Services, Load Balancing, and Networking

Concepts and resources behind networking in Kubernetes.

Kubernetes networking addresses four concerns:

- Containers within a Pod use networking to communicate via loopback.
- Cluster networking provides communication between different Pods.
- The Service resource lets you expose an application running in Pods to be reachable from outside your cluster.
- You can also use Services to publish services only for consumption inside your cluster.

1 - Service

An abstract way to expose an application running on a set of Pods as a network service.
With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

Motivation

Kubernetes Pods are created and destroyed to match the state of your cluster. Pods are nonpermanent resources. If you use a Deployment to run your app, it can create and destroy Pods dynamically.

Each Pod gets its own IP address, however in a Deployment, 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*.

Service resources

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a selector. To learn about other ways to define Service endpoints, see Services *without* selectors.

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.

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 Endpoints, that get updated 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.

Defining a Service

A Service in Kubernetes is a REST object, similar to a Pod. Like all of the REST objects, you can `POST` a Service definition to the API server to create a new instance. The name of a Service object must be a valid DNS label name.

For example, suppose you have a set of Pods where each listens on TCP port 9376 and contains a label `app=MyApp`:

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

This specification creates a new Service object named "my-service", which targets TCP port 9376 on any Pod with the `app=MyApp` label.

Kubernetes assigns this Service an IP address (sometimes called the "cluster IP"), which is used by the Service proxies (see Virtual IPs and service proxies below).

The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an Endpoint object also named "my-service".

> **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 in Pods have names, and you can reference these names in the `targetPort` attribute of a Service. 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.

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

Services without selectors

Services most commonly abstract access to Kubernetes Pods, but they can also abstract other kinds of backends. 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* a Pod selector. 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 Endpoints object is not created automatically. You can manually map the Service to the network address and port where it's running, by adding an Endpoints object manually:

```yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9376
```

The name of the Endpoints object must be a valid DNS subdomain name.

> **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).
>
> Endpoint IP addresses cannot be the cluster IPs of other Kubernetes Services, because kube-proxy doesn't support virtual IPs as a destination.

Accessing a Service without a selector works the same as if it had a selector. In the example above, traffic is routed to the single endpoint defined in the YAML: `192.0.2.42:9376` (TCP).

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 later in this document.

Over Capacity Endpoints

If an Endpoints resource has more than 1000 endpoints then a Kubernetes v1.21 (or later) cluster annotates that Endpoints with `endpoints.kubernetes.io/over-capacity: warning`. This annotation indicates that the affected Endpoints object is over capacity.

EndpointSlices

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

EndpointSlices are an API resource that can provide a more scalable alternative to Endpoints. Although conceptually quite similar to Endpoints, EndpointSlices allow for distributing network endpoints across multiple resources. By default, an EndpointSlice is considered "full" once it reaches 100 endpoints, at which point additional EndpointSlices will be created to store any additional endpoints.

EndpointSlices provide additional attributes and functionality which is described in detail in EndpointSlices.

Application protocol

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

The `appProtocol` field provides a way to specify an application protocol for each Service port. The value of this field is mirrored by the corresponding Endpoints and EndpointSlice objects.

This field follows standard Kubernetes label syntax. Values should either be IANA standard service names or domain prefixed names such as `mycompany.com/my-custom-protocol`.

Virtual IPs and service proxies

Every node in a Kubernetes cluster runs a `kube-proxy`. `kube-proxy` is responsible for implementing a form of virtual IP for `Services` of type other than `ExternalName`.

Why not use round-robin DNS?

A question that pops up every now and then is why Kubernetes relies on proxying to forward inbound traffic to backends. What about other approaches? For example, would it be possible to configure DNS records that have multiple A values (or AAAA for IPv6), and rely on round-robin name resolution?

There are a few reasons for using proxying for Services:

- There is a long history of DNS implementations not respecting record TTLs, and caching the results of name lookups after they should have expired.
- Some apps do DNS lookups only once and cache the results indefinitely.
- Even if apps and libraries did proper re-resolution, the low or zero TTLs on the DNS records could impose a high load on DNS that then becomes difficult to manage.

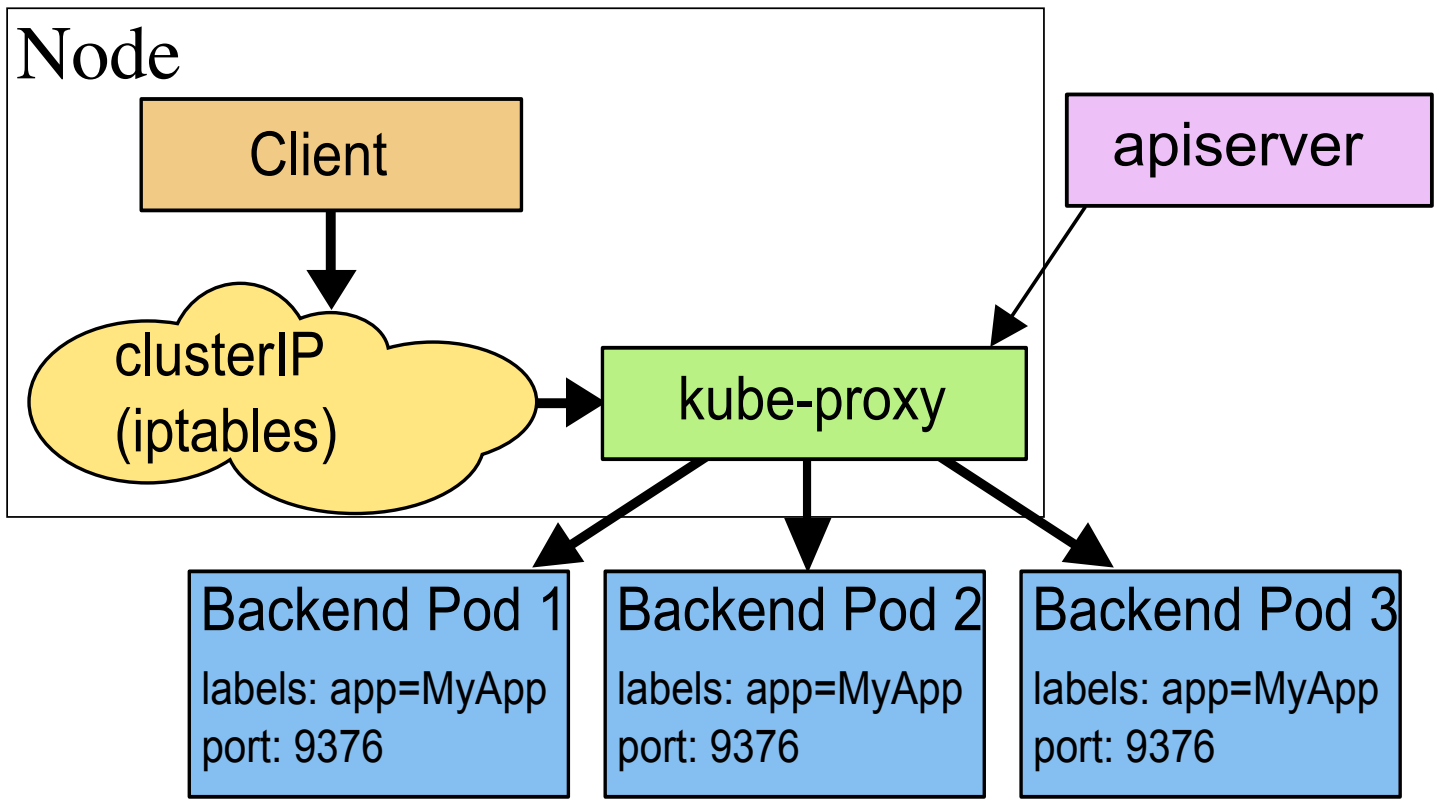
User space proxy mode

In this mode, kube-proxy watches the Kubernetes control plane for the addition and removal of Service and Endpoint objects. For each Service it opens a port (randomly chosen) on the local node. Any connections to this "proxy port" are proxied to one of the Service's backend Pods (as reported via

Endpoints). kube-proxy takes the `SessionAffinity` setting of the Service into account when deciding which backend Pod to use.

Lastly, the user-space proxy installs iptables rules which capture traffic to the Service's `clusterIP` (which is virtual) and `port`. The rules redirect that traffic to the proxy port which proxies the backend Pod.

By default, kube-proxy in userspace mode chooses a backend via a round-robin algorithm.



`iptables` proxy mode

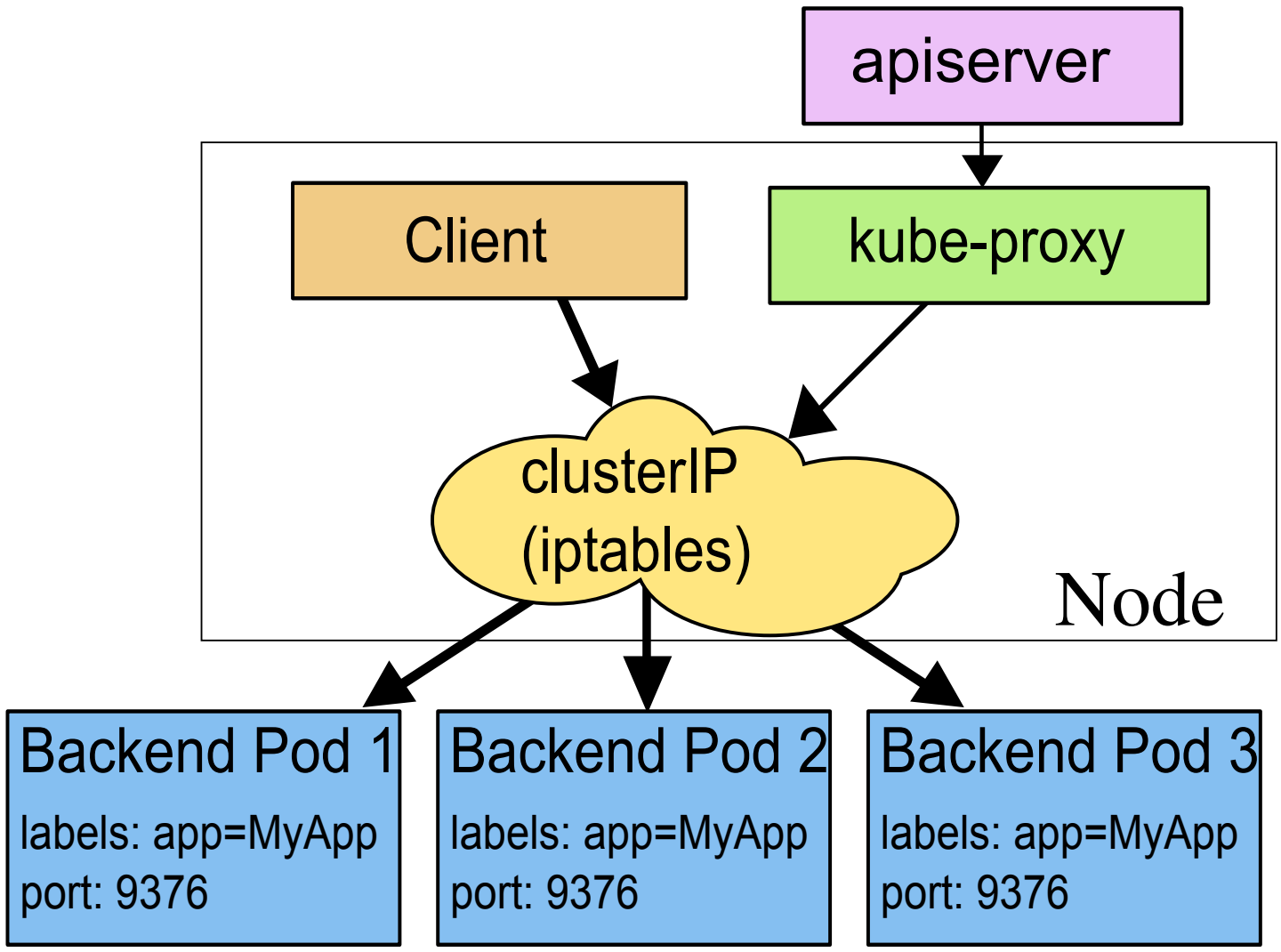
In this mode, kube-proxy watches the Kubernetes control plane for the addition and removal of Service and Endpoint objects. For each Service, it installs iptables rules, which capture traffic to the Service's `clusterIP` and `port`, and redirect that traffic to one of the Service's backend sets. For each Endpoint object, it installs iptables rules which select a backend Pod.

By default, kube-proxy in iptables mode chooses a backend at random.

Using iptables to handle traffic has a lower system overhead, because traffic is handled by Linux netfilter without the need to switch between userspace and the kernel space. This approach is also likely to be more reliable.

If kube-proxy is running in iptables mode and the first Pod that's selected does not respond, the connection fails. This is different from userspace mode: in that scenario, kube-proxy would detect that the connection to the first Pod had failed and would automatically retry with a different backend Pod.

You can use Pod readiness probes to verify that backend Pods are working OK, so that kube-proxy in iptables mode only sees backends that test out as healthy. Doing this means you avoid having traffic sent via kube-proxy to a Pod that's known to have failed.



IPVS proxy mode

FEATURE STATE: `Kubernetes v1.11 [stable]`

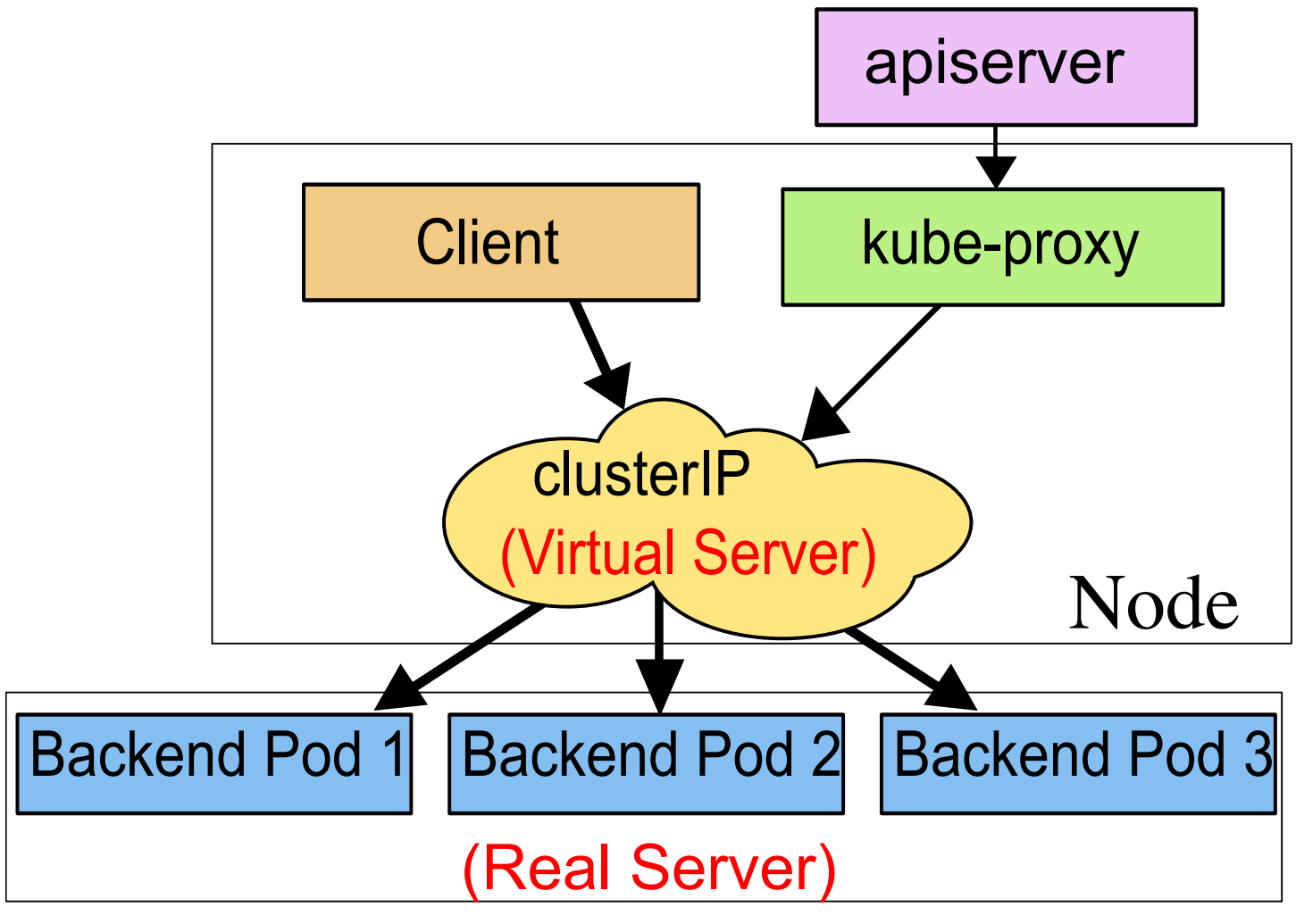
In `ipvs` mode, kube-proxy watches Kubernetes Services and Endpoints, calls `netlink` interface to create IPVS rules accordingly and synchronizes IPVS rules with Kubernetes Services and Endpoints periodically. This control loop ensures that IPVS status matches the desired state. When accessing a Service, IPVS directs traffic to one of the backend Pods.

The IPVS proxy mode is based on netfilter hook function that is similar to iptables mode, but uses a hash table as the underlying data structure and works in the kernel space. That means kube-proxy in IPVS mode redirects traffic with lower latency than kube-proxy in iptables mode, with much better performance when synchronising proxy rules. Compared to the other proxy modes, IPVS mode also supports a higher throughput of network traffic.

IPVS provides more options for balancing traffic to backend Pods; these are:

- `rr`: round-robin
- `lc`: least connection (smallest number of open connections)
- `dh`: destination hashing
- `sh`: source hashing
- `sed`: shortest expected delay
- `nq`: never queue

> **Note:**
> To run kube-proxy in IPVS mode, you must make IPVS available on the node before starting kube-proxy.
>
> When kube-proxy starts in IPVS proxy mode, it verifies whether IPVS kernel modules are available. If the IPVS kernel modules are not detected, then kube-proxy falls back to running in iptables proxy mode.



In these proxy models, the traffic bound for the Service's IP:Port is proxied to an appropriate backend without the clients knowing anything about Kubernetes or Services or Pods.

If you want to make sure that connections from a particular client are passed to the same Pod each time, you can select the session affinity based on the client's IP addresses by setting `service.spec.sessionAffinity` to "ClientIP" (the default is "None"). You can also set the maximum session sticky time by setting `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` appropriately. (the default value is 10800, which works out to be 3 hours).

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: 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.

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.

Discovering services

Kubernetes supports 2 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 supports both Docker links compatible variables (see makeLinkVariables) and simpler `{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-master` which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11, produces the following environment variables:

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_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.

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 Pods and Services.

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 (`.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 `Endpoints` records in the API, and modifies the DNS configuration to return A records (IP addresses) that point directly to the `Pods` backing the `Service` .

Without selectors

For headless Services that do not define selectors, the endpoints controller does not create `Endpoints` records. However, the DNS system looks for and configures either:

- CNAME records for `ExternalName`-type Services.
- A records for any `Endpoints` that share a name with the Service, for all other types.

Publishing Services (ServiceTypes)

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

Kubernetes `ServiceTypes` allow you to specify what kind of Service you want. The default is `ClusterIP` .

`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 `ServiceType` .
- `NodePort`: Exposes the Service on each Node's IP at a static port (the `NodePort`). A `ClusterIP` Service, to which the `NodePort` Service routes, is automatically created. You'll be able to contact the `NodePort` Service, from outside the cluster, by requesting `<NodeIP>:<NodePort>` .
- `LoadBalancer`: Exposes the Service externally using a cloud provider's load balancer. `NodePort` and `ClusterIP` Services, to which the external load balancer routes, are automatically created.
- `ExternalName`: Maps the Service to the contents of the `externalName` field (e.g. `foo.bar.example.com`), by returning a `CNAME` record with its value. No proxying of any kind is set up.

> **Note:** You need either `kube-dns` version 1.7 or CoreDNS version 0.0.8 or higher to use the `ExternalName` type.

You can also use Ingress to expose your Service. Ingress is not a Service type, but it acts as the entry point for your cluster. It lets you consolidate your routing rules into a single resource as it can expose multiple services under 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.

If you want to specify particular IP(s) 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).

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.

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' IPs directly.

Note that 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 filtered node IP(s).

For example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: MyApp
  ports:
      # By default and for convenience, the `targetPort` is set to the same value as t
    - port: 80
      targetPort: 80
      # Optional field
      # By default and for convenience, the Kubernetes control plane will allocate a p
      nodePort: 30007
```

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:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: 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.

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 loadBalancer 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:**
> On **Azure**, if you want to use a user-specified public type `loadBalancerIP`, you first need to create a static type public IP address resource. This public IP address resource should be in the same resource group of the other automatically created resources of the cluster. For example, `MC_myResourceGroup_myAKSCluster_eastus`.
>
> Specify the assigned IP address as loadBalancerIP. Ensure that you have updated the securityGroupName in the cloud provider configuration file. For information about troubleshooting `CreatingLoadBalancerFailed` permission issues see, Use a static IP address with the Azure Kubernetes Service (AKS) load balancer or CreatingLoadBalancerFailed on AKS cluster with advanced networking.

Load balancers with mixed protocol types

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

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.

If the feature gate `MixedProtocolLBService` is enabled for the kube-apiserver it is allowed to use different protocols when there is more than one port defined.

> **Note:** The set of protocols that can be used for LoadBalancer type of Services is still defined by the cloud provider.

Disabling load balancer NodePort allocation

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

Starting in v1.20, you can optionally disable node port allocation for a Service 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. You must enable the `ServiceLBNodePortControl` feature gate to use this field.

Specifying class of load balancer implementation

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

Starting in v1.21, you can optionally specify the class of a load balancer implementation for `LoadBalancer` type of Service by setting the field `spec.loadBalancerClass`. By default, `spec.loadBalancerClass` is `nil` and a `LoadBalancer` type of Service uses the cloud provider's default load balancer implementation. If `spec.loadBalancerClass` is specified, 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. You must enable the `ServiceLoadBalancerClass` feature gate to use this field.

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

Select one of the tabs.

TLS support on AWS

For partial TLS / SSL support on clusters running on AWS, you can add three annotations to a `LoadBalancer` service:

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:us-east-1:12345
```

The first specifies the ARN of the certificate to use. It can be either a certificate from a third party issuer that was uploaded to IAM or one created within AWS Certificate Manager.

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: (https|http|ssl|tcp
```

The second annotation specifies which protocol a Pod speaks. For HTTPS and SSL, the ELB expects the Pod to authenticate itself over the encrypted connection, using a certificate.

HTTP and HTTPS selects layer 7 proxying: the ELB terminates the connection with the user, parses headers, and injects the `X-Forwarded-For` header with the user's IP address (Pods only see the IP address of the ELB at the other end of its connection) when forwarding requests.

TCP and SSL selects layer 4 proxying: the ELB forwards traffic without modifying the headers.

In a mixed-use environment where some ports are secured and others are left unencrypted, you can use the following annotations:

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
        service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443,8443"
```

In the above example, if the Service contained three ports, `80`, `443`, and `8443`, then `443` and `8443` would use the SSL certificate, but `80` would be proxied HTTP.

From Kubernetes v1.9 onwards you can use predefined AWS SSL policies with HTTPS or SSL listeners for your Services. To see which policies are available for use, you can use the `aws` command line tool:

```
aws elb describe-load-balancer-policies --query 'PolicyDescriptions[].PolicyName'
```

You can then specify any one of those policies using the "`service.beta.kubernetes.io/aws-load-balancer-ssl-negotiation-policy`" annotation; for example:

```
    metadata:
      name: my-service
```

```
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-ssl-negotiation-policy: "ELBSecur
```

PROXY protocol support on AWS

To enable PROXY protocol support for clusters running on AWS, you can use the following service annotation:

```
  metadata:
    name: my-service
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "*"
```

Since version 1.3.0, the use of this annotation applies to all ports proxied by the ELB and cannot be configured otherwise.

ELB Access Logs on AWS

There are several annotations to manage access logs for ELB Services on AWS.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-enabled` controls whether access logs are enabled.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval` controls the interval in minutes for publishing the access logs. You can specify an interval of either 5 or 60 minutes.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name` controls the name of the Amazon S3 bucket where load balancer access logs are stored.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix` specifies the logical hierarchy you created for your Amazon S3 bucket.

```
  metadata:
    name: my-service
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-access-log-enabled: "true"
      # Specifies whether access logs are enabled for the load balancer
      service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval: "60"
      # The interval for publishing the access logs. You can specify an interval of
      service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name: "my-bu
      # The name of the Amazon S3 bucket where the access logs are stored
      service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix: "my-
      # The logical hierarchy you created for your Amazon S3 bucket, for example `my-
```

Connection Draining on AWS

Connection draining for Classic ELBs can be managed with the annotation `service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled` set to the value of `"true"`. The annotation `service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout` can also be used to set maximum time, in seconds, to keep the existing connections open before deregistering the instances.

```
  metadata:
    name: my-service
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled: "tru
      service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout: "60"
```

Other ELB annotations

There are other annotations to manage Classic Elastic Load Balancers that are described below.

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout: "60"
        # The time, in seconds, that the connection is allowed to be idle (no data has

        service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled
        # Specifies whether cross-zone load balancing is enabled for the load balancer

        service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags: "enviro
        # A comma-separated list of key-value pairs which will be recorded as
        # additional tags in the ELB.

        service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold: ""
        # The number of successive successful health checks required for a backend to
        # be considered healthy for traffic. Defaults to 2, must be between 2 and 10

        service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-threshold:
        # The number of unsuccessful health checks required for a backend to be
        # considered unhealthy for traffic. Defaults to 6, must be between 2 and 10

        service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval: "20"
        # The approximate interval, in seconds, between health checks of an
        # individual instance. Defaults to 10, must be between 5 and 300

        service.beta.kubernetes.io/aws-load-balancer-healthcheck-timeout: "5"
        # The amount of time, in seconds, during which no response means a failed
        # health check. This value must be less than the service.beta.kubernetes.io/aw
        # value. Defaults to 5, must be between 2 and 60

        service.beta.kubernetes.io/aws-load-balancer-security-groups: "sg-53fae93f"
        # A list of existing security groups to be configured on the ELB created. Unli
        # service.beta.kubernetes.io/aws-load-balancer-extra-security-groups, this rep
        # of a uniquely generated security group for this ELB.
        # The first security group ID on this list is used as a source to permit incom
        # If multiple ELBs are configured with the same security group ID, only a sing
        # of those ELBs it will remove the single permit line and block access for all
        # This can cause a cross-service outage if not used properly

        service.beta.kubernetes.io/aws-load-balancer-extra-security-groups: "sg-53fae9
        #  A list of additional security groups to be added to the created ELB, this l
        # has a unique security group ID and a matching permit line to allow traffic t
        # Security groups defined here can be shared between services.

        service.beta.kubernetes.io/aws-load-balancer-target-node-labels: "ingress-gw,gw
        # A comma separated list of key-value pairs which are used
        # to select the target nodes for the load balancer
```

Network Load Balancer support on AWS

FEATURE STATE: `Kubernetes v1.15 [beta]`

To use a Network Load Balancer on AWS, use the annotation `service.beta.kubernetes.io/aws-load-balancer-type` with the value set to `nlb`.

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
```

> **Note:** NLB only works with certain instance classes; see the AWS documentation on Elastic Load Balancing for a list of supported instance types.

Unlike Classic Elastic Load Balancers, Network Load Balancers (NLBs) forward the client's IP address through to the node. If a Service's `.spec.externalTrafficPolicy` is set to `Cluster`, the client's IP address is not propagated to the end Pods.

By setting `.spec.externalTrafficPolicy` to `Local`, the client IP addresses is propagated to the end Pods, but this could result in uneven distribution of traffic. Nodes without any Pods for a particular LoadBalancer Service will fail the NLB Target Group's health check on the auto-assigned

`.spec.healthCheckNodePort` and not receive any traffic.

In order to achieve even traffic, either use a DaemonSet or specify a pod anti-affinity to not locate on the same node.

You can also use NLB Services with the internal load balancer annotation.

In order for client traffic to reach instances behind an NLB, the Node security groups are modified with the following IP rules:

Rule	Protocol	Port(s)	IpRange(s)	IpRange Description
Health Check	TCP	NodePort(s) (`.spec.healthCheckNodePort` for `.spec.externalTrafficPolicy = Local`)	Subnet CIDR	kubernetes.io/rule/nlb/health=<loadBalancerName>
Client Traffic	TCP	NodePort(s)	`.spec.loadBalancerSourceRanges` (defaults to `0.0.0.0/0`)	kubernetes.io/rule/nlb/client=<loadBalancerName>
MTU Discovery	ICMP	3,4	`.spec.loadBalancerSourceRanges` (defaults to `0.0.0.0/0`)	kubernetes.io/rule/nlb/mtu=<loadBalancerName>

In order to limit which client IP's can access the Network Load Balancer, specify `loadBalancerSourceRanges`.

```yaml
spec:
  loadBalancerSourceRanges:
    - "143.231.0.0/16"
```

> **Note:** If `.spec.loadBalancerSourceRanges` is not set, Kubernetes allows traffic from `0.0.0.0/0` to the Node Security Group(s). If nodes have public IP addresses, be aware that non-NLB traffic can also reach all instances in those modified security groups.

Other CLB annotations on Tencent Kubernetes Engine (TKE)

There are other annotations for managing Cloud Load Balancers on TKE as shown below.

```yaml
    metadata:
      name: my-service
      annotations:
        # Bind Loadbalancers with specified nodes
        service.kubernetes.io/qcloud-loadbalancer-backends-label: key in (value1, value

        # ID of an existing load balancer
        service.kubernetes.io/tke-existed-lbid: lb-6swtxxxx

        # Custom parameters for the load balancer (LB), does not support modification
        service.kubernetes.io/service.extensiveParameters: ""

        # Custom parameters for the LB listener
        service.kubernetes.io/service.listenerParameters: ""

        # Specifies the type of Load balancer;
        # valid values: classic (Classic Cloud Load Balancer) or application (Applicat
        service.kubernetes.io/loadbalance-type: xxxxx

        # Specifies the public network bandwidth billing method;
        # valid values: TRAFFIC_POSTPAID_BY_HOUR(bill-by-traffic) and BANDWIDTH_POSTPA
```

```
    service.kubernetes.io/qcloud-loadbalancer-internet-charge-type: xxxxxx

    # Specifies the bandwidth value (value range: [1,2000] Mbps).
    service.kubernetes.io/qcloud-loadbalancer-internet-max-bandwidth-out: "10"

    # When this annotation is set, the loadbalancers will only register nodes
    # with pod running on it, otherwise all nodes will be registered.
    service.kubernetes.io/local-svc-only-bind-node-with-pod: true
```

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`:

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

Note: ExternalName accepts an IPv4 address string, but as a DNS names comprised of digits, not as an IP address. ExternalNames that resemble IPv4 addresses are not resolved by CoreDNS or ingress-nginx because ExternalName is intended to specify a canonical DNS name. To hardcode an 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`.

Warning:
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.

Note: This section is indebted to the Kubernetes Tips - Part 1 blog post from Alen Komljen.

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes Services can be exposed on those `externalIPs`. Traffic that ingresses into the cluster with the external IP (as destination IP), on the Service port, will be routed to one of the Service endpoints. `externalIPs` are not managed by Kubernetes and are the responsibility of the cluster administrator.

In the Service spec, `externalIPs` can be specified along with any of the `ServiceTypes`. In the example below, "`my-service`" can be accessed by clients on "`80.11.12.10:80`" (`externalIP:port`)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
```

```
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
  externalIPs:
    - 80.11.12.10
```

Shortcomings

Using the userspace proxy for VIPs works at small to medium scale, but will not scale to very large clusters with thousands of Services. The original design proposal for portals has more details on this.

Using the userspace proxy obscures the source IP address of a packet accessing a Service. This makes some kinds of network filtering (firewalling) impossible. The iptables proxy mode does not obscure in-cluster source IPs, but it does still impact clients coming through a load balancer or node-port.

The `Type` field is designed as nested functionality - each level adds to the previous. This is not strictly required on all cloud providers (e.g. Google Compute Engine does not need to allocate a `NodePort` to make `LoadBalancer` work, but AWS does) but the current API requires it.

Virtual IP implementation

The previous information should be sufficient for many people who want to use Services. However, there is a lot going on behind the scenes that may be worth understanding.

Avoiding collisions

One of the primary philosophies of Kubernetes is that you should not be exposed to situations that could cause your actions to fail through no fault of your own. For the design of the Service resource, this means not making you choose your own port number if that choice might collide with someone else's choice. That is an isolation failure.

In order to allow you to choose a port number for your Services, we must ensure that no two Services can collide. Kubernetes does that by allocating each Service its own IP address.

To ensure each Service receives a unique IP, an internal allocator atomically updates a global allocation map in etcd prior to creating each Service. The map object must exist in the registry for Services to get IP address assignments, otherwise creations will fail with a message indicating an IP address could not be allocated.

In the control plane, a background controller is responsible for creating that map (needed to support migrating from older versions of Kubernetes that used in-memory locking). Kubernetes also uses controllers to check for invalid assignments (eg due to administrator intervention) and for cleaning up allocated IP addresses that are no longer used by any Services.

Service IP addresses

Unlike Pod IP addresses, which actually route to a fixed destination, Service IPs are not actually answered by a single host. Instead, kube-proxy uses iptables (packet processing logic in Linux) to define *virtual* IP addresses which are transparently redirected as needed. When clients connect to the VIP, their traffic is automatically transported to an appropriate endpoint. The environment variables and DNS for Services are actually populated in terms of the Service's virtual IP address (and port).

kube-proxy supports three proxy modes—userspace, iptables and IPVS—which each operate slightly differently.

Userspace

As an example, consider the image processing application described above. When the backend Service is created, the Kubernetes master assigns a virtual IP address, for example 10.0.0.1. Assuming the Service port is 1234, the Service is observed by all of the kube-proxy instances in the cluster. When a proxy sees a new Service, it opens a new random port, establishes an iptables redirect from the virtual IP address to this new port, and starts accepting connections on it.

When a client connects to the Service's virtual IP address, the iptables rule kicks in, and redirects the packets to the proxy's own port. The "Service proxy" chooses a backend, and starts proxying traffic from the client to the backend.

This means that Service owners can choose any port they want without risk of collision. Clients can connect to an IP and port, without being aware of which Pods they are actually accessing.

iptables

Again, consider the image processing application described above. When the backend Service is created, the Kubernetes control plane assigns a virtual IP address, for example 10.0.0.1. Assuming the Service port is 1234, the Service is observed by all of the kube-proxy instances in the cluster. When a proxy sees a new Service, it installs a series of iptables rules which redirect from the virtual IP address to per-Service rules. The per-Service rules link to per-Endpoint rules which redirect traffic (using destination NAT) to the backends.

When a client connects to the Service's virtual IP address the iptables rule kicks in. A backend is chosen (either based on session affinity or randomly) and packets are redirected to the backend. Unlike the userspace proxy, packets are never copied to userspace, the kube-proxy does not have to be running for the virtual IP address to work, and Nodes see traffic arriving from the unaltered client IP address.

This same basic flow executes when traffic comes in through a node-port or through a load-balancer, though in those cases the client IP does get altered.

IPVS

iptables operations slow down dramatically in large scale cluster e.g 10,000 Services. IPVS is designed for load balancing and based on in-kernel hash tables. So you can achieve performance consistency in large number of Services from IPVS-based kube-proxy. Meanwhile, IPVS-based kube-proxy has more sophisticated load balancing algorithms (least conns, locality, weighted, persistence).

API Object

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

Supported protocols

TCP

You can use TCP for any kind of Service, and it's the default network protocol.

UDP

You can use UDP for most Services. For type=LoadBalancer Services, UDP support depends on the cloud provider offering this facility.

SCTP

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

When using a network plugin that supports SCTP traffic, you can use SCTP for most Services. For type=LoadBalancer Services, SCTP support depends on the cloud provider offering this facility. (Most do not).

Warnings

Support for multihomed SCTP associations

> **Warning:**
> The support of multihomed SCTP associations requires that the CNI plugin can support the assignment of multiple interfaces and IP addresses to a Pod.
>
> NAT for multihomed SCTP associations requires special logic in the corresponding kernel modules.

Windows

> **Note:** SCTP is not supported on Windows based nodes.

Userspace kube-proxy

> **Warning:** The kube-proxy does not support the management of SCTP associations when it is in userspace mode.

HTTP

If your cloud provider supports it, you can use a Service in LoadBalancer mode to set up external HTTP / HTTPS reverse proxying, forwarded to the Endpoints of the Service.

> **Note:** You can also use Ingress in place of Service to expose HTTP/HTTPS Services.

PROXY protocol

If your cloud provider supports it, you can use a Service in LoadBalancer mode to configure a load balancer outside of Kubernetes itself, that will forward connections prefixed with PROXY protocol.

The load balancer will send an initial series of octets describing the incoming connection, similar to this example

```
PROXY TCP4 192.0.2.202 10.0.42.7 12345 7\r\n
```

followed by the data from the client.

What's next

- Read Connecting Applications with Services
- Read about Ingress
- Read about EndpointSlices

2 - Topology-aware traffic routing with topology keys

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

> **Note:** This feature, specifically the alpha `topologyKeys` API, is deprecated since Kubernetes v1.21. Topology Aware Hints, introduced in Kubernetes v1.21, provide similar functionality.

Service Topology enables a service to route traffic based upon the Node topology of the cluster. For example, a service can specify that traffic be preferentially routed to endpoints that are on the same Node as the client, or in the same availability zone.

Topology-aware traffic routing

By default, traffic sent to a `ClusterIP` or `NodePort` Service may be routed to any backend address for the Service. Kubernetes 1.7 made it possible to route "external" traffic to the Pods running on the same Node that received the traffic. For `ClusterIP` Services, the equivalent same-node preference for routing wasn't possible; nor could you configure your cluster to favor routing to endpoints within the same zone. By setting `topologyKeys` on a Service, you're able to define a policy for routing traffic based upon the Node labels for the originating and destination Nodes.

The label matching between the source and destination lets you, as a cluster operator, designate sets of Nodes that are "closer" and "farther" from one another. You can define labels to represent whatever metric makes sense for your own requirements. In public clouds, for example, you might prefer to keep network traffic within the same zone, because interzonal traffic has a cost associated with it (and intrazonal traffic typically does not). Other common needs include being able to route traffic to a local Pod managed by a DaemonSet, or directing traffic to Nodes connected to the same top-of-rack switch for the lowest latency.

Using Service Topology

If your cluster has the `ServiceTopology` feature gate enabled, you can control Service traffic routing by specifying the `topologyKeys` field on the Service spec. This field is a preference-order list of Node labels which will be used to sort endpoints when accessing this Service. Traffic will be directed to a Node whose value for the first label matches the originating Node's value for that label. If there is no backend for the Service on a matching Node, then the second label will be considered, and so forth, until no labels remain.

If no match is found, the traffic will be rejected, as if there were no backends for the Service at all. That is, endpoints are chosen based on the first topology key with available backends. If this field is specified and all entries have no backends that match the topology of the client, the service has no backends for that client and connections should fail. The special value `"*"` may be used to mean "any topology". This catch-all value, if used, only makes sense as the last value in the list.

If `topologyKeys` is not specified or empty, no topology constraints will be applied.

Consider a cluster with Nodes that are labeled with their hostname, zone name, and region name. Then you can set the `topologyKeys` values of a service to direct traffic as follows.

- Only to endpoints on the same node, failing if no endpoint exists on the node: `["kubernetes.io/hostname"]`.
- Preferentially to endpoints on the same node, falling back to endpoints in the same zone, followed by the same region, and failing otherwise: `["kubernetes.io/hostname", "topology.kubernetes.io/zone", "topology.kubernetes.io/region"]`. This may be useful, for example, in cases where data locality is critical.
- Preferentially to the same zone, but fallback on any available endpoint if none are available within this zone: `["topology.kubernetes.io/zone", "*"]`.

Constraints

- Service topology is not compatible with `externalTrafficPolicy=Local`, and therefore a Service cannot use both of these features. It is possible to use both features in the same cluster on different Services, only not on the same Service.

- Valid topology keys are currently limited to `kubernetes.io/hostname`, `topology.kubernetes.io/zone`, and `topology.kubernetes.io/region`, but will be generalized to other node labels in the future.
- Topology keys must be valid label keys and at most 16 keys may be specified.
- The catch-all value, `"*"`, must be the last value in the topology keys, if it is used.

Examples

The following are common examples of using the Service Topology feature.

Only Node Local Endpoints

A Service that only routes to node local endpoints. If no endpoints exist on the node, traffic is dropped:

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

Prefer Node Local Endpoints

A Service that prefers node local Endpoints but falls back to cluster wide endpoints if node local endpoints do not exist:

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

Only Zonal or Regional Endpoints

A Service that prefers zonal then regional endpoints. If no endpoints exist in either, traffic is dropped.

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

```
  topologyKeys:
    - "topology.kubernetes.io/zone"
    - "topology.kubernetes.io/region"
```

Prefer Node Local, Zonal, then Regional Endpoints

A Service that prefers node local, zonal, then regional endpoints but falls back to cluster wide endpoints.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
    - "topology.kubernetes.io/zone"
    - "topology.kubernetes.io/region"
    - "*"
```

What's next

- Read about enabling Service Topology
- Read Connecting Applications with Services

3 - DNS for Services and Pods

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

Introduction

Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names.

Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. 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 sets this file for each pod. For example, a query for just `data` may be expanded to `data.test.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 a DNS A or AAAA record, depending on the IP family of the service, for a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. This resolves to the cluster IP of the Service.

"Headless" (without a cluster IP) Services are also assigned a DNS A or AAAA record, depending on the IP family of the service, for a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. Unlike normal Services, this resolves to the set of IPs 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 would have the form `_my-port-name._my-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 `auto-generated-name.my-svc.my-namespace.svc.cluster-domain.example`.

Pods

A/AAAA records

In general a pod has the following DNS resolution:

`pod-ip-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 created by a Deployment or DaemonSet exposed by a Service have the following DNS resolution available:

`pod-ip-address.deployment-name.my-namespace.svc.cluster-domain.example`.

Pod's hostname and subdomain fields

Currently when a pod is created, its hostname is the Pod's `metadata.name` value.

The Pod spec has an optional `hostname` field, which can be used to specify the Pod's hostname. When specified, it takes precedence over the Pod's name to be the hostname of the pod. For example, given a Pod with `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 specify its subdomain. For example, a Pod with `hostname` set to "`foo`", and `subdomain` set to "`bar`", in namespace "`my-namespace`", will have the fully qualified domain name (FQDN) "`foo.bar.my-namespace.svc.cluster-domain.example`".

Example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: default-subdomain
spec:
  selector:
    name: busybox
  clusterIP: None
  ports:
  - name: foo # Actually, no port is needed.
    port: 1234
    targetPort: 1234
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostname: busybox-1
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
---
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
```

If there exists a headless service in the same namespace as the pod and with the same name as the subdomain, the cluster's DNS Server also returns an A or AAAA record for the Pod's fully qualified hostname. For example, given a Pod with the hostname set to "`busybox-1`" and the subdomain set to "`default-subdomain`", and a headless Service named "`default-subdomain`" in the same namespace, the pod will see its own FQDN as "`busybox-1.default-subdomain.my-namespace.svc.cluster-domain.example`". DNS serves an A or AAAA record at that name, pointing to the Pod's IP. Both pods "`busybox1`" and "`busybox2`" can have their distinct A or AAAA records.

The Endpoints object can specify the `hostname` for any endpoint addresses, along with its IP.

> **Note:** Because A or 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 (`default-subdomain.my-namespace.svc.cluster-domain.example`), pointing to the Pod's IP address. Also, Pod needs to become ready in order to have a record unless `publishNotReadyAddresses=True` is set on the Service.

Pod's setHostnameAsFQDN field

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

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.default-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 the upstream nameserver inherited from the node. 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 "`ClusterFirstWithHostNet`".
- "`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`.

```
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 for at most 6 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

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
```

```
    nameservers:
      - 1.2.3.4
    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 1.2.3.4
search ns1.svc.cluster-domain.example my.dns.search.suffix
options ndots:2 edns0
```

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

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

The output is similar to this:

```
nameserver fd00:79:30::a
search default.svc.cluster-domain.example svc.cluster-domain.example cluster-domain.exa
options ndots:5
```

What's next

For guidance on administering DNS configurations, check Configure DNS Service

4 - Connecting Applications with Services

The Kubernetes model for connecting containers

Now that you have a continuously running, replicated application you can expose it on a network. Before discussing the Kubernetes approach to networking, it is worthwhile to contrast it with the "normal" way networking works with Docker.

By default, Docker uses host-private networking, so containers can talk to other containers only if they are on the same machine. In order for Docker containers to communicate across nodes, there must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or ports must be allocated dynamically.

Coordinating port allocations across multiple developers or teams that provide containers is very difficult to do at scale, and exposes users to cluster-level issues outside of their control. Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports. This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT. The rest of this document elaborates on how you can run reliable services on such a networking model.

This guide uses a simple nginx server to demonstrate proof of concept.

Exposing pods to the cluster

We did this in a previous example, but let's do it once again and focus on the networking perspective. Create an nginx Pod, and note that it has a container port specification:

service/networking/run-my-nginx.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

This makes it accessible from any node in your cluster. Check the nodes the Pod is running on:

```
kubectl apply -f ./run-my-nginx.yaml
kubectl get pods -l run=my-nginx -o wide
```

```
NAME                        READY     STATUS    RESTARTS   AGE       IP            NODE
my-nginx-3800858182-jr4a2   1/1       Running   0          13s       10.244.3.4    kube
my-nginx-3800858182-kna2y   1/1       Running   0          13s       10.244.2.5    kube
```

Check your pods' IPs:

```
kubectl get pods -l run=my-nginx -o yaml | grep podIP
    podIP: 10.244.3.4
    podIP: 10.244.2.5
```

You should be able to ssh into any node in your cluster and curl both IPs. Note that the containers are *not* using port 80 on the node, nor are there any special NAT rules to route traffic to the pod. This means you can run multiple nginx pods on the same node all using the same containerPort and access them from any other pod or node in your cluster using IP. Like Docker, ports can still be published to the host node's interfaces, but the need for this is radically diminished because of the networking model.

You can read more about how we achieve this if you're curious.

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

You can create a Service for your 2 nginx replicas with `kubectl expose`:

```
kubectl expose deployment/my-nginx
```

```
service/my-nginx exposed
```

This is equivalent to `kubectl apply -f` the following yaml:

service/networking/nginx-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

This specification will create a Service which targets TCP port 80 on any Pod with the `run: my-nginx` label, and expose it on an abstracted Service port (`targetPort`: is the port the container accepts traffic on, `port`: is the abstracted Service port, which can be any port other pods use to access the Service). View Service API object to see the list of supported fields in service definition. Check your Service:

```
kubectl get svc my-nginx
```

```
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
my-nginx   ClusterIP   10.0.162.149   <none>        80/TCP    21s
```

As mentioned previously, a Service is backed by a group of Pods. These Pods are exposed through `endpoints`. The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named `my-nginx`. When a Pod dies, it is automatically removed from the endpoints, and new Pods matching the Service's selector will automatically get added to the endpoints. Check the endpoints, and note that the IPs are the same as the Pods created in the first step:

```
kubectl describe svc my-nginx
```

```
Name:                my-nginx
Namespace:           default
Labels:              run=my-nginx
Annotations:         <none>
Selector:            run=my-nginx
Type:                ClusterIP
IP:                  10.0.162.149
Port:                <unset> 80/TCP
Endpoints:           10.244.2.5:80,10.244.3.4:80
Session Affinity:    None
Events:              <none>
```

```
kubectl get ep my-nginx
```

```
NAME       ENDPOINTS                       AGE
my-nginx   10.244.2.5:80,10.244.3.4:80   1m
```

You should now be able to curl the nginx Service on `<CLUSTER-IP>:<PORT>` from any node in your cluster. Note that the Service IP is completely virtual, it never hits the wire. If you're curious about how this works you can read more about the service proxy.

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the CoreDNS cluster addon.

> **Note:** If the service environment variables are not desired (because possible clashing with expected program ones, too many variables to process, only using DNS, etc) you can disable this mode by setting the `enableServiceLinks` flag to `false` on the pod spec.

Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx Pods (your Pod name will be different):

```
kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 Pods and waiting for the Deployment to recreate them. This time around the Service exists *before* the replicas. This will give you scheduler-level Service spreading of your Pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
kubectl scale deployment my-nginx --replicas=0; kubectl scale deployment my-nginx --rep

kubectl get pods -l run=my-nginx -o wide
```

```
NAME                        READY     STATUS    RESTARTS   AGE     IP            NODE
my-nginx-3800858182-e9ihh   1/1       Running   0          5s      10.244.2.7    kuberr
my-nginx-3800858182-j4rm4   1/1       Running   0          5s      10.244.3.8    kuberr
```

You may notice that the pods have different names, since they are killed and recreated.

```
kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
```

DNS

Kubernetes offers a DNS cluster addon Service that automatically assigns dns names to other Services. You can check if it's running on your cluster:

```
kubectl get services kube-dns --namespace=kube-system
```

```
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   ClusterIP   10.0.0.10    <none>        53/UDP,53/TCP   8m
```

The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a DNS server that has assigned a name to that IP. Here we use the CoreDNS cluster addon (application name `kube-dns`), so you can talk to the Service from any pod in your cluster using standard methods (e.g. `gethostbyname()`). If CoreDNS isn't running, you can enable it referring to the CoreDNS README or Installing CoreDNS. Let's run another curl application to test this:

```
kubectl run curl --image=radial/busyboxplus:curl -i --tty
```

```
Waiting for pod default/curl-131556218-9fnch to be running, status is Pending, pod reac
Hit enter for command prompt
```

Then, hit enter and run `nslookup my-nginx`:

```
[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      my-nginx
Address 1: 10.0.162.149
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure. For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A secret that makes the certificates accessible to pods

You can acquire all these from the nginx https example. This requires having go and make tools installed. If you don't want to install those, then follow the manual steps later. In short:

```
make keys KEY=/tmp/nginx.key CERT=/tmp/nginx.crt
kubectl create secret tls nginxsecret --key /tmp/nginx.key --cert /tmp/nginx.crt
```

```
secret/nginxsecret created
```

```
kubectl get secrets
```

```
NAME                  TYPE                                  DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1         1d
nginxsecret           kubernetes.io/tls                     2         1m
```

And also the configmap:

```
kubectl create configmap nginxconfigmap --from-file=default.conf
```

```
configmap/nginxconfigmap created
```

```
kubectl get configmaps
```

```
NAME             DATA   AGE
nginxconfigmap   1      114s
```

Following are the manual steps to follow in case you run into problems running make (on windows for example):

```
# Create a public private key pair
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /d/tmp/nginx.key -out /d/tr
# Convert the keys to base64 encoding
cat /d/tmp/nginx.crt | base64
cat /d/tmp/nginx.key | base64
```

Use the output from the previous commands to create a yaml file as follows. The base64 encoded value should all be on a single line.

```
apiVersion: "v1"
kind: "Secret"
metadata:
  name: "nginxsecret"
```

```
  namespace: "default"
type: kubernetes.io/tls
data:
  tls.crt: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURIekNDQWdlZ0F3SUJBZ0lKQUp5M3lQK0p
  tls.key: "LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1JSUV2UUlCQURBTkJna3Foa2lHOXcwQkFRRUZ
```

Now create the secrets using the file:

```
kubectl apply -f nginxsecrets.yaml
kubectl get secrets
```

```
NAME                  TYPE                                  DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1         1d
nginxsecret           kubernetes.io/tls                     2         1m
```

Now modify your nginx replicas to start an https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

service/networking/nginx-secure-app.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    run: my-nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      - name: configmap-volume
        configMap:
          name: nginxconfigmap
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
```

```
        - mountPath: /etc/nginx/ssl
          name: secret-volume
        - mountPath: /etc/nginx/conf.d
          name: configmap-volume
```

Noteworthy points about the nginx-secure-app manifest:

- It contains both Deployment and Service specification in the same file.
- The nginx server serves HTTP traffic on port 80 and HTTPS traffic on 443, and nginx Service exposes both ports.
- Each container has access to the keys through a volume mounted at `/etc/nginx/ssl`. This is setup *before* the nginx server is started.

```
kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.yaml
```

At this point you can reach the nginx server from any node.

```
kubectl get pods -o yaml | grep -i podip
    podIP: 10.244.3.5
node $ curl -k https://10.244.3.5
...
<h1>Welcome to nginx!</h1>
```

Note how we supplied the `-k` parameter to curl in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell curl to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup. Let's test this from a pod (the same secret is being reused for simplicity, the pod only needs nginx.crt to access the Service):

service/networking/curlpod.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curl-deployment
spec:
  selector:
    matchLabels:
      app: curlpod
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: curlpod
        command:
        - sh
        - -c
        - while true; do sleep 1; done
        image: radial/busyboxplus:curl
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

```
kubectl apply -f ./curlpod.yaml
kubectl get pods -l app=curlpod
```

```
NAME                                READY     STATUS    RESTARTS   AGE
curl-deployment-1515033274-1410r    1/1       Running   0          1m
```

```
kubectl exec curl-deployment-1515033274-1410r -- curl https://my-nginx --cacert /etc/ng
...
<title>Welcome to nginx!</title>
...
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The Service created in the last section already used `NodePort`, so your nginx HTTPS replica is ready to serve traffic on the internet if your node has a public IP.

```
kubectl get svc my-nginx -o yaml | grep nodePort -C 5
  uid: 07191fb3-f61a-11e5-8ae5-42010af00002
spec:
  clusterIP: 10.0.162.149
  ports:
  - name: http
    nodePort: 31704
    port: 8080
    protocol: TCP
    targetPort: 80
  - name: https
    nodePort: 32453
    port: 443
    protocol: TCP
    targetPort: 443
  selector:
    run: my-nginx
```

```
kubectl get nodes -o yaml | grep ExternalIP -C 1
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...

$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
...
<h1>Welcome to nginx!</h1>
```

Let's now recreate the Service to use a cloud load balancer. Change the `Type` of `my-nginx` Service from `NodePort` to `LoadBalancer`:

```
kubectl edit svc my-nginx
kubectl get svc my-nginx
```

```
NAME       TYPE           CLUSTER-IP     EXTERNAL-IP        PORT(S)               AGE
my-nginx   LoadBalancer   10.0.162.149     xx.xxx.xxx.xxx     8080:30163/TCP        21s
```

```
curl https://<EXTERNAL-IP> -k
...
<title>Welcome to nginx!</title>
```

The IP address in the `EXTERNAL-IP` column is the one that is available on the public internet. The `CLUSTER-IP` is only available inside your cluster/private cloud network.

Note that on AWS, type `LoadBalancer` creates an ELB, which uses a (long) hostname, not an IP. It's too long to fit in the standard `kubectl get svc` output, in fact, so you'll need to do `kubectl describe service my-nginx` to see it. You'll see something like this:

```
kubectl describe service my-nginx
...
LoadBalancer Ingress:   a320587ffd19711e5a37606cf4a74574-1142138393.us-east-1.elb.amaz
...
```

What's next

- Learn more about Using a Service to Access an Application in a Cluster
- Learn more about Connecting a Front End to a Back End Using a Service
- Learn more about Creating an External Load Balancer

5 - Ingress

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.

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:

```
graph LR; client([client])-. Ingress-managed
load balancer .->ingress[Ingress]; ingress-->|routing rule|service[Service];
subgraph cluster ingress; service-->pod1[Pod]; service-->pod2[Pod]; end classDef
plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s
fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster
fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class
ingress,service,pod1,pod2 k8s; class client plain; class cluster cluster;
```

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:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

As with all other Kubernetes resources, an Ingress needs `apiVersion`, `kind`, and `metadata` 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 controller 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.

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 matches 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. The `defaultBackend` is conventionally a configuration option of the Ingress controller and is not specified in your Ingress resources.

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-asset
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
```

IngressClass resources contain an optional parameters field. This can be used to reference additional implementation-specific configuration for this class.

Namespace-scoped parameters

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

`Parameters` field has a `scope` and `namespace` field that can be used to reference a namespace-specific resource for configuration of an Ingress class. `Scope` field defaults to `Cluster`, meaning, the default is cluster-scoped resource. Setting `Scope` to `Namespace` and setting the `Namespace` field will reference a parameters resource in a specific namespace:

service/networking/namespaced-params.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
    namespace: external-configuration
    scope: Namespace
```

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.

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

```
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:

graph LR; client([client])-. Ingress-managed load balancer .->ingress[Ingress, 178.91.123.132]; ingress-->|/foo|service1[Service service1:4200]; ingress-->|/bar|service2[Service service2:8080]; subgraph cluster ingress; service1-->pod1[Pod]; service1-->pod2[Pod]; service2-->pod3[Pod]; service2-->pod4[Pod]; end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class ingress,service1,service2,pod1,pod2,pod3,pod4 k8s; class client plain; class cluster cluster;

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
```

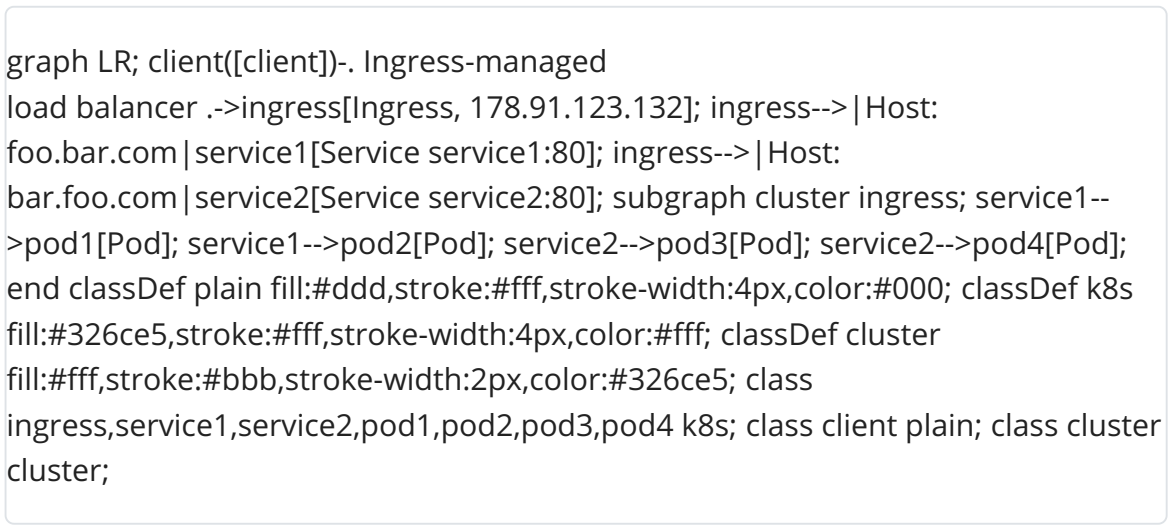
```
Name:             simple-fanout-example
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:4200 (10.8.0.90:4200)
               /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.



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 to the IP address without a hostname defined in request (that is, without a request header being presented) 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

6 - Ingress Controllers

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

Caution: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects. This page follows CNCF website guidelines by listing projects alphabetically. To add a project to this list, read the content guide before submitting a change.

- AKS Application Gateway Ingress Controller is an ingress controller that configures the Azure Application Gateway.
- Ambassador API Gateway is an Envoy-based ingress controller.
- 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.
- The Citrix ingress controller works with Citrix Application Delivery Controller.
- Contour is an Envoy based ingress controller.
- EnRoute is an Envoy 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.
- Gloo is an open-source ingress controller based on Envoy, which offers API gateway functionality.
- HAProxy Ingress is an ingress controller for HAProxy.
- 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.
- The NGINX Ingress Controller for Kubernetes works with the NGINX webserver (as a proxy).
- 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.

Using multiple Ingress controllers

You may deploy any number of ingress controllers within a cluster. When you create an ingress, you should annotate each ingress with the appropriate `ingress.class` to indicate which ingress controller should be used if more than one exists within your cluster.

If you do not define a class, your cloud provider may use a default ingress controller.

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.

7 - EndpointSlices

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

EndpointSlices provide a simple way to track network endpoints within a Kubernetes cluster. They offer a more scalable and extensible alternative to Endpoints.

Motivation

The Endpoints API has provided a simple and straightforward way of tracking network endpoints in Kubernetes. Unfortunately as Kubernetes clusters and Services have grown to handle and send more traffic to more backend Pods, 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 resource, those resources could get quite large. That affected the performance of Kubernetes components (notably the master control plane) and resulted in significant amounts of network traffic and processing when Endpoints changed. EndpointSlices help you mitigate those issues as well as provide an extensible platform for additional features such as topological routing.

EndpointSlice resources

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 resource for 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. When enabled, they should provide a performance improvement for services with large numbers of endpoints.

Address types

EndpointSlices support three address types:

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

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 v1.20 [alpha]`

`serving` is identical to the `ready` condition, except it does not account for terminating states. Consumers of the EndpointSlice API should check this 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 break 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 v1.20 [alpha]`

`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 be 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 preferrable 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. Implementations using EndpointSlice

must be able to have the endpoint appear in more than one slice. A reference implementation of how to perform endpoint deduplication can be found in the `EndpointSliceCache` implementation in `kube-proxy`.

What's next

- Read Connecting Applications with Services

8 - Service Internal Traffic Policy

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

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

Once you have enabled the `ServiceInternalTrafficPolicy` feature gate, you can enable an internal-only traffic policy for a Services, 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: 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` or missing, all endpoints are considered. When the feature gate `ServiceInternalTrafficPolicy` is enabled, `spec.internalTrafficPolicy` defaults to "Cluster".

Constraints

- Service Internal Traffic Policy is not used when `externalTrafficPolicy` 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.

What's next

- Read about enabling Topology Aware Hints
- Read about Service External Traffic Policy
- Read Connecting Applications with Services

9 - Topology Aware Hints

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

Topology Aware Hints enable topology aware routing by including suggestions for how clients should consume endpoints. This approach adds metadata to enable consumers of EndpointSlice and / or Endpoints objects, so that traffic to those network endpoints can be routed closer to where it originated.

For example, you can route traffic within a locality to reduce costs, or to improve network performance.

Motivation

Kubernetes clusters are increasingly deployed in multi-zone environments. *Topology Aware Hints* provides a mechanism to help keep traffic within the zone it originated from. This concept is commonly referred to as "Topology Aware Routing". 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 the kube-proxy can then consume those hints, and use them to influence how traffic to is routed (favoring topologically closer endpoints).

Using Topology Aware Hints

If you have enabled the overall feature, you can activate Topology Aware Hints for a Service by setting the `service.kubernetes.io/topology-aware-hints` annotation to `auto`. This tells the EndpointSlice controller to set topology hints if it is deemed safe. Importantly, this does not guarantee that hints will always be set.

How it works

The functionality enabling this feature is split into two components: The EndpointSlice controller and the kube-proxy. This section provides a high level overview of how each component implements this feature.

EndpointSlice controller

The EndpointSlice controller is responsible for setting hints on EndpointSlices when this feature 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 allocated 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 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 either `externalTrafficPolicy` or `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 does not take into account tolerations when deploying 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.

What's next

- Read about enabling Topology Aware Hints

- Read Connecting Applications with Services

</>

</>

10 - Network Policies

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), 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.

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.

Isolated and Non-isolated Pods

By default, pods are non-isolated; they accept traffic from any source.

Pods become isolated by having a NetworkPolicy that selects them. Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. (Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.)

Network policies do not conflict; they are additive. If any policy or policies select a pod, the pod is restricted to what is allowed by the union of those policies' ingress/egress rules. Thus, order of evaluation does not affect the policy result.

For a network flow between two pods to be allowed, both the egress policy on the source pod and the ingress policy on the destination pod need to allow the traffic. If either the egress policy on the source, or the ingress policy on the destination denies the traffic, the traffic will be denied.

The NetworkPolicy resource

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

An example NetworkPolicy might look like this:

```
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 Containers Using 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; this policy:

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

contains a single `from` element allowing connections from Pods with the label `role=client` in namespaces with the label `user=alice`. But *this* policy:

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

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" 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. This policy does not change the default egress isolation behavior.

Default allow all ingress traffic

If you want to allow all traffic to all pods in a namespace (even if policies are added that cause some pods to be treated as "isolated"), you can create a policy that explicitly allows all traffic in that namespace.

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
```

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 default ingress isolation behavior.

Default allow all egress traffic

If you want to allow all traffic from all pods in a namespace (even if policies are added that cause some pods to be treated as "isolated"), you can create a policy that explicitly allows all egress traffic 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
```

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.

SCTP support

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

As a stable feature, this is enabled by default. To disable SCTP at a cluster level, you (or your cluster administrator) will need to disable the `SCTPSupport` feature gate for the API server with `--feature-gates=SCTPSupport=false,…`. When the feature gate is enabled, you can set the `protocol` field of a NetworkPolicy to `SCTP`.

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

Targeting a range of Ports

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

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:

```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 `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:

- As an alpha feature, this is disabled by default. To enable the `endPort` field at a cluster level, you (or your cluster administrator) need to enable the `NetworkPolicyEndPort` feature gate for the API server with `--feature-gates=NetworkPolicyEndPort=true,…`.
- The `endPort` field must be equal than or greater to 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.

Targeting a Namespace by its name

FEATURE STATE: `Kubernetes 1.21 [beta]`

The Kubernetes control plane sets an immutable label `kubernetes.io/metadata.name` on all namespaces, provided that the `NamespaceDefaultLabelName` feature gate is enabled. 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.

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

As of Kubernetes 1.21, 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).

What's next

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

11 - Adding entries to Pod /etc/hosts with HostAliases

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

Modification not using HostAliases is not suggested because the file is managed by the kubelet and can be overwritten on during Pod creation/restart.

Default hosts file content

Start an Nginx Pod which is assigned a Pod IP:

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

```
pod/nginx created
```

Examine a Pod IP:

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

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

The hosts file content would look like this:

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

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

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

Adding additional entries with hostAliases

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

service/networking/hostaliases-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
```

```
  restartPolicy: Never
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "foo.local"
    - "bar.local"
  - ip: "10.1.2.3"
    hostnames:
    - "foo.remote"
    - "bar.remote"
  containers:
  - name: cat-hosts
    image: busybox
    command:
    - cat
    args:
    - "/etc/hosts"
```

You can start a Pod with that configuration by running:

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

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

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

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

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

The `hosts` file content looks like this:

```
kubectl logs hostaliases-pod
```

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

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

with the additional entries specified at the bottom.

Why does the kubelet manage the hosts file?

The kubelet manages the `hosts` file for each container of the Pod to prevent Docker from modifying the file after the containers have already been started.

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

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

12 - IPv4/IPv6 dual-stack

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

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 (such as Kubenet or Calico)

Configure IPv4/IPv6 dual-stack

To use IPv4/IPv6 dual-stack, ensure the `IPv6DualStack` feature gate is enabled for the relevant components of your cluster. (Starting in 1.21, IPv4/IPv6 dual-stack defaults to enabled.)

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>`

> **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)
>
> Starting in 1.21, IPv4/IPv6 dual-stack defaults to enabled. You can disable it when necessary by specifying `--feature-gates="IPv6DualStack=false"` on the kube-apiserver, kube-controller-manager, kubelet, and kube-proxy command line.

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. (If the cluster has `--feature-gates="IPv6DualStack=false"`, this setting follows the same behavior as `SingleStack`.)
- `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 immutable because the `.spec.ClusterIP` cannot be reallocated on a Service that already exists. If you want to change `.spec.ipFamilies`, delete and recreate the 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: MyApp
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

1. 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: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

1. 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: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  ipFamilies:
  - IPv6
  - IPv4
  selector:
    app: 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 will enable dual-stack unless `--feature-gates="IPv6DualStack=false"` is set.)

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: MyApp
spec:
  selector:
    app: 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: 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: MyApp
  type: ClusterIP
status:
  loadBalancer: {}
```

1. 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: MyApp
spec:
  selector:
    app: 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
```

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

```
spec:
  ipFamilyPolicy: SingleStack
```

After:

```
spec:
  ipFamilyPolicy: PreferDualStack
```

1. 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.

What's next

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