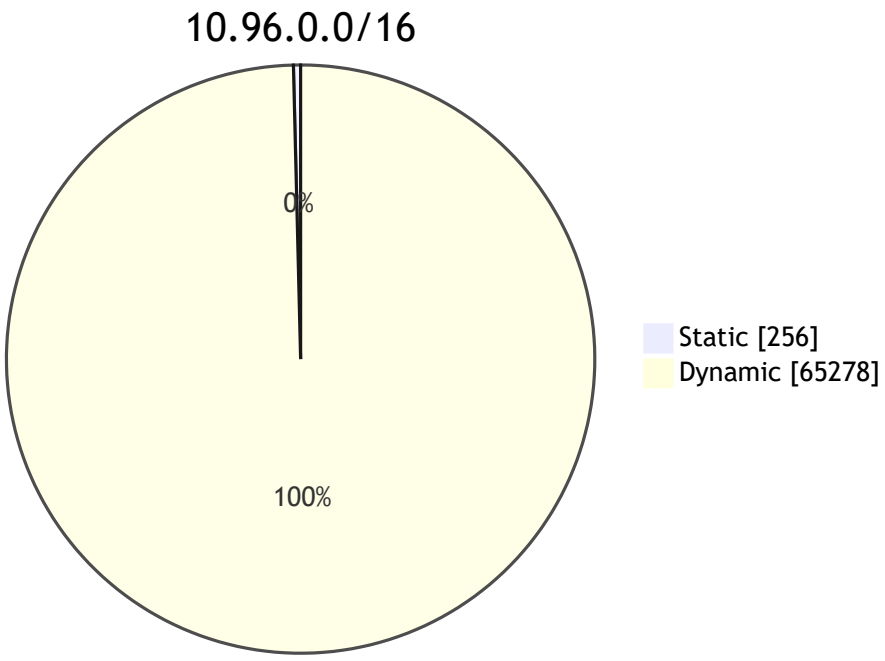
Range Size: 2^{12} - 2 = 4094
Band Offset: `min(max(16, 4096/16), 256) = min(256, 256)` = 256
Static band start: 10.96.0.1
Static band end: 10.96.1.0
Range end: 10.96.15.254



Example 3

This example uses the IP address range: 10.96.0.0/16 (CIDR notation) for the IP addresses of Services.

Range Size: 2^{16} - 2 = 65534
Band Offset: `min(max(16, 65536/16), 256) = min(4096, 256)` = 256
Static band start: 10.96.0.1
Static band ends: 10.96.1.0
Range end: 10.96.255.254



What's next

- Read about Service External Traffic Policy
- Read about Connecting Applications with Services
- Read about Services

12 - Service Internal Traffic Policy

If two Pods in your cluster want to communicate, and both Pods are actually running on the same node, use *Service Internal Traffic Policy* to keep network traffic within that node. Avoiding a round trip via the cluster network can help with reliability, performance (network latency and throughput), or cost.

FEATURE STATE: `Kubernetes v1.26 [stable]`

Service Internal Traffic Policy enables internal traffic restrictions to only route internal traffic to endpoints within the node the traffic originated from. The "internal" traffic here refers to traffic originated from Pods in the current cluster. This can help to reduce costs and improve performance.

Using Service Internal Traffic Policy

You can enable the internal-only traffic policy for a Service, by setting its `.spec.internalTrafficPolicy` to `Local`. This tells kube-proxy to only use node local endpoints for cluster internal traffic.

> **Note:** For pods on nodes with no endpoints for a given Service, the Service behaves as if it has zero endpoints (for Pods on this node) even if the service does have endpoints on other nodes.

The following example shows what a Service looks like when you set `.spec.internalTrafficPolicy` to `Local`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  internalTrafficPolicy: Local
```

How it works

The kube-proxy filters the endpoints it routes to based on the `spec.internalTrafficPolicy` setting. When it's set to `Local`, only node local endpoints are considered. When it's `Cluster` (the default), or is not set, Kubernetes considers all endpoints.

What's next

- Read about Topology Aware Routing
- Read about Service External Traffic Policy
- Follow the Connecting Applications with Services tutorial