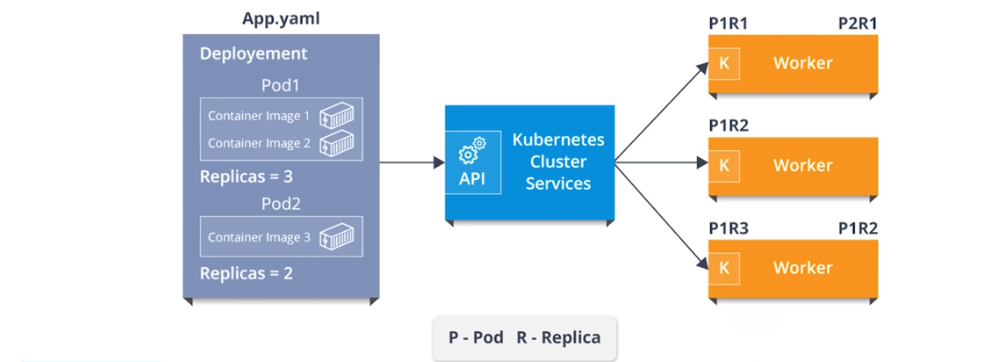
**Kubernetes Networking**

**Kubernetes Cluster**: Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit. The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines.

Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.

A Kubernetes cluster consists of two types of resources:

* The Master coordinates the cluster
* Nodes are the worker that run applications



**Kubernetes** [**Namespaces**](https://kubernetes.io/docs/tasks/administer-cluster/namespaces-walkthrough/)**:**

Namespace as a **virtual cluster** inside your Kubernetes cluster. You can have multiple namespaces inside a single Kubernetes cluster, and they are all **logically isolated from each other**. Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision. It can be as a virtual wall between multiple clusters.

Inside the same namespace **you can discover the other applications by service name**. The isolation namespaces provide allow you to **reuse the same service name in different namespaces**, resolving to the applications running in those namespaces. **This allows you to create your different “environments” in the same cluster** if you wish to do so. For development, test, acceptance and production you would create 4 separate namespaces

Kubectl get namespace

All objects such as pods, services, volumes, etc… are part of a namespace. If you do not specify a namespace when creating or viewing your objects, they will be created in the “default” namespace. When you want to interact with objects in a different namespace than “default”, you must pass the -n flag to kubectl

kubectl get pods -n kube-system

kubectl get pod --all-namespaces

There are 3 [Namespaces](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/) in k8s.

**Default**: The default namespace for objects with no other namespace

**Kube-system**: The namespace for objects created by the Kubernetes system

**Kube-public**: This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster.

**How to create custom namespace:**

kubectl create namespace test

kubectl delete ns test

vi deployment-namspace.yml

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: nginx-deployment

**namespace: prod**

spec:

selector:

matchLabels:

app: nginx

replicas: 2 # tells deployment to run 2 pods matching the template

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.9.1

ports:

- containerPort: 80

kubectl get deployment --all-namespaces

Similarly you create namespaces and assign to your **deployment, services, pod, replicas, replication controller** etc.

In[**Kubernetes Networking**](https://kubernetes.io/docs/concepts/cluster-administration/networking/)we have

* **Container to container communication**: two or more containers communication
* **Pod to Pod communication**: is the communication in between two different pods, having various images and replicas.
* **Pod-to-Service communication**: is how a service enables pod to communicate with any other pod
* **External-to-service Communication**: is how an external to service communicate which are from external network sources to cluster sources via ingress network.

### **How Does Kubernetes Networking Compared to Docker Networking?**

Kubernetes manages networking through CNI’s on top of docker and just attaches devices to docker. While docker with docker swarm also has its own networking capabilities, such as overlay, bridging, etc, the CNI’s provide similar types of functions.

[**Network**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/) **plugins in Kubernetes:**

**CNI plugins**: The CNI networking plugin supports pod’s ingress and egress traffic. The CNI plugin is selected by passing Kubelet the --network-plugin=cni

**Kubenet plugin**: Kubenet is a very basic, simple network plugin, on Linux only. It does not, of itself, implement more advanced features like cross-node networking or network policy. It is typically used together with a cloud provider that sets up routing rules for communication between nodes, or in single-node environments.

Kubenet creates a Linux bridge named cbr0 and creates a veth pair for each pod with the host end of each pair connected to cbr0

The kubenet plugin is selected by passing Kubelet the --network-plugin=kubenet

#### **Services**

* A service in Kubernetes is the entry for traffic into your application. It can be used for accessing an application just internally in the Kubernetes cluster or to expose the application.
* Basically, services are a type of resource that configures a proxy to forward the requests to a set of pods, which will receive traffic & is determined by the selector. Once the service is created it has an assigned IP address which will accept requests on the port.
* Now, there are various service types that give you the option for exposing a service outside of your cluster IP address.

**Types of Services**

There are mainly 3 types of services.

#### C**lusterIP:** This is the default service type which exposes the service on a cluster-internal IP by making the service only reachable within the cluster.

**NodePort:** This exposes the service on each Node’s IP at a static port. Since, a **ClusterIP**service, to which the NodePort service will route, is automatically created. We can contact the NodePort (range 30000–32767) service outside the cluster.

**LoadBalancer:**This is the service type which exposes the service externally using a cloud provider’s load balancer. So, the NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

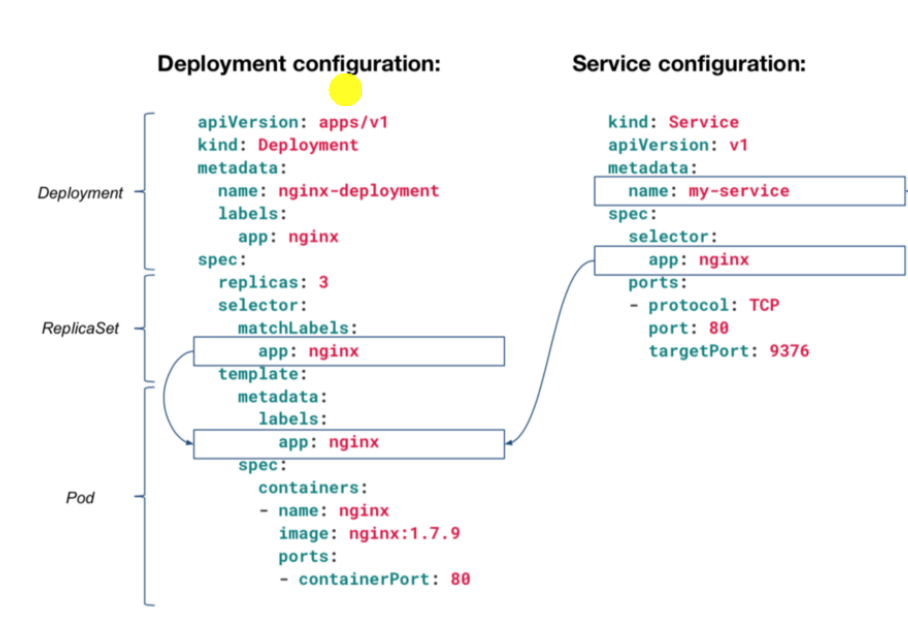
How do external services connect to these networks right?

**NodePort, LoadBalancer Ingress Network**

### **Ingress Network**

[Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) network is also an option for exposing services as it is a collection of rules that allow inbound connections that can be configured to give services externally through reachable URLs. **It supports URL based routing**. So, it basically acts as an entry point to the Kubernetes cluster that manages external access to the services in a cluster. **CNI** as an interface between network providers and **Kubernetes** networking

**How to map your services to Deployment:**



**Lab 1:**

Scenario: How can we assign a service to running deployment?

**Step 1:** Create a folder in your directory and change the working directory path to that folder

mkdir service-assignment

cd service-assignment

**Step2:** Now create deployment YAML files, for the web application

vi webapp.yml

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: webapp1

labels:

app: webapp-sql

tier: frontend

spec:

replicas: 1

selector:

matchLabels:

app: webapp-sql

tier: frontend

template:

metadata:

labels:

app: webapp-sql

tier: frontend

spec:

containers:

- name: webapp1

image: nginx:1.7.9

ports:

- containerPort: 80

Step3: Once you create the deployment files, deploy the applications.

kubectl apply -f webapp.yml

kubectl get deployment

Step 4: Now, you have to create services (NodePort) for the applications.

vi webservice.yml

apiVersion: v1

kind: Service

metadata:

name: webapp-sql

spec:

selector:

app: webapp-sql

tier: frontend

ports:

- port: 80

**type: NodePort**

kubectl apply -f webservice.yml

kubectl get service

Step 5: Now, check the configuration of running pods.

kubectl get pods

Lab 2: [Load Balancer](https://travix.io/deploying-your-application-to-kubernetes-2abaee6db222)

vi deployment-for-load-balancer.yml

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx-deployment

replicas: 2 # tells deployment to run 2 pods matching the template

template:

metadata:

labels:

app: nginx-deployment

spec:

containers:

- name: nginx-deployment

image: nginx:1.9.1

ports:

- containerPort: 80

vi service-for-loadbalancer.yml

apiVersion: v1

kind: Service

metadata:

name: nginx-deployment

spec:

selector:

**app: nginx-deployment**

ports:

- port: 80

**type: LoadBalancer**

<VM IP>:80 , would work within cluster.

For external, map svc port and ip with AWS ELB (Though we installed **kubernetes using kubeadmin** and haven’t mapped with [AWS](https://aws.amazon.com/blogs/opensource/kubernetes-ingress-aws-alb-ingress-controller/)/or any cloud so can’t get load balancer IP, we will get pending status for **EXTERNAL-IP**)

kubectl get svc nginx-deployment

**Service discovery**

Service discovery is the process of figuring out how to connect to a [service](http://kubernetesbyexample.com/service/). While there is a service discovery option based on [environment variables](https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/#environment-variables) available (disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies), the DNS-based (you can talk to the Service from any pod in your cluster, ) service discovery is preferable.

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

**Endpoints in Services:**

* Endpoints track the IP Addresses of the objects the service send traffic to.
* When a service selector matches a pod label, that IP Address is added to your endpoints

kubectl get svc

kubectl describe svc <service name>

kubectl get pods

kubeclt describe pod <pod name>