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 RFC 1035 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).

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

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

Over Capacity Endpoints

If an Endpoints resource has more than 1000 endpoints then a Kubernetes v1.22 (or later) cluster annotates that Endpoints with `endpoints.kubernetes.io/over-capacity: truncated`. This annotation indicates that the affected Endpoints object is over capacity and that the endpoints controller has truncated the number of endpoints to 1000.

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.

Later in this page you can read about various kube-proxy implementations work. Overall, you should note that, when running `kube-proxy`, kernel level rules may be modified (for example, iptables rules might get created), which won't get cleaned up, in some cases until you reboot. Thus, running kube-proxy is something that should only be done by an administrator which understands the consequences of having a low level, privileged network proxying service on a computer. Although the `kube-proxy` executable supports a `cleanup` function, this function is not an official feature and thus is only available to use as-is.

Configuration

Note that the kube-proxy starts up in different modes, which are determined by its configuration.

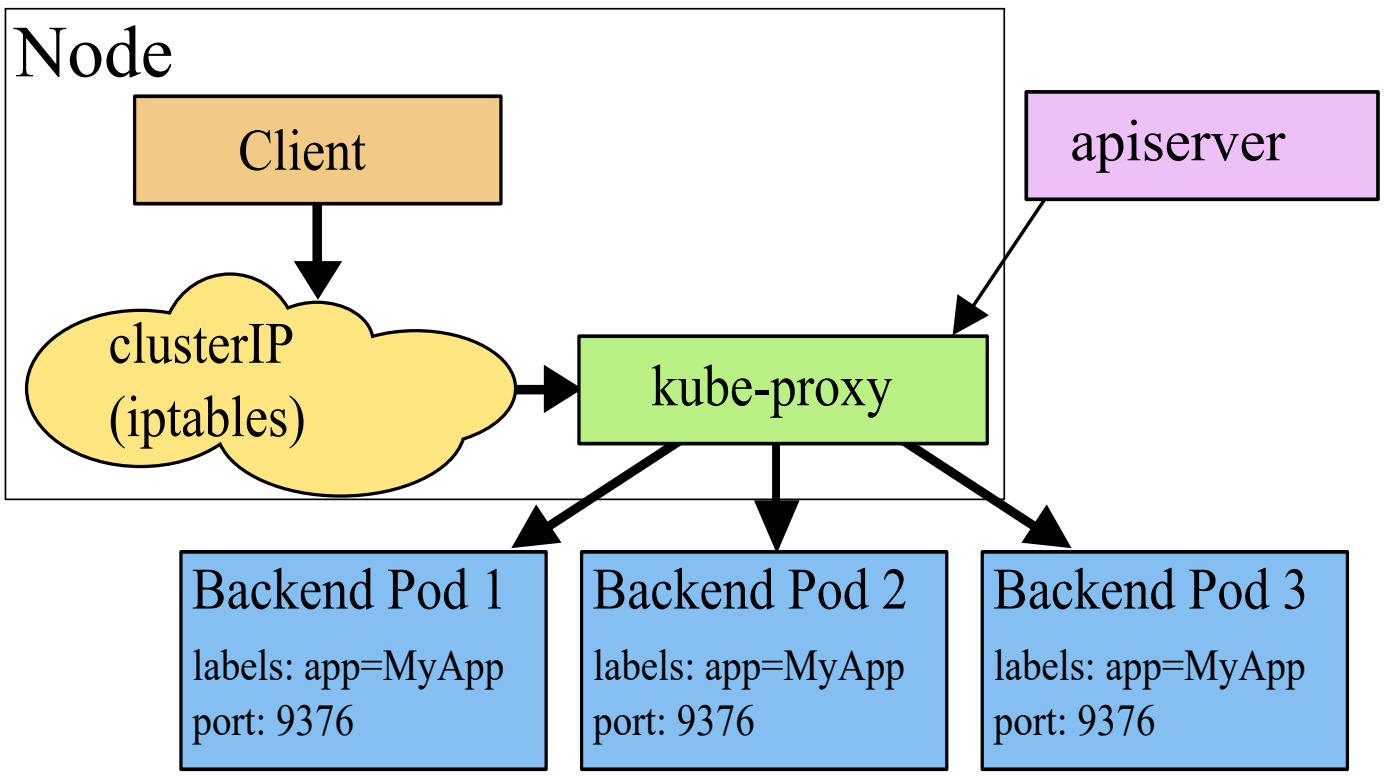
- The kube-proxy's configuration is done via a ConfigMap, and the ConfigMap for kube-proxy effectively deprecates the behaviour for almost all of the flags for the kube-proxy.
- The ConfigMap for the kube-proxy does not support live reloading of configuration.
- The ConfigMap parameters for the kube-proxy cannot all be validated and verified on startup. For example, if your operating system doesn't allow you to run iptables commands, the standard kernel kube-proxy implementation will not work. Likewise, if you have an operating system which doesn't support `netsh`, it will not run in Windows userspace mode.

User space proxy mode

In this (legacy) 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

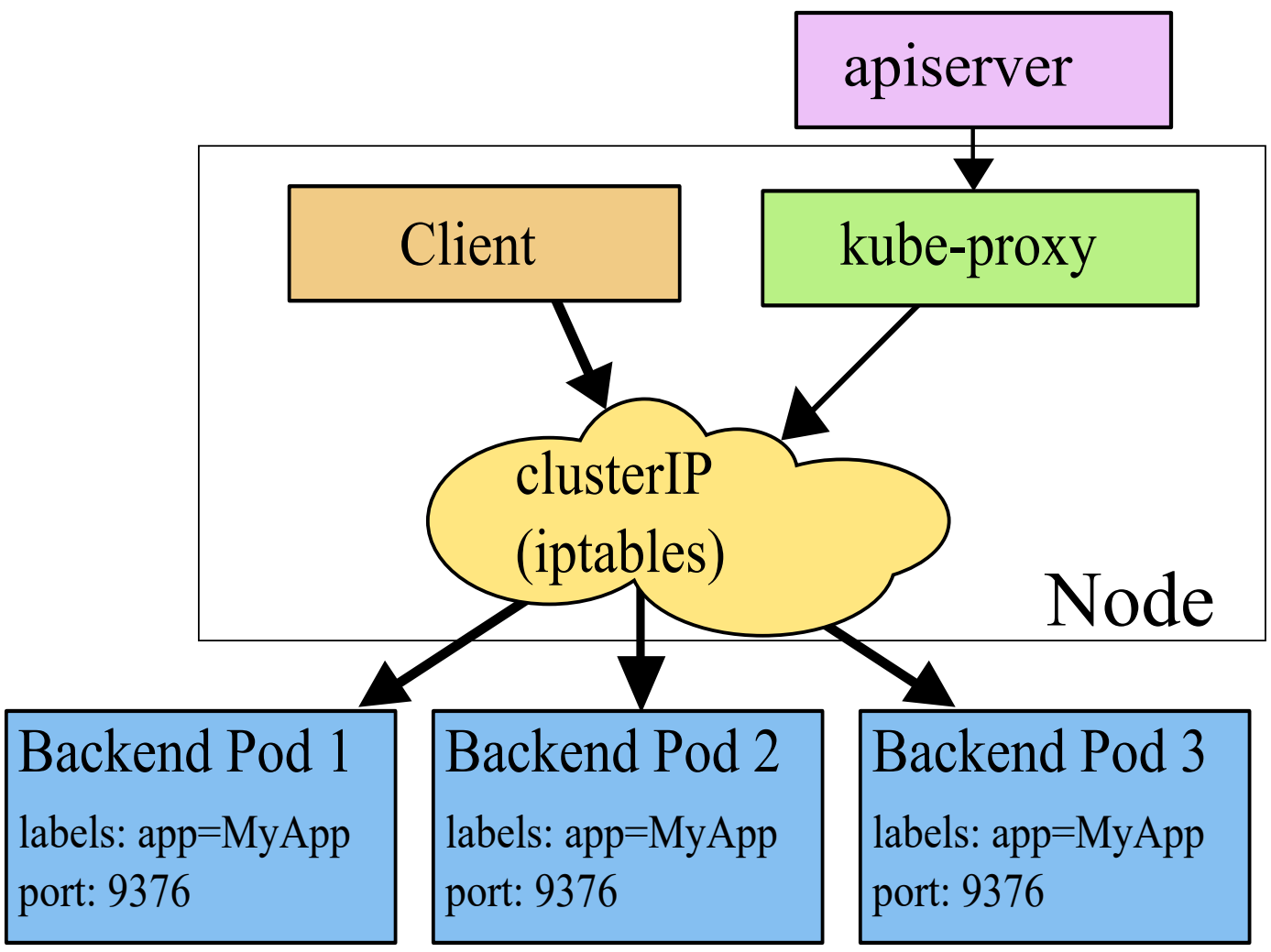
In this mode, kube-proxy watches the Kubernetes control plane for the addition and removal of Service and Endpoint objects. For each Service, it installs iptables rules, which capture traffic to the Service's `clusterIP` and `port`, and redirect that traffic to one of the Service's backend sets. For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

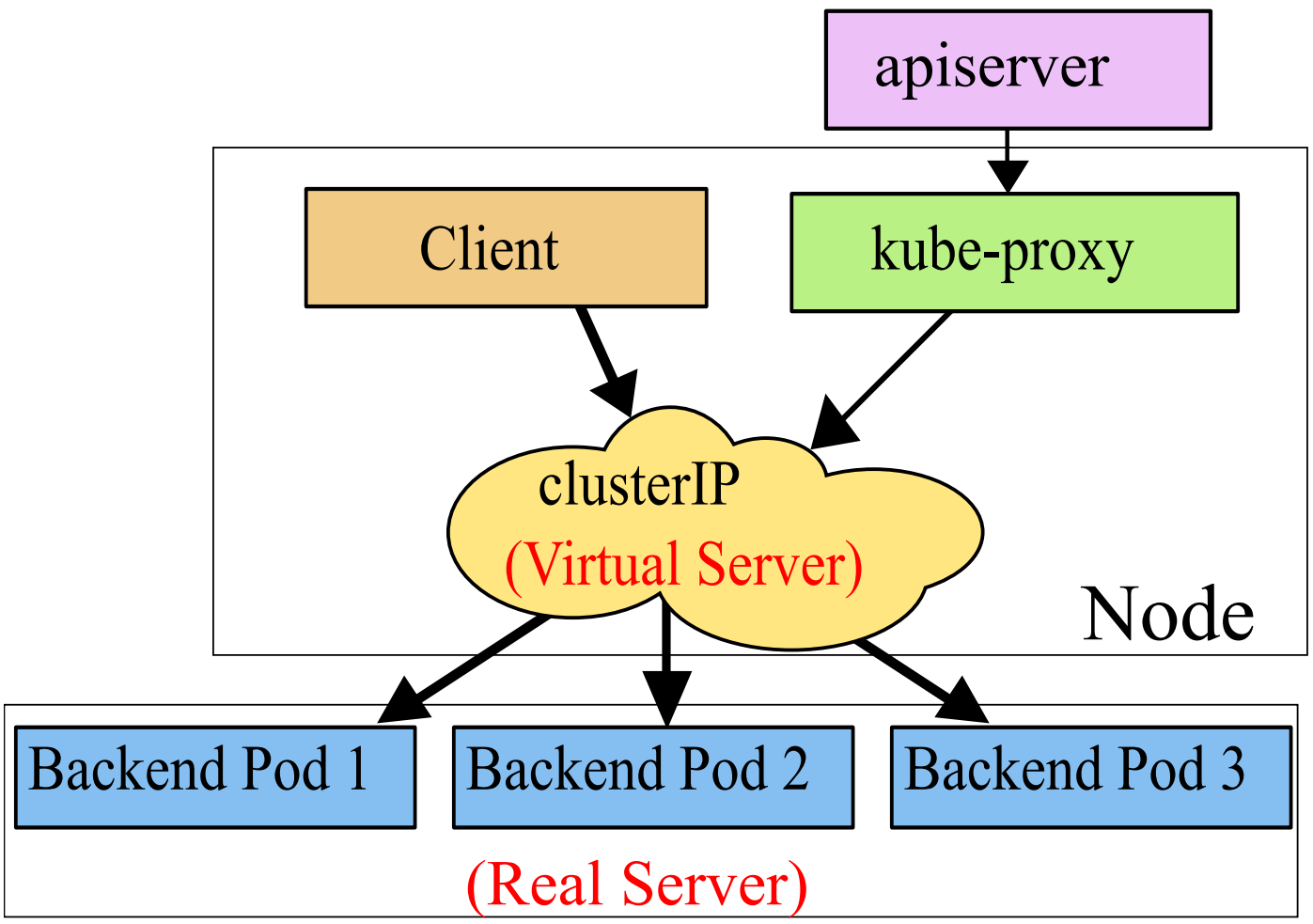
In `ipvs` mode, kube-proxy watches Kubernetes Services and Endpoints, calls `netlink` interface to create IPVS rules accordingly and synchronizes IPVS rules with Kubernetes Services and Endpoints periodically. This control loop ensures that IPVS status matches the desired state. When accessing a Service,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

Traffic policies

External traffic policy

You can set the `spec.externalTrafficPolicy` field to control how traffic from external sources is routed. Valid values are `Cluster` and `Local`. Set the field to `Cluster` to route external traffic to all ready endpoints and `Local` to only route to ready node-local endpoints. If the traffic policy is `Local` and there are are no node-local endpoints, the kube-proxy does not forward any traffic for the relevant Service.

> **Note:**
>
> **FEATURE STATE:** `Kubernetes v1.22 [alpha]`
>
> If you enable the `ProxyTerminatingEndpoints` feature gate for the kube-proxy, the kube-proxy checks if the node has local endpoints and whether or not all the local endpoints are marked as terminating. If there are local endpoints and **all** of those are terminating,

then the kube-proxy ignores any external traffic policy of `Local`. Instead, whilst the node-local endpoints remain as all terminating, the kube-proxy forwards traffic for that Service to healthy endpoints elsewhere, as if the external traffic policy were set to `Cluster`. This forwarding behavior for terminating endpoints exists to allow external load balancers to gracefully drain connections that are backed by `NodePort` Services, even when the health check node port starts to fail. Otherwise, traffic can be lost between the time a node is still in the node pool of a load balancer and traffic is being dropped during the termination period of a pod.

Internal traffic policy

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

You can set the `spec.internalTrafficPolicy` field to control how traffic from internal sources is routed. Valid values are `Cluster` and `Local`. Set the field to `Cluster` to route internal traffic to all ready endpoints and `Local` to only route to ready node-local endpoints. If the traffic policy is `Local` and there are no node-local endpoints, traffic is dropped by kube-proxy.

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:

```yaml
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 the
    - port: 80
      targetPort: 80
      # Optional field
      # By default and for convenience, the Kubernetes control plane will allocate a port
      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:

```yaml
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 v1.22 [beta]`

`spec.loadBalancerClass` enables you to use a load balancer implementation other than the cloud provider default. This feature is available from v1.21, you must enable the `ServiceLoadBalancerClass` feature gate to use this field in v1.21, and the feature gate is enabled by default from v1.22 onwards. By default, `spec.loadBalancerClass` is `nil` and a `LoadBalancer` type of Service uses the cloud provider's default load balancer implementation if the cluster is configured with a cloud provider using the `--cloud-provider` component flag. If `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.

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-1:1234567
```

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: "ELBSecurity
```

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 ei
        service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name: "my-buck
        # 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-bu
        # The logical hierarchy you created for your Amazon S3 bucket, for example `my-b
```

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: "true"
        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 be

        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: "environm
        # 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: "3
    # 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/aws-
    # 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. Unlike
    # service.beta.kubernetes.io/aws-load-balancer-extra-security-groups, this replac
    # of a uniquely generated security group for this ELB.
    # The first security group ID on this list is used as a source to permit incoming
    # If multiple ELBs are configured with the same security group ID, only a single
    # of those ELBs it will remove the single permit line and block access for all El
    # This can cause a cross-service outage if not used properly

    service.beta.kubernetes.io/aws-load-balancer-extra-security-groups: "sg-53fae93f
    #  A list of additional security groups to be added to the created ELB, this leav
    # has a unique security group ID and a matching permit line to allow traffic to t
    # Security groups defined here can be shared between services.

    service.beta.kubernetes.io/aws-load-balancer-target-node-labels: "ingress-gw,gw-n
    # 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.

Further documentation on annotations for Elastic IPs and other common use-cases may be found in the AWS Load Balancer Controller documentation.

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, value2

        # 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 of
        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 (Applicatio
        service.kubernetes.io/loadbalance-type: xxxxx

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

```yaml
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

Feedback

Was this page helpful?

Yes No

Last modified November 20, 2021 at 5:33 PM PST : Fix broken ref in services-networking/service.md (20685c80c)