04/07/24

Kubernetes Networking

Container Networking Interface (CNI)

~~The CNI is a program~~ CNI defines how the network plugin should be ~~defined~~ developed and how container run times should invoke them. For every containerization solution the below is a common practice -

~~Bridge~~
1) Create Network Namespace
2) Create Bridge Network / Interface
3) Create VETH Pairs (Pipe, Virtual cable)
4) Attach vEth to Namespace
5) Attach other vEth to Bridge
6) Assign IP Addresses
7) Bring the interfaces up
8) Enable NAT - IP Masquerade

For container runtime, CNI specifies that it is

04/07/24

responsible for creating a network namespace for each container. It should then identify the networks. The container must attach to container runtime, must then invoke the plugin.

When a container is created using the 'ADD' command and also invoke the plugin when the container is deleted using the 'Del' command. It also specifies how to configure in network plugin on the container runtime environment using a JSON file.

On the plugin side, it defines that the plugin should support 'Add', 'Del' and 'check' command link arguments and that these should accept parameters like container and network namespace. The plugins should take care of assigning IP addresses to the pods and any associated routes required for the containers to reach the other containers in the network.

At the end the results should be specified in a particular format. As long as the container

04/07/24

runtime and plugins adhere to these standards, they can all live together. Any runtime should be able to work with any plugin.

CNI comes with a set of supported plugins already. Such as bridge, VLAN, IPVLAN, MACVLAN, one for Windows, as well as IPAM plugins like host-local and dhcp. There are other plugins available from third party organizations as well. Some examples are weave, flannel, cilium, Vmware NSX, calico, infoblox etc. All of these container runtimes implement CNI standards.

Docker doesn't implement CNI. Docker has its own set of standards known as CNM which stands for Container Network Model. Due to the differences, these plugins don't natively integrate with docker.

When kubernetes creates docker containers, it creates them on the 'None' networks. It then invokes the configured CNI plugins who takes care of the rest of the configuration.

04/07/24

Cluster Networking

The Kubernetes cluster consists of master and worker nodes. Each node must have at least 1 interface connected to a network. Each interface must have an address configured. The hosts must have a unique hostname set, as well as unique mac address.

There are some ports needs to be opened as well. These are used by the various components in the control plane. The master should accept connections on port 6443 for the api server. The worker nodes, kubectl tools, external users and all other control plane component access the kube-api server via this port. The kubelet on the master and worker nodes listen on 10250. The kubelet can be present in the master node as well. The kube-scheduler requires port 10259 to be open. The kube controller manager requires port 10257 to be open.

04/07/24

the worker node expose services for external access on ports 30000 to 32767. Finally the etcd server listens on port 2379.

For the multiple master nodes, all of these ports need to be open on those as well. An additional port 2380 open, so the etcd clients can communicate with each other.

Master node(s)

Protocol	Direction	Port Range	Purpose	Used by
TCP	Inbound	6443*	Kubernetes API Server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control Plane
TCP	Inbound	10259	kube-scheduler	Self
TCP	Inbound	10257	kube-controller-manager	Self

Worker node(s)

TCP	Inbound	~~Port~~ 10250	~~Purpose~~ Kubelet API	Self, Control Plane
TCP	Inbound	30000-32767	Node port Services	All

04/07/24

Pod Networking

Networking Model

→ Every pod should have an ip address.
→ Every pod should be able to communicate with every other pod in the same node.
→ Every pod should be able to communicate with every other pod on the other nodes without NAT.

Imagine there are three nodes. The nodes are part of an external network and has ip address in the 192.168.1.0 series. Node 1 is 192.168.1.11, Node 2 192.168.1.12, Node 3 192.168.1.13.

When containers are created, kubernetes create network namespaces for them. To enable communication between them, these namespaces to a network. The bridge network can be created within nodes to attach namespaces. Each bridge network will be on its own subnet. Any private ip address

04/07/24

range can be chosen say 10.244.1.0, 10.244.2.0, 10.244.3.0. Everytime a container is created it will get an ip address, and other necessary settings from the respective subnet using a script, lets call it net-script.sh.

Suppose a pod in node1, 10.244.1.2 wants to ping pod 10.244.2.2 on node2. As of now, the first has no idea where the destination address is. It routes to node1 ip, as it is said to be default gateway. Add a route to node1's routing table to route traffic to 10.244.2.2 where the node2's ip address at 192.168.1.12.

ip route add 10.244.2.2 via 192.168.1.12

Once the route is added, pods of the node2 is reachable. Similar tasks for all other pods in all other nodes.

This approach is fruitful for a simple setup. But when the network gets complicated it won't works well. Instead of having to configure routes on each server, a better solution is to do that on

a router ~~if you~~ and point all hosts to use that as the default gateway. ~~So it~~ With that, the individual virtual networks were created with the address 10.244.1.0/24 on each node, now from a single large network with the address 10.244.0.0/16.

How to do everything in an automated way?

CNI does the tricks.

CNI tells kubernetes that this is how you should call a script ~~as~~ as soon as you create a container, and CNI tells us 'this is how the script should looks like'. For CNI standard, add (add) ADD) and DEL) (delete) sections. The container runtime on each node is responsible for creating containers. Whenever a container is created, the container runtime looks at the CNI configuration passed as a command-line argument when it ~~was~~ runs and identifies the script's name. It then looks in the CNI's bin directory to find the script

04/07/24

and then executes the script with the "Add" command and the name and the namespace of the container, and then the script takes care of the rest.

CNI in Kubernetes

CNI plugin must be invoked by the component within Kubernetes that is responsible for creating containers. Because that component must then invoke the appropriate network plugin after the container is created. The component that is responsible for creating containers is the container runtime. The two good examples of are containerD and cri-o.

How to use network plugins?

The network plugins are all installed in the directory '/opt/cni/bin'. The plugin will be used is located in '/etc/cni/net.d'. There may be multiple configuration files in this directory. ~~In~~

In the '/opt/cni/bin', it has all the supported CNI plugins as executables. The '/etc/cni/net.d' has the set-up configuration files.

04/07/24

If ≠ in '/etc/cni/net.d' has multiple configuration files, it will choose according to the ~~or~~ alphabetical order.

An example bridge configuration file

10-bridge.conf

```
{ "cniVersion": "0.2.0",
  "name": "mynet",
  "type": "bridge",
  "bridge": "cni0",
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
      "type": "host-local",
      "subnet": "10.22.0.0/16",
      "routes": [
          { "dst": "0.0.0.0/0" }
      ]
  }
}
```

IPAM CNI

CNI comes with two built in plugins to which the tasks of ipam is outsourced. One of the plugin is host local plugin. It is still the responsibility of the admin to invoke the plugin in the script. The script can be made dynamic to support different kinds of plugins. The cni configuration file has a section called ipam in which it can be specified the type of plugin to be used, the subnet

04/07/24

and the route to be used.

The weave cidr is 10.32.0.0/12.

Service Networking

If a pod needs to access services hosted on another pod, a service is always used.

For example, ~~iso~~ Node 1, Node 2 and Node 3 are three nodes. Node 1 has an orange service. The orange service gets an ip address and a name assigned to it. The blue pod in node 1 can now access the orange pod in node ~~1~~ through the orange services ip or its name.

What about access from other pods from other nodes?

When a service is created, it is accessible from all parts of the cluster, irrespective of what nodes the pods are on. While a pod is hosted on a node, a service is hosted across the cluster. E.

The service is only accessible from within the cluster. This type of service is known

04/09/24

as ClusterIP.

If the orange pod was hosting a db that is to be only accessed from within the cluster, then this type of service works just fine. For instance, the purple pod ~~was~~ in node 2 was hosting a web application. To make the app on the pod accessible outside the cluster, another service is created which is NodePort. This service also gets an ip address assigned to it and works just like ClusterIP, ~~as~~ as in all the other pods can access this service using it's ip. But in addition, it also exposes the application on a port on all nodes in the cluster. That way external users on application have access to the service.

For example, there are three nods in the cluster.

Similar to kube-scheduler, kube-proxy watches the changes in the cluster ~~thro~~ through kube-api server, and everytime a new service is to be created, kube-proxy gets into action.

Unlike pods, services are not created on each

node or assigned to each node. Services are a cluster wide concept. They exist across all the nodes in the cluster. There is no server or service really listening on the ip of the service. There are no processes or namespaces, or interfaces for a service. It's just a virtual object.

When a service object is created in kubernetes, it is assigned an ip address from a predefined range. The kube-proxy components running on each node, gets that ip address and creates forwarding rules on each node in the cluster, saying any traffic coming to this ip, the ip of the service should go to the ip of the pod.

Once that is in place, whenver a pod tries to reach the ip of the service, it is forwarded to the pods ip which is accessible from any node in the cluster. Whenever services are created or deleted the kube-proxy component creates or deletes these rules. Kubeproxy supports different ways, such as userspace where kube-proxy

04/07/24

listens on a port for each service and proxies connections to the pods. By creating ipvs rules or the third and the default option and the one familiar to us is using iptables. The proxy mode can be set using the proxy mode option while configuring the kube-proxy service. If this is not set, it defaults to iptables.

```
kubectl get service
```

To determine the service ip range.

```
kube-api-server --service-cluster-ip-range ipNet
```

```
ps aux | grep kube-api-server
```

To see the iptables rule for a particular service

```
iptables -L -t nat | grep db-service
```

This is done by adding a DNAT rule to iptables.

Similarly, when NodePort, kube-proxy creates iptables rules to forward all traffic coming on a port on all nodes to the respective backend pods.

Kube-proxy creates these entries in the

04/07/24

kube-proxy logs itself. In the logs ~~/var/log/~~ '/var/log/kube-proxy.log', can find what proxier it uses. In this case, its iptables and the 'Add' an entry when it added a new service for the db.

Cluster DNS

There are 3 nodes in the cluster.

kubernetes deploys a built-in dns server by default when a cluster is set-up. If the kubernetes is set manually, then it has to be done by oneself.

In two different nodes there is test pod and web pod and a web service. To make a web server accessible to test pod a service named web-service is created. The service gets an ip address 10.107.37.188.

Whenever a service is created, the kubernetes dns service creates a record for the service. It maps the service name to the ip address.

04/07/24

Hostname	IP Address
web-service	10.107.37.188

So within the cluster any pod can now reach the service using its service name. In this case, since the test pod and the web pod and its associated service are all in the same namespace, the default namespace.

Let's assume the web service is in a separate namespace name 'apps'. Then it refer it from the default namespace, I have to use 'web-service.apps'. The last name of the service is now the name of the namespace.

web-service.apps
- web-service: service name
- apps: name of the namespace

The dns server creates a subdomain. All the services are grouped together into another sub-domain called 'SVC'.

For each namespace, the dns server creates a subdomain with its name. All pods and services

04/07/24

for a namespace are thus grouped together within a subdomain in the name of the namespace. All the services are further grouped together into another subdomain called 'svc'. So to reach the web-service use the following name-

web-service.apps.svc

Finally, all the services and pods are grouped together into a root domain for the cluster, which is set to 'cluster.local' by default.

web-service.apps.svc.cluster.local

This is the FQDN.

Records for pods are not created by default. It can be enabled explicitly. So records for pods are created as well. It doesn't use the pod name. For each pod Kubernetes generates a name by replacing ('.') in the ip address with dashes. The namespace remains the same and type is set to pod. The root domain is always cluster.local.

04/07/24

10-244-2-5.apps.pod.cluster.local

Similarly, the test pod in the default namespace, gets a record in the DNS server, with its ip converted to a dashed hostname. This

10-244-1-5.apps.pod.cluster.local

This resolves to the ip address of the pod.

IP Address	Namespace	Type	Root	Hostname
10.244.2.5	apps	pod	cluster.local	10-244-2-5
10.244.1.5	default	pod	cluster.local	10-244-1-5

05/07/24

CoreDNS

There are two pods with two ip addresses. For pods it forms hostnames by replacing dots with dashes in the ip address of the pod.

Kubernetes implements dns in the same way. It deploys a dns server within the cluster. Prior to version 1.12, the dns implementation was known as Kube-dns. With Kubernetes version 1.12 the

05/07/24

The recommended dns server is coreDNS.

The coreDNS server is deployed as a pod in the kube-system namespace in the kubernetes cluster. They are deployed as two pods for redundancy as part of a replicaset. This pod runs the coreDNS executable, the same executable ~~when~~ is used when the coreDNS is deployed.

coreDNS requires a configuration file. The file is located in '/etc/coredns' directory. It uses a file name 'corefile'.

cat /etc/coredns/corefile

```
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
       pods insecure
       upstream
       fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    proxy . /etc/resolv.conf
    cache 30
    reload
}
```

05/07/24

Within this file, a number of plugin is ~~installed~~ configured.

Plugins are configured to handle errors, reporting health, monitoring metrics, cache etc. The plugin that makes coreDNS to work with Kubernetes is 'kubernetes' plugin. This is where the top level domain name for the cluster is set. So every record in the coreDNS server in this case cluster.local falls under this domain. Within Kubernetes plugin there are ~~multip~~ multiple options. The 'pod' option is responsible for creating a record for pods in the cluster. The conversion of ip addresses '.' to '-' is disable by default. But it can be enabled.

Any record that coreDNS server can't solve, for example say a pod tries to reach 'google.com', it is forwarded to the nameserver specified in the coreDNS pods '/etc/resolv.conf' file.

05/07/24

The '/etc/resolv.conf' file is set to use the nameserver from the kubernetes node. Also note that, this core file is passed into the pod has a configMap object.

kubectl get configMap -n kube-system

So if it is needed to modify configuration, the configMap object can be edited.

The coreDNS watches the kubernetes cluster for new pods or services, and everytime a pod or a service is created, it adds a record for it in its database.

Next step is for the pod to point to the coreDNS server. When the coreDNS solution is created, it also creates a service to make it available for other components within a cluster. The service is named as kube-dns by default. The ip address of this service is configured as the name server on the pods. The dns conf.

05/07/24

on pods are done by kubernetes automatically when the pods are created. The kubelet is responsible for this action. In the config file of the kubelet, there is ip of the dns server and domain in it. (/var/lib/kubelet/config.yaml).

Once the pods are configured with the right nameserver, it can resolve other pods and services. The 'resolv.conf' file also has a search entry which is set to 'default.svc.cluster.local', 'svc.cluster.local' and 'cluster.local'. This allows to find the service with FQDN.

06/07/24.

Ingress

~~Upto 6 minutes 39 seconds~~

Ingress helps users access an application using a single externally accessible URL. This URL can be configured to route to different services within the cluster based on the URL path, at the same time, implement SSL security as well.

Ingress can be assumed as layer 7 load balancer.

in the kubernetes cluster. It can be configured using native kubernetes primitives, just like any other object in kubernetes.

Even with ingress, it is still needed to be exposed to make it accessible outside the cluster. So, either publish it as a NodePort or with a cloud-native load balancer. But this configuration is just one-time configuration. Going forward the load balancing authentication, SSL and URL based routing configurations on the ingress controller.

Without ingress, nginx, HA proxy, traefik are deployed and configure them to route traffic to other services.

Ingress is implemented in kind of the same way. First deploy a supported solution, which happens to be nginx, HA proxy or traefik and then specify a set of rules to configure ingress. The solution deployed is called an ingress controller.

The set of rules configured are called ingress resources. Ingress resources are created using definition files like the ones are used to create pods, deployments and services.

A kubernetes cluster doesn't come with ingress controller by default.

Ingress Controller

There are number of solution available for ingress, a few of them being GCE (Googles layer seven HTTP load balancer), Nginx, Contour, HAProxy, Traefik, Istio. Out of these GCE and Nginx are currently supported and maintained by the kubernetes project.

The ingress controllers are not just another load balancer or nginx server. The load balancer component are just a part of it. The ingress controllers have additional intelligence built into them to monitor the kubernetes cluster for new definitions or ingress resources and configure the nginx server accordingly.

08/07/24

An nginx controller is deployed as just another deployment in Kubernetes.

nginx-ingress-definition.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-controller
                 /nginx-ingress-controller:0.21.0
          args:
            - /nginx-ingress-controller
            - --configmap=$(POD_NAMESPACE)/nginx-configuration
```

(same level as args)

```
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldpath: metadata.
                             namespace
          ports:
            - name: http
              containerPort: 80
            - name: https
              containerPort: 443
```

06/07/24

nginx

This image is particularly built for ingress in the kubernetes. It has its own set of requirements. Within the image, the Nginx program is stored at location Nginx ingress controller. So it must pass that as the command to start the nginx controller service.

Nginx has a set of configuration options such as the path to store logs, keep-alive threshold, SSL settings, session timeouts etc. In order to decouple these configuration data from the nginx controller image, first create a config map object and pass that in.

nginx-configmap.yaml

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

06/07/24

The configmap doesn't need't have any entry at this point, a blank object will do. Must pass two environment variables that carry the pod's name and namespace, it is deployed to. The nginx service requires these to read the configuration data from within the pod.

Finally, specify the ports used by the ingress controller, which happens to be 80 and 443. After this, a service is needed to expose the ingress controller.

nginx-ingress-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
```

(same level as ports)

```
  selector:
    name: nginx-ingress
```

06/07/24

Ingress controller has additional intelligence ~~this~~ built into them to monitor the Kubernetes cluster for ingress resources and configure the underlying Nginx server when something is changed. But for the ingress controller to do this, it requires a service account with the right set of permissions. So create service deployment files with correct role and role bindings.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-ingress-serviceaccount
```

Ingress Resources

An ingress resources is a set of rules and configurations applied on the ingress controller. For example, simply forward all incoming traffic to a single application or route traffic to different applications based on the URL; route users based on the domain name itself.

The ingress resource is created with a Kubernetes definition file.

06/07/24

Ingress-wear.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  backend:
    serviceName: wear-service
    servicePort: 80
```

The backend section defines where the traffic will be routed to.

```
kubectl create -f Ingress-wear.yaml
```

```
kubectl get ingress
```

Rules are used to route traffic. Create a rule definition file. The rules here is to handle all

06/07/24

traffic coming to "my-online-store.com" and route them based on the URL path. For this, a single rule is enough, since only handling traffic to a single domain name, which is 'my-online-store.com'. Under 'rules' there 'http' rule where different paths are specified. ~~Paths~~ 'paths' is an array of multiple items, one path for each URL.

Use the 'backend' section from the ~~ngx~~ ingress resource file (ingress-wear.yaml).

(ingress rule & definition file)

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch

spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80

      -path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

06/07/24

Know about the ingress resource

```
kubectl describe ingress ingress-wear-watch
```

Output:

```
Rules:
  Host        Path    Backends
  ----        ----    --------
              /wear
              /watch
```

To split traffic using domain name use 'host' field. The host field in each rule matches the specified value with the domain name used in the requested URL and routes traffic to the appropriate backend.

Ingress-wear-watch.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch

spec:
  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend
          serviceName: watch-service
          servicePort: 80
```

06/07/24

If the hostname is not mentioned, it will consider it as '*' (all), means accept all the incoming traffic through that particular rule without matching the hostname.

—o—

Ingress Controller Rewrite Target

The Nginx ingress controller can rewrite URLs.
There are two applications - watch and wear. These applications are accessed via 'http://<domain>/'.
I want those to be accessed using

http://<domain>/watch; http://<domain>/wear;

However, the applications themselves dont recognize the '/watch' or '/wear' paths. Without URL rewriting, a user's request to 'http://<domain>/watch' would attempt to access the mentioned URL, which would result http error 404.

To solve this, the rewrite-target option of the nginx ingress controller. This allows to modify the incoming URL path before forwarding the

06/07/24

request to the backend service. Essentially, replace '/watch' and '/wear' with '/'.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  namespace: critical-space
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /

spec:
  rules:
  - http:
      paths:
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80

      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
```

06/07/24

In this example,

→ The annotation 'nginx.ingress.kubernetes.io/rewrite-target: /' specifies that the incoming path should be rewritten to '/'.

→ The paths '/watch' and '/wear' are defined under 'rules -> http -> paths'.

→ Requests to 'http://<domain>/watch' are forwarded to the 'watch-service' backend service at port 80, with the path rewritten to '/'.

→ Similarly, requests to 'http://<domain>/wear' are forwarded to the 'wear-service' backend service at port 80, with the path rewritten to '/'.

More advanced Example

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: dynamic-rewrite-ingress
  namespace: critical-space
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
```

06/07/24

```
spec:
  rules:
  - http:
      paths:
      - path: /watch(/|$)(.*)
        backend:
          serviceName: watch-service
          servicePort: 80
      - path: /wear(/|$)(.*)
        backend:
          serviceName: wear-service
          servicePort: 80
```

Path: '/watch(/|$)(.*)'

This matches '/watch' followed by an optional '/' or the end of the string, and captures everything after '/watch' in the second group '(.*)'.

Rewrite Target: '/$2' where '$2' is the second captured group.

wear/legg/shirt

If the incoming request, 'http://example.com/watch/view [movie]

06/07/24

→ The path ~~/wear(/1$).*~~ '/wear(/1$)(.*)' matches '/wear/buy/shirt'.

→ The '(.*)' captures 'buy/shirt', which becomes '$2'.

→ The rewrite target '/$2' replaces '/wear/buy/shirt' with 'buy/shirt'.

→ The request is forwarded to the 'wear-service' backend service with the path '/buy/shirt'.

—o—

07/07/24

Application Lifecycle Management

Rolling Updates and Rollbacks

When a deployment is first created, it triggers a rollout. A new rollout creates a new deployment ~~version~~ revision. In the future, when the app is upgraded meaning the container version is updated to a new one, a new rollout is triggered and a new deployment revision is created, named revision2. This helps to keep track of the changes made to the

07/09/24

deployment. It also enables to rollbacks to a previous version of deployment if necessary.

To see the rollout status

```
kubectl rollout status deployment/myapp-deployment
```

Rollout history

```
kubectl rollout history deployment/myapp-deployment
```

Deployment Strategy

There are two types of deployment ~~strategy~~ strategies.

For example, there are five replicas of our web application instance deployed. One way to upgrade these to a newer version is to destroy all of these and then create newer versions of app instances. Meaning, first destroy the five running instances, and then deploy five new instances of the new app version.

The problem with this, during the period after the older versions are down and before any newer version is up, the app is down and inaccessible to users.

This strategy is known as the Recreate strategy.

07/07/24

Rolling Update

The second strategy is, to take down the older version and bring up a newer version one by one. This way the app never goes down and the upgrade is seamless.

If any strategy is not specified while creating the deployment, it will assume it to be rolling update.

The update can be different things such as updating app version, updating the version of docker containers used, updating their labels, updating the number of replicas etc. Once the necessary changes are made into the deployment file, use the necessary command to apply the changes.

```
kubectl set image deployment/myapp-deployment \ nginx-
container = nginx: 1.9.1
```

'kubectl set image' command will update the image of the container.