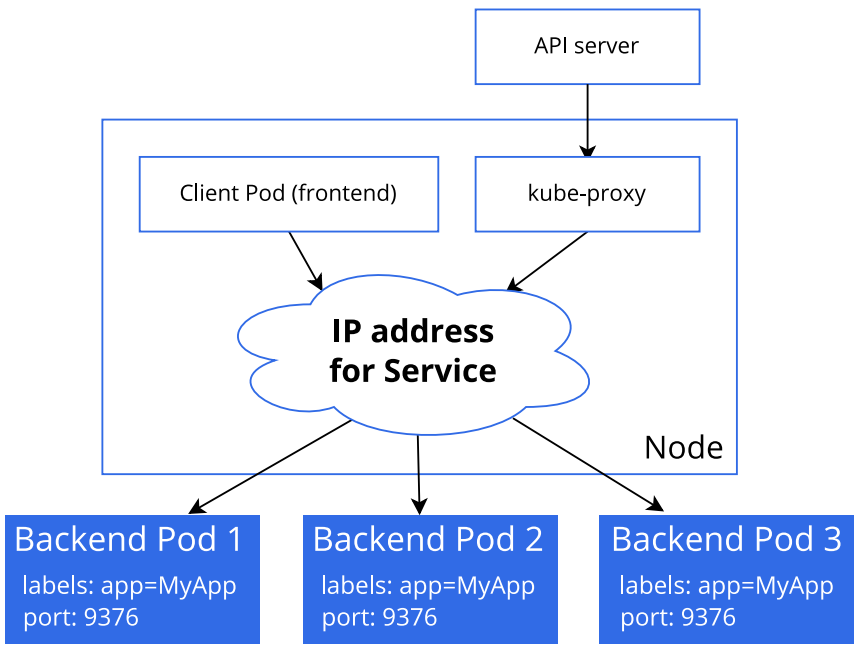
Virtual IPs and Service Proxies

Every node in a Kubernetes cluster runs a kube-proxy (unless you have deployed your own alternative component in place of `kube-proxy`).

The `kube-proxy` component is responsible for implementing a *virtual IP* mechanism for Services of `type` other than `ExternalName`. Each instance of kube-proxy watches the Kubernetes control plane for the addition and removal of Service and EndpointSlice objects. For each Service, kube-proxy calls appropriate APIs (depending on the kube-proxy mode) to configure the node to capture traffic to the Service's `clusterIP` and `port`, and redirect that traffic to one of the Service's endpoints (usually a Pod, but possibly an arbitrary user-provided IP address). A control loop ensures that the rules on each node are reliably synchronized with the Service and EndpointSlice state as indicated by the API server.



Virtual IP mechanism for Services, using iptables mode

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 how 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.

Some of the details in this reference refer to an example: the backend Pods for a stateless image-processing workloads, running with three 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.

Proxy modes

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

On Linux nodes, the available modes for kube-proxy are:

`iptables`

A mode where the kube-proxy configures packet forwarding rules using iptables.

`ipvs`

a mode where the kube-proxy configures packet forwarding rules using ipvs.

`nftables`

a mode where the kube-proxy configures packet forwarding rules using nftables.

There is only one mode available for kube-proxy on Windows:

`kernelspace`

a mode where the kube-proxy configures packet forwarding rules in the Windows kernel

`iptables` proxy mode

This proxy mode is only available on Linux nodes.

In this mode, kube-proxy configures packet forwarding rules using the iptables API of the kernel netfilter subsystem. For each endpoint, it installs iptables rules which, by default, select a backend Pod at random.

Example

As an example, consider the image processing application described earlier in the page. When the backend Service is created, the Kubernetes control plane assigns a virtual IP address, for example 10.0.0.1. For this example, assume that the Service port is 1234. All of the kube-proxy instances in the cluster observe the creation of the new Service.

When kube-proxy on a node sees a new Service, it installs a series of iptables rules which redirect from the virtual IP address to more iptables rules, defined per Service. The per-Service rules link to further rules for each backend endpoint, and the per- endpoint rules 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 without rewriting the 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 address does get altered.

Optimizing iptables mode performance

In iptables mode, kube-proxy creates a few iptables rules for every Service, and a few iptables rules for each endpoint IP address. In clusters with tens of thousands of Pods and Services, this means tens of thousands of iptables rules, and kube-proxy may take a long time to update the rules in the kernel when Services (or their EndpointSlices) change. You can adjust the syncing behavior of kube-proxy via options in the `iptables` section of the kube-proxy configuration file (which you specify via `kube-proxy --config <path>`):

```
...
iptables:
  minSyncPeriod: 1s
  syncPeriod: 30s
...
```

`minSyncPeriod`

The `minSyncPeriod` parameter sets the minimum duration between attempts to resynchronize iptables rules with the kernel. If it is `0s`, then kube-proxy will always immediately synchronize the rules every time any Service or Endpoint changes. This works fine in very small clusters, but it results in a lot of redundant work when lots of things change in a small time period. For example, if you

have a Service backed by a Deployment with 100 pods, and you delete the Deployment, then with `minSyncPeriod: 0s`, kube-proxy would end up removing the Service's endpoints from the iptables rules one by one, for a total of 100 updates. With a larger `minSyncPeriod`, multiple Pod deletion events would get aggregated together, so kube-proxy might instead end up making, say, 5 updates, each removing 20 endpoints, which will be much more efficient in terms of CPU, and result in the full set of changes being synchronized faster.

The larger the value of `minSyncPeriod`, the more work that can be aggregated, but the downside is that each individual change may end up waiting up to the full `minSyncPeriod` before being processed, meaning that the iptables rules spend more time being out-of-sync with the current API server state.

The default value of `1s` should work well in most clusters, but in very large clusters it may be necessary to set it to a larger value. Especially, if kube-proxy's `sync_proxy_rules_duration_seconds` metric indicates an average time much larger than 1 second, then bumping up `minSyncPeriod` may make updates more efficient.

Updating legacy `minSyncPeriod` configuration

Older versions of kube-proxy updated all the rules for all Services on every sync; this led to performance issues (update lag) in large clusters, and the recommended solution was to set a larger `minSyncPeriod`. Since Kubernetes v1.28, the iptables mode of kube-proxy uses a more minimal approach, only making updates where Services or EndpointSlices have actually changed.

If you were previously overriding `minSyncPeriod`, you should try removing that override and letting kube-proxy use the default value (`1s`) or at least a smaller value than you were using before upgrading.

If you are not running kube-proxy from Kubernetes 1.29, check the behavior and associated advice for the version that you are actually running.

`syncPeriod`

The `syncPeriod` parameter controls a handful of synchronization operations that are not directly related to changes in individual Services and EndpointSlices. In particular, it controls how quickly kube-proxy notices if an external component has interfered with kube-proxy's iptables rules. In large clusters, kube-proxy also only performs certain cleanup operations once every `syncPeriod` to avoid unnecessary work.

For the most part, increasing `syncPeriod` is not expected to have much impact on performance, but in the past, it was sometimes useful to set it to a very large value (eg, `1h`). This is no longer recommended, and is likely to hurt functionality more than it improves performance.

IPVS proxy mode

This proxy mode is only available on Linux nodes.

In `ipvs` mode, kube-proxy uses the kernel IPVS and iptables APIs to create rules to redirect traffic from Service IPs to endpoint IPs.

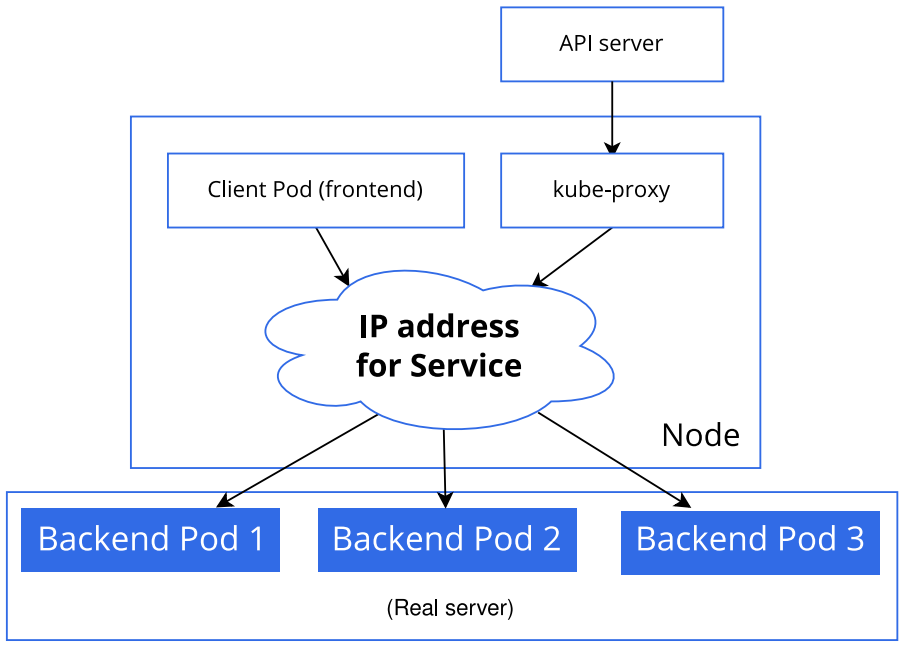
The IPVS proxy mode is based on netfilter hook function that is similar to iptables mode, but uses a hash table as the underlying data structure and works in the kernel space. That means kube-proxy in IPVS mode redirects traffic with lower latency than kube-proxy in iptables mode, with much better performance when synchronizing proxy rules. Compared to the iptables proxy mode, 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): Traffic is equally distributed amongst the backing servers.
- `wrr` (Weighted Round Robin): Traffic is routed to the backing servers based on the weights of the servers. Servers with higher weights receive new connections and get more requests than servers with lower weights.
- `lc` (Least Connection): More traffic is assigned to servers with fewer active connections.
- `wlc` (Weighted Least Connection): More traffic is routed to servers with fewer connections relative to their weights, that is, connections divided by weight.
- `lblc` (Locality based Least Connection): Traffic for the same IP address is sent to the same backing server if the server is not overloaded and available; otherwise the traffic is sent to servers with fewer connections, and keep it for future assignment.
- `lblcr` (Locality Based Least Connection with Replication): Traffic for the same IP address is sent to the server with least connections. If all the backing servers are overloaded, it picks up one with fewer connections and add it to the target set. If the target set has not changed for the specified time, the most loaded server is removed from the set, in order to avoid high degree of replication.

- `sh` (Source Hashing): Traffic is sent to a backing server by looking up a statically assigned hash table based on the source IP addresses.
- `dh` (Destination Hashing): Traffic is sent to a backing server by looking up a statically assigned hash table based on their destination addresses.
- `sed` (Shortest Expected Delay): Traffic forwarded to a backing server with the shortest expected delay. The expected delay is `(C + 1) / U` if sent to a server, where `C` is the number of connections on the server and `U` is the fixed service rate (weight) of the server.
- `nq` (Never Queue): Traffic is sent to an idle server if there is one, instead of waiting for a fast one; if all servers are busy, the algorithm falls back to the `sed` behavior.

> **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 exits with an error.



Virtual IP address mechanism for Services, using IPVS mode

`nftables` proxy mode

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

This proxy mode is only available on Linux nodes.

In this mode, kube-proxy configures packet forwarding rules using the nftables API of the kernel netfilter subsystem. For each endpoint, it installs nftables rules which, by default, select a backend Pod at random.

The nftables API is the successor to the iptables API, and although it is designed to provide better performance and scalability than iptables, the kube-proxy nftables mode is still under heavy development as of 1.29 and is not necessarily expected to outperform the other Linux modes at this time.

`kernelspace` proxy mode

This proxy mode is only available on Windows nodes.

The kube-proxy configures packet filtering rules in the Windows *Virtual Filtering Platform* (VFP), an extension to Windows vSwitch. These rules process encapsulated packets within the node-level virtual networks, and rewrite packets so that the destination IP address (and layer 2 information) is correct for getting the packet routed to the correct destination. The Windows VFP is analogous to tools such as Linux `nftables` or `iptables`. The Windows VFP extends the *Hyper-V Switch*, which was initially implemented to support virtual machine networking.

When a Pod on a node sends traffic to a virtual IP address, and the kube-proxy selects a Pod on a different node as the load balancing target, the `kernelspace` proxy mode rewrites that packet to be destined to the target backend Pod. The Windows *Host Networking Service* (HNS) ensures that packet rewriting rules are configured so that the return traffic appears to come from the virtual IP address and not the specific backend Pod.

Direct server return for `kernelspace` mode

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

As an alternative to the basic operation, a node that hosts the backend Pod for a Service can apply the packet rewriting directly, rather than placing this burden on the node where the client Pod is running. This is called *direct server return*.

To use this, you must run kube-proxy with the `--enable-dsr` command line argument **and** enable the `WinDSR` feature gate.

Direct server return also optimizes the case for Pod return traffic even when both Pods are running on the same node.

Session affinity

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 `.spec.sessionAffinity` to `ClientIP` for a Service (the default is `None`).

Session stickiness timeout

You can also set the maximum session sticky time by setting `.spec.sessionAffinityConfig.clientIP.timeoutSeconds` appropriately for a Service. (the default value is 10800, which works out to be 3 hours).

> **Note:** On Windows, setting the maximum session sticky time for Services is not supported.

IP address assignment to Services

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 packet processing logic (such as Linux iptables) 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).

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 IP address if that choice might collide with someone else's choice. That is an isolation failure.

In order to allow you to choose an IP address for your Services, we must ensure that no two Services can collide. Kubernetes does that by allocating each Service its own IP address from within the `service-cluster-ip-range` CIDR range that is configured for the API Server.

IP address allocation tracking

To ensure each Service receives a unique IP address, 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 (for example: due to administrator intervention) and for cleaning up allocated IP addresses that are no longer used by any Services.

IP address allocation tracking using the Kubernetes API

FEATURE STATE: `Kubernetes v1.27 [alpha]`

If you enable the `MultiCIDRServiceAllocator` feature gate and the `networking.k8s.io/v1alpha1` API group, the control plane replaces the existing etcd allocator with a revised implementation that uses IPAddress and ServiceCIDR objects instead of an internal global allocation map. Each cluster IP address associated to a Service then references an IPAddress object.

Enabling the feature gate also replaces a background controller with an alternative that handles the IPAddress objects and supports migration from the old allocator model. Kubernetes 1.29 does not support migrating from IPAddress objects to the internal allocation map.

One of the main benefits of the revised allocator is that it removes the size limitations for the IP address range that can be used for the cluster IP address of Services. With `MultiCIDRServiceAllocator` enabled, there are no limitations for IPv4, and for IPv6 you can use IP address netmasks that are a /64 or smaller (as opposed to /108 with the legacy implementation).

Making IP address allocations available via the API means that you as a cluster administrator can allow users to inspect the IP addresses assigned to their Services. Kubernetes extensions, such as the Gateway API, can use the IPAddress API to extend Kubernetes' inherent networking capabilities.

Here is a brief example of a user querying for IP addresses:

```
kubectl get services
```

```
NAME         TYPE        CLUSTER-IP        EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   2001:db8:1:2::1   <none>        443/TCP   3d1h
```

```
kubectl get ipaddresses
```

```
NAME              PARENTREF
2001:db8:1:2::1   services/default/kubernetes
2001:db8:1:2::a   services/kube-system/kube-dns
```

Kubernetes also allow users to dynamically define the available IP ranges for Services using ServiceCIDR objects. During bootstrap, a default ServiceCIDR object named `kubernetes` is created from the value of the `--service-cluster-ip-range` command line argument to kube-apiserver:

```
kubectl get servicecidrs
```

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

Users can create or delete new ServiceCIDR objects to manage the available IP ranges for Services:

```
cat <<'EOF' | kubectl apply -f -
apiVersion: networking.k8s.io/v1alpha1
kind: ServiceCIDR
metadata:
  name: newservicecidr
spec:
  cidrs:
  - 10.96.0.0/24
EOF
```

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

```
kubectl get servicecidrs
```

```
NAME             CIDRS          AGE
kubernetes       10.96.0.0/28   17m
newservicecidr   10.96.0.0/24   7m
```

IP address ranges for Service virtual IP addresses

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

Kubernetes divides the `ClusterIP` range into two bands, based on the size of the configured `service-cluster-ip-range` by using the following formula `min(max(16, cidrSize / 16), 256)`. That formula paraphrases as *never less than 16 or more than 256, with a graduated step function between them*.

Kubernetes prefers to allocate dynamic IP addresses to Services by choosing from the upper band, which means that if you want to assign a specific IP address to a `type: ClusterIP` Service, you should manually assign an IP address from the **lower** band. That approach reduces the risk of a conflict over allocation.

Traffic policies

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

Internal traffic policy

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

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.

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.

Traffic to terminating endpoints

FEATURE STATE: `Kubernetes v1.28 [stable]`

If the `ProxyTerminatingEndpoints` feature gate is enabled in kube-proxy and the traffic policy is `Local`, that node's kube-proxy uses a more complicated algorithm to select endpoints for a Service. With the feature enabled, 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 them are terminating, then kube-proxy will forward traffic to those terminating endpoints. Otherwise, kube-proxy will always prefer forwarding traffic to endpoints that are not terminating.

This forwarding behavior for terminating endpoints exist to allow `NodePort` and `LoadBalancer` Services to gracefully drain connections when using `externalTrafficPolicy: Local`.

As a deployment goes through a rolling update, nodes backing a load balancer may transition from N to 0 replicas of that deployment. In some cases, external load balancers can send traffic to a node with 0 replicas in between health check probes. Routing traffic to terminating endpoints ensures that Node's that are scaling down Pods can gracefully receive and drain traffic to those terminating Pods. By the time the Pod completes termination, the external load balancer should have seen the node's health check failing and fully removed the node from the backend pool.

What's next

To learn more about Services, read Connecting Applications with Services.

You can also:

- Read about Services as a concept

- Read about Ingresses as a concept
- Read the API reference for the Service API

Feedback

Was this page helpful?

Yes No

Last modified November 26, 2023 at 9:21 PM PST: Clarify iptables performance slightly (d5c530002f)