**Kube-Proxy**

Behind every Kubernetes network, there’s a crucial component that operates behind the scenes, converting your Services into functional networking rules. This component is known as Kube-Proxy.

In this article, we’ll delve into what Kube-Proxy is, how it functions, and its various modes. We’ll also show you how to inspect IPtables rules for a ClusterIP Service.

**What is Kube-Proxy?**

Kube-Proxy is a Kubernetes agent installed on every node within the cluster. It monitors changes to Service objects and their endpoints, translating these changes into actual network rules within the node.

Typically, Kube-Proxy runs in your cluster as a DaemonSet, but it can also be installed directly as a Linux process on the node.

If you use kubeadm, it will install Kube-Proxy as a DaemonSet. If you manually install the cluster components using official Linux tarball binaries, it will run directly as a process on the node.

### The Role of Kube-Proxy

In Kubernetes, Pods are ephemeral, meaning they can be easily terminated and recreated. When a Pod is recreated, it receives a new IP address, different from its predecessor. This lack of IP permanence makes it impractical to rely on a Pod’s IP address for communication.

Services are crucial in pod networking as they provide a stable IP address and DNS name for a set of pods. This allows other applications to communicate with the pods using a consistent address, even as the pods themselves are terminated and recreated.

Services also load balance traffic between pods, ensuring that requests are evenly distributed and that individual pods do not become overwhelmed. By using Services in conjunction with pods, developers can create highly available and scalable applications capable of handling a wide range of network traffic.

For Service-to-Pod mapping to work, continuous re-mapping at the networking level is necessary. This is precisely what Kube-Proxy does.

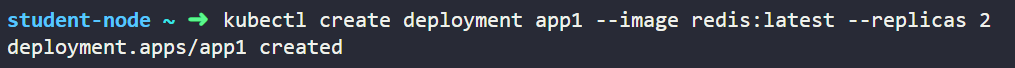
Kube-Proxy facilitates Service-to-Pod mapping by maintaining a network routing table that maps Service IP addresses to the IP addresses of the pods that belong to the Service. When a request is made to a Service, Kube-Proxy uses this mapping to forward the request to a Pod belonging to the Service.

### How Kube-Proxy Works

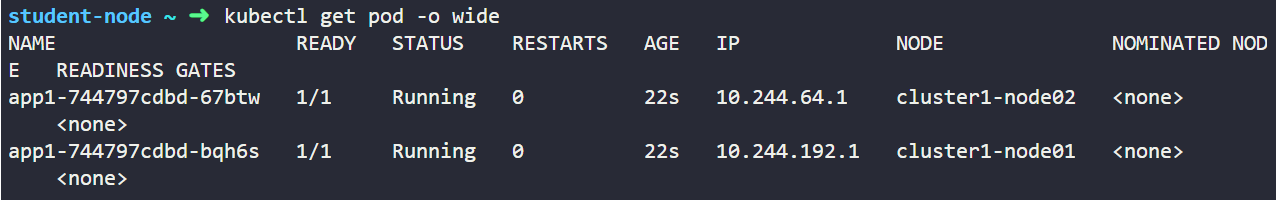
Once Kube-Proxy is installed, it authenticates with the API server. When new Services or endpoints are added or removed, the API server communicates these changes to Kube-Proxy.

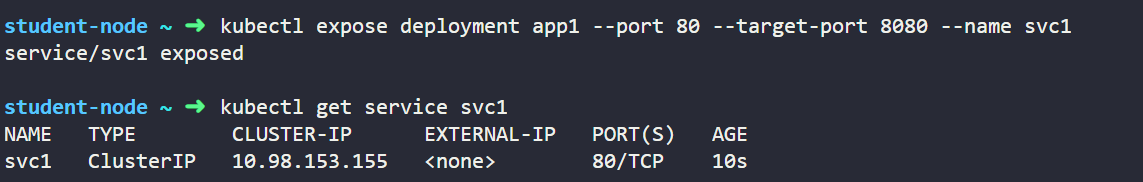
Kube-Proxy then implements these changes as NAT rules within the node. These NAT rules map Service IPs to Pod IPs. When a request is sent to a Service, it is redirected to a backend Pod based on these rules.

Let’s dive into more details with an example.

Let us create a deployment named app1 using image redis:latest with 2 replicas/pods   
  


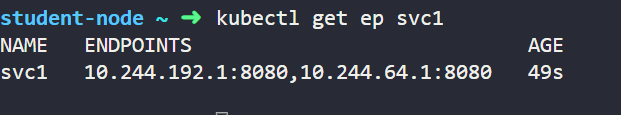
Now let us check the status of our pods, both the pods are running.

s

Now let us expose this deployment app1 as a clusterIP service svc1. Not mentioning the type for the service creates it as a default internal only clusterIP service.  
  


As we know that when a service is created it creates an abstraction called endpoint objects in K8s which are essentially the POD Ips.

We can notice that these are the IP addresses of the pods of our app1 deployment.



This is what makes the pods communicate internally.

When this Service is created, the API server identifies which Pods should be associated with this Service by looking for Pods with labels that match the Service’s label selector.

Let’s call these Pods Pod1 and Pod2. The API server then creates an abstraction called an endpoint, with each endpoint representing the IP of one of the Pods. SVC1 is now linked to two endpoints corresponding to our Pods, which we’ll call EP1 and EP2.

The API server maps the IP address of SVC1 to the IP addresses of EP1 and EP2.

We need this mapping to be applied to the network so that traffic directed to the IP of SVC1 can be forwarded to either EP1 or EP2.

To accomplish this, the API server broadcasts the new mapping to Kube-Proxy on each node, which then implements it as an internal rule.

Now, traffic directed to the SVC1 IP will adhere to this DNAT rule and be forwarded to the Pods. Keep in mind that EP1 and EP2 are essentially the IPs of the Pods.

Here are a few important points to note:

1. The Service and endpoint involve both IP and Port mappings, not just an IP. For example, an IP and Port mapping might look like this: 10.10.30.22:80.
2. In the example, the DNAT translation occurred on the source node because we used a ClusterIP type of Service. This type of Service creates an internal-only NAT rule that cannot be routed outside the cluster, meaning no one outside the cluster is aware of this IP.
3. If a different type of Service is used, the rules are installed differently within the nodes

### Kube-Proxy Modes

Kube-Proxy can operate in three different modes: user-space mode, IPtables mode, and IPVS mode. The mode determines how Kube-Proxy implements the NAT rules. Let’s explore each mode.

#### User-Space Mode

This is a legacy mode that is rarely used today. In user-space mode, Kube-Proxy relies on a Linux feature called IPtables, which functions as an internal packet processing and filtering component. IPtables inspects incoming and outgoing traffic to the Linux machine and applies specific rules to packets that match certain criteria.

In user-space mode, Kube-Proxy inserts a NAT rule inside IPtables. This rule redirects traffic to a local port on the Kube-Proxy itself. Kube-Proxy listens for connections on this port and forwards them to the appropriate backend Pods. The downside of this mode is that traffic needs to be redirected twice: from the kernel space back to the user space and then to the Pod. This negatively impacts latency and throughput, making this mode unpopular.

#### IPtables Mode

This is the default and most widely used mode today. It also relies on the IPtables feature to insert rules into the nodes but works differently from user-space mode. Instead of forwarding connections to Kube-Proxy itself, this mode inserts the Service-to-Pod rules directly into IPtables. This approach frees Kube-Proxy from being in the traffic path, reducing latency.

In this mode, Kube-Proxy acts as the “installer” of rules rather than an actual proxy. However, IPtables uses a sequential approach for table lookups, which can be inefficient as the number of rules increases. This sequential algorithm follows O(n) performance, meaning the number of lookups increases linearly with the number of rules. Additionally, IPtables does not support advanced load-balancing algorithms, using a random equal-cost distribution instead.

#### IPVS Mode

IPVS is a Linux feature designed specifically for load balancing, making it an ideal choice for Kube-Proxy. In this mode, Kube-Proxy inserts rules into IPVS instead of IPtables. IPVS has an optimized lookup algorithm with a complexity of O(1), providing consistent performance regardless of the number of rules.

This mode offers more efficient connection processing for Services and endpoints and supports various load-balancing algorithms such as round robin, least connections, and hashing approaches. Despite its advantages, IPVS might not be present in all Linux systems, whereas IPtables is a core feature of almost every Linux operating system. For clusters with fewer Services, IPtables should work perfectly.

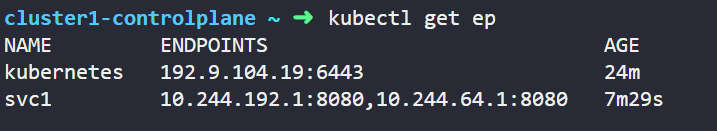
### How to Check Kube-Proxy Mode

By default, Kube-Proxy runs on port 10249 and exposes a set of endpoints that you can use to query Kube-Proxy for information. You can use the /proxyMode endpoint to check the Kube-Proxy mode.

First, connect to one of the nodes in the cluster through SSH. Once connected, run the command:

curl -v localhost:10249/proxyMode

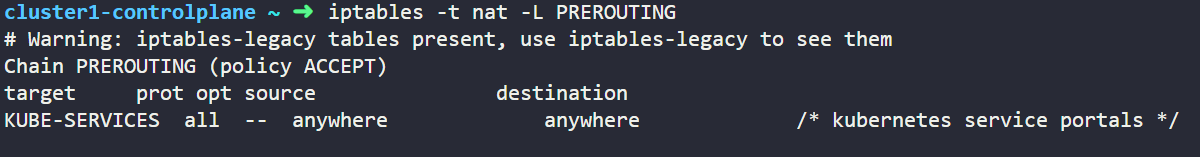
Let us now see the magic under the hood. We saw that the service svc1 has two endpoints that correspond to the POD ips of our app1 deployment.



We’ll list the IPtables rules on one of the nodes.

**Note**: You need to SSH into the node first to run the following commands.

Running this command iptables -n nat -L PREROUTING will give output something similar like this

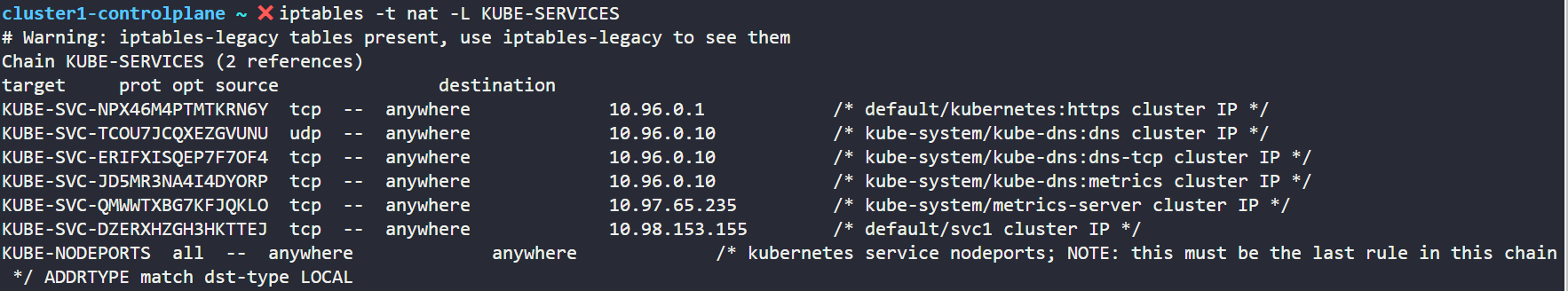


We will just dive into the iptables here and talk about the only important information here

Let’s explain our command options first. The **“-t nat”**refers to which type of table we want to list. IPtables include multiple table types; for Kube-Proxy, it uses the NAT table. This is because Kube-Proxy uses IPtables mainly for translating Service IPs.

The **“-L PREROUTING”**hererefers to the name of the chain in the table. A chain is a group of rules that are applied to packets. The PREROUTING is a chain that exists by default in the IPtables. Kube-Proxy hooks its rules into that chain.

Now let’s move to the command output. The most important part of the output here is the **KUBE-SERVICES** line. This is a custom chain created by Kube-Proxy for the Services.



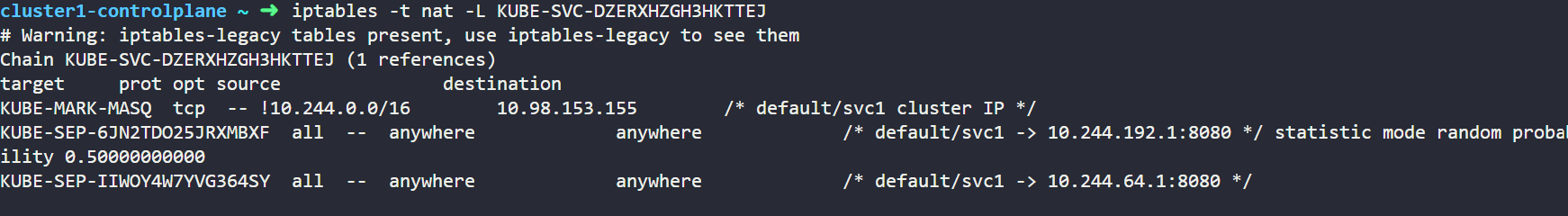
From the output we see there are more chains and we are more interested into the IP addresses here. We can see specific chain created with a rule for the destination IP of the Service. This means that traffic destined for the Service will enter that chain. Let us have a look into the chain “KUBE-SVC-DZERXHZGH3HKTTEJ tcp -- anywhere 10.98.153.155” for our svc1.

Run the command iptables -t nat -L KUBE-SVC-DZERXHZGH3HKTTEJ

We get an output like this

You can also see some information to the right of the IP address. These represent the Namespace, Service name, Service type, and port.

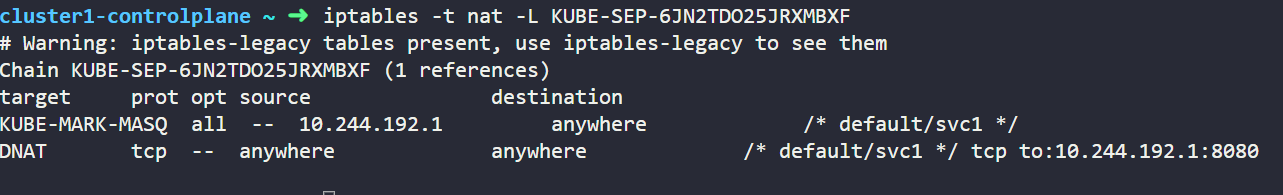
All of this ensures that we’re looking at the correct section of IPtables. Now let’s get into this chain by running the command:



Finally, we’ve arrived at our NAT rules. You’ll find 2 additional chains created. Each starts with **KUBE-SEP**and then a random id. These correspond to the Service endpoints (SEP). You can see the IP address of each Pod listed for each chain.

Notice this **statistic mode random probability** line in the middle? This is the random load balancing rule that the IPtables make between the Pods.

Going into any of these KUBE-SEP chains, we can see it’s basically a DNAT rule.



So that’s how we see iptables rules for our service and how it works at the backend.