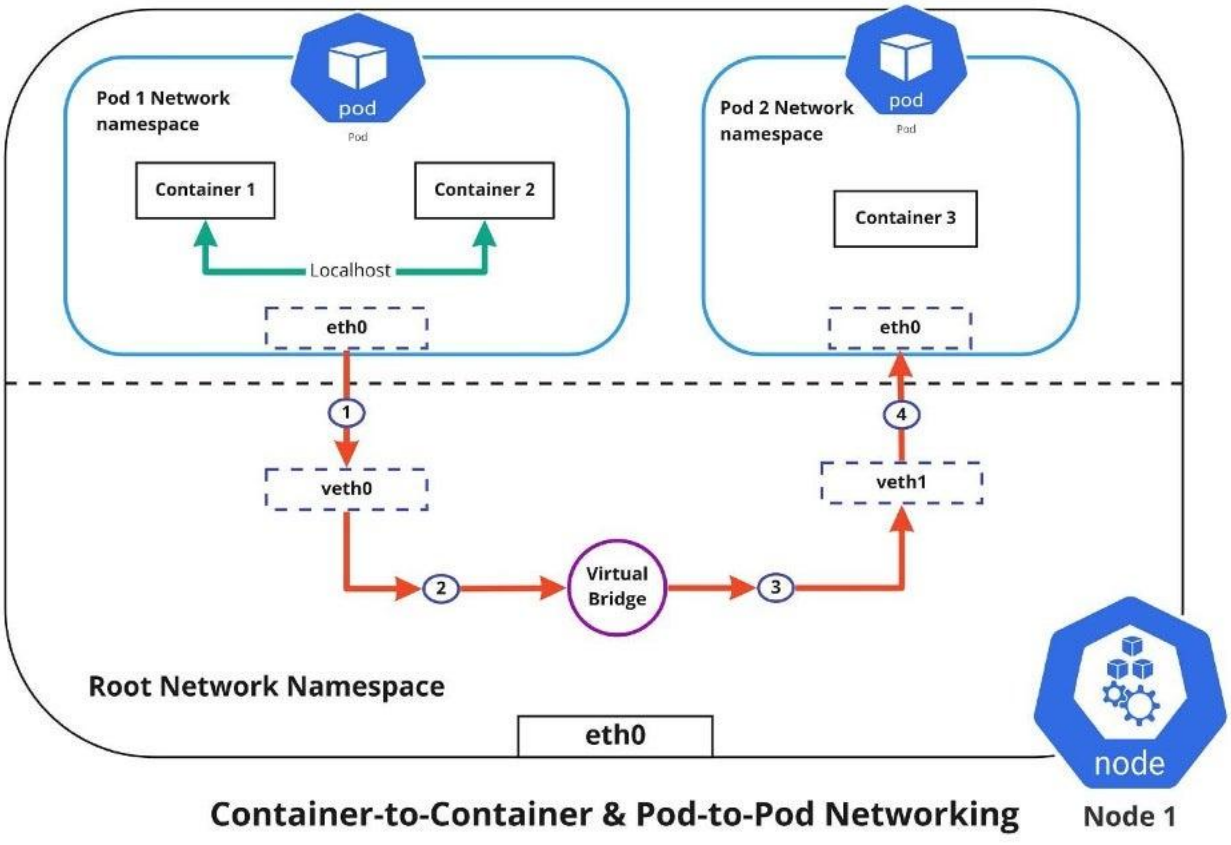
A Visual Guide to Kubernetes Networking Fundamentals

Written by Nived Velayudhan

What Kubernetes networking solves

Kubernetes networking is designed to ensure that the different entity types within Kubernetes can communicate. The layout of a Kubernetes infrastructure has, by design, a lot of separation. Namespaces, containers, and Pods are meant to keep components distinct from one another, so a highly structured plan for communication is important.



(Nived Velayudhan, CC BY-SA 4.0)

The Kubernetes Network Model has a few general rules to keep in mind:

1. Every Pod gets its own IP address: There should be no need to create links between Pods and no need to map container ports to host ports.
2. NAT is not required: Pods on a node should be able to communicate with all Pods on all nodes without NAT.
3. Agents get all-access passes: Agents on a node (system daemons, Kubelet) can communicate with all the Pods in that node.
4. Shared namespaces: Containers within a Pod share a network namespace (IP and MAC address), so they can communicate with each other using the loopback address.

Container-to-container networking

Container-to-container networking happens through the Pod network namespace. Network namespaces allow you to have separate network interfaces and routing tables that are isolated from the rest of the system and operate independently. Every Pod has its own network namespace, and containers inside that Pod share the same IP address and ports. All communication between these containers happens through localhost, as they are all part of the same namespace. (Represented by the green line in the diagram.)

Pod-to-Pod networking

With Kubernetes, every node has a designated CIDR range of IPs for Pods. This ensures that every Pod receives a unique IP address that other Pods in the cluster can see. When a new Pod is created, the IP addresses never overlap. Unlike container-to-container networking,

Pod-to-Pod communication happens using real IPs, whether you deploy the Pod on the same node or a different node in the cluster.

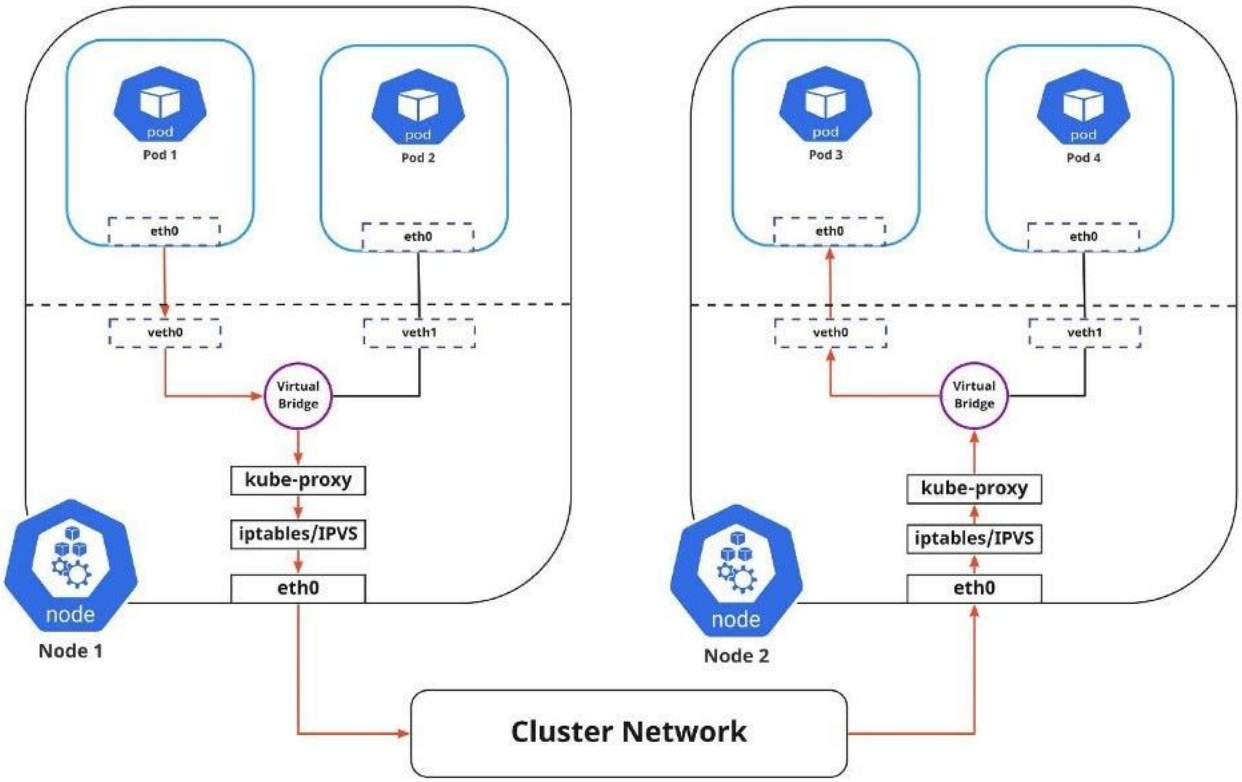
The diagram shows that for Pods to communicate with each other, the traffic must flow between the Pod network namespace and the Root network namespace. This is achieved by connecting both the Pod namespace and the Root namespace by a virtual ethernet device or a veth pair (veth0 to Pod namespace 1 and veth1 to Pod namespace 2 in the diagram). A virtual network bridge connects these virtual interfaces, allowing traffic to flow between them using the Address Resolution Protocol (ARP).

When data is sent from Pod 1 to Pod 2, the flow of events is:

1. Pod 1 traffic flows through eth0 to the Root network namespace's virtual interface veth0.
2. Traffic then goes through veth0 to the virtual bridge, which is connected to veth1.
3. Traffic goes through the virtual bridge to veth1.
4. Finally, traffic reaches the eth0 interface of Pod 2 through veth1.

Pod-to-Service networking

Pods are very dynamic. They may need to scale up or down based on demand. They may be created again in case of an application crash or a node failure. These events cause a Pod's IP address to change, which would make networking a challenge.



(Nived Velayudhan, CC BY-SA 4.0)

Kubernetes solves this problem by using the Service function, which does the following:

1. Assigns a static virtual IP address in the frontend to connect any backend Pods associated with the Service.
2. Load-balances any traffic addressed to this virtual IP to the set of backend Pods.
3. Keeps track of the IP address of a Pod, such that even if the Pod IP address changes, the clients don't have any trouble connecting to the Pod because they only directly connect with the static virtual IP address of the Service itself.

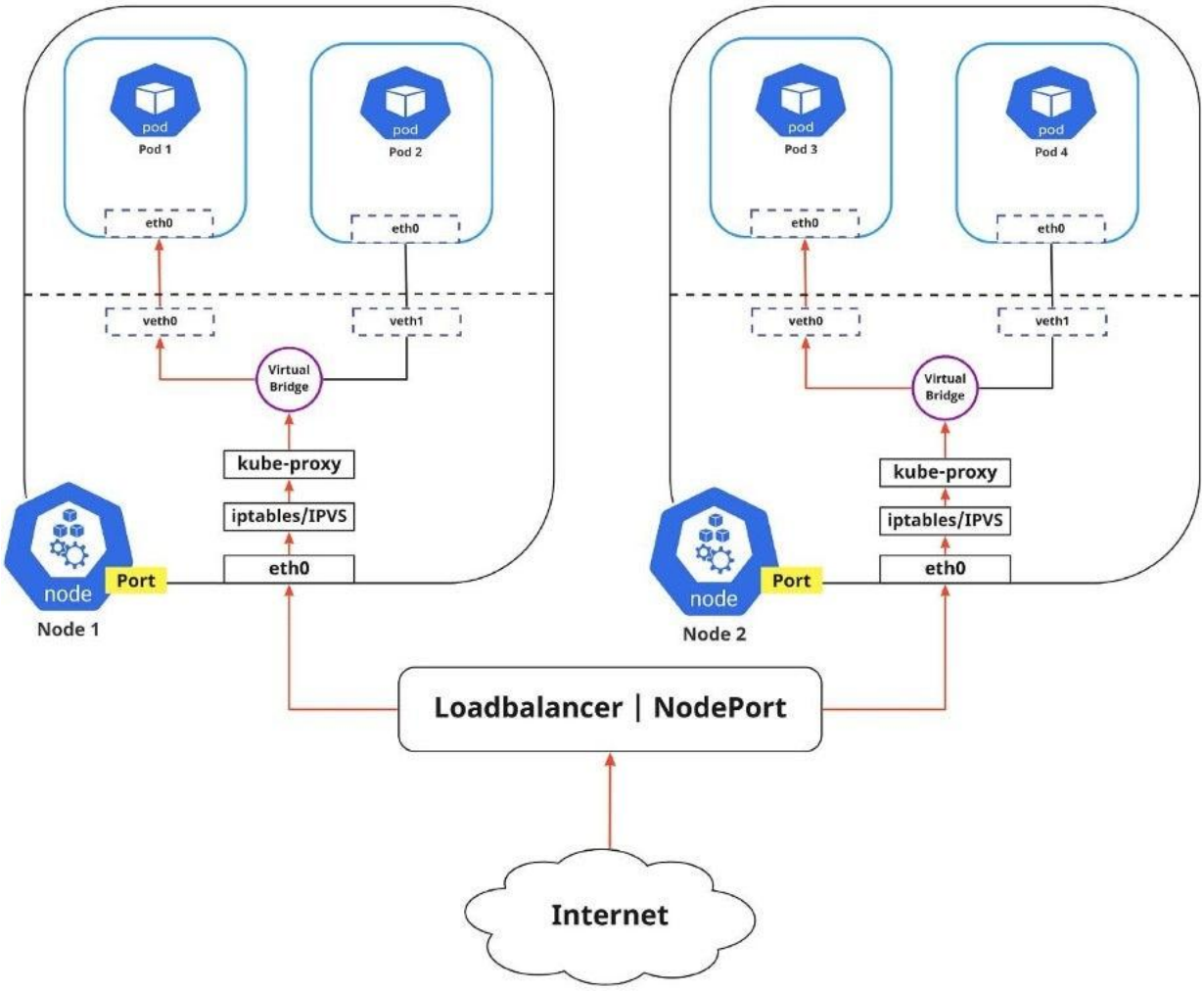
The in-cluster load balancing occurs in two ways:

1. IPTABLES: In this mode, kube-proxy watches for changes in the API Server. For each new Service, it installs iptables rules, which capture traffic to the Service's clusterIP and port, then redirects traffic to the backend Pod for the Service. The Pod is selected randomly. This mode is reliable and has a lower system overhead because Linux Netfilter handles traffic without the need to switch between userspace and kernel space.
2. IPVS: IPVS is built on top of Netfilter and implements transport-layer load balancing. IPVS uses the Netfilter hook function, using the hash table as the underlying data structure, and works in the kernel space. This means that kube-proxy in IPVS mode redirects traffic with lower latency, higher throughput, and better performance than kube-proxy in iptables mode.

The diagram above shows the package flow from Pod 1 to Pod 3 through a Service to a different node (marked in red). The package traveling to the virtual bridge would have to use the default route (eth0) as ARP running on the bridge wouldn't understand the Service. Later, the packages have to be filtered by iptables, which uses the rules defined in the node by kube-proxy. Therefore the diagram shows the path as it is.

Internet-to-Service networking

So far, I have discussed how traffic is routed within a cluster. There's another side to Kubernetes networking, though, and that's exposing an application to the external network.



(Nived Velayudhan, CC BY-SA 4.0)

You can expose an application to an external network in two different ways.

1. Egress: Use this when you want to route traffic from your Kubernetes Service out to the Internet. In this case, iptables performs the source NAT, so the traffic appears to be coming from the node and not the Pod.
2. Ingress: This is the incoming traffic from the external world to Services. Ingress also allows and blocks particular communications with Services using rules for connections. Typically, there are two ingress solutions that function on

different network stack regions: the service load balancer and the ingress controller.

Discovering Services

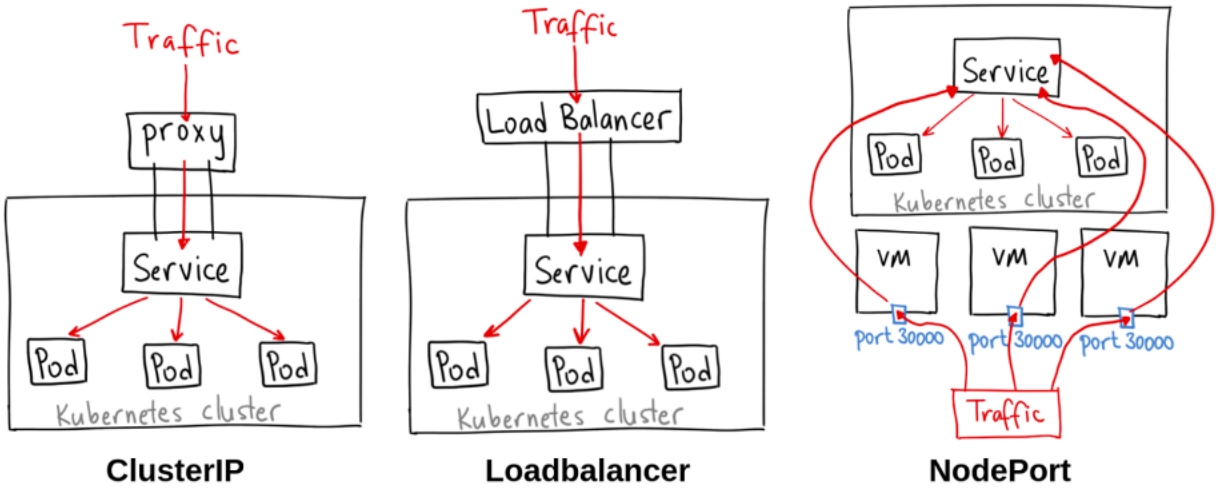
There are two ways Kubernetes discovers a Service:

1. Environment Variables: The kubelet service running on the node where your Pod runs is responsible for setting up environment variables for each active service in the format {SVCNAME}_SERVICE_HOST and {SVCNAME}_SERVICE_PORT. You must create the Service before the client Pods come into existence. Otherwise, those client Pods won't have their environment variables populated.
2. DNS: The DNS service is implemented as a Kubernetes service that maps to one or more DNS server Pods, which are scheduled just like any other Pod. Pods in the cluster are configured to use the DNS service, with a DNS search list that includes the Pod's own namespace and the cluster's default domain. 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 is enabled throughout your cluster, all Pods can automatically resolve Services by their DNS name. The Kubernetes DNS server is the only way to access ExternalName Services.

ServiceTypes for publishing Services:

Kubernetes Services provide you with a way of accessing a group of Pods, usually defined by using a label selector. This could be applications trying to access other applications within the cluster, or it could allow you to expose an application running in the cluster to

the external world. Kubernetes ServiceTypes enable you to specify what kind of Service you want.



(Ahmet Alp Balkan, CC BY-SA 4.0)

The different ServiceTypes are:

1. ClusterIP: This is the default ServiceType. It makes the Service only reachable from within the cluster and allows applications within the cluster to communicate with each other. There is no external access.
2. LoadBalancer: This ServiceType exposes the Services externally using the cloud provider's load balancer. Traffic from the external load balancer is directed to the backend Pods. The cloud provider decides how it is load-balanced.
3. NodePort: This allows the external traffic to access the Service by opening a specific port on all the nodes. Any traffic sent to this Port is then forwarded to the Service.
4. ExternalName: This type of Service maps a Service to a DNS name by using the contents of the externalName field by returning a CNAME record with its value. No proxying of any kind is set up.

Networking software

Networking within Kubernetes isn't so different from networking in the physical world, as long as you understand the technologies used. Study up, remember networking basics, and you'll have no trouble enabling communication between containers, Pods, and Services.