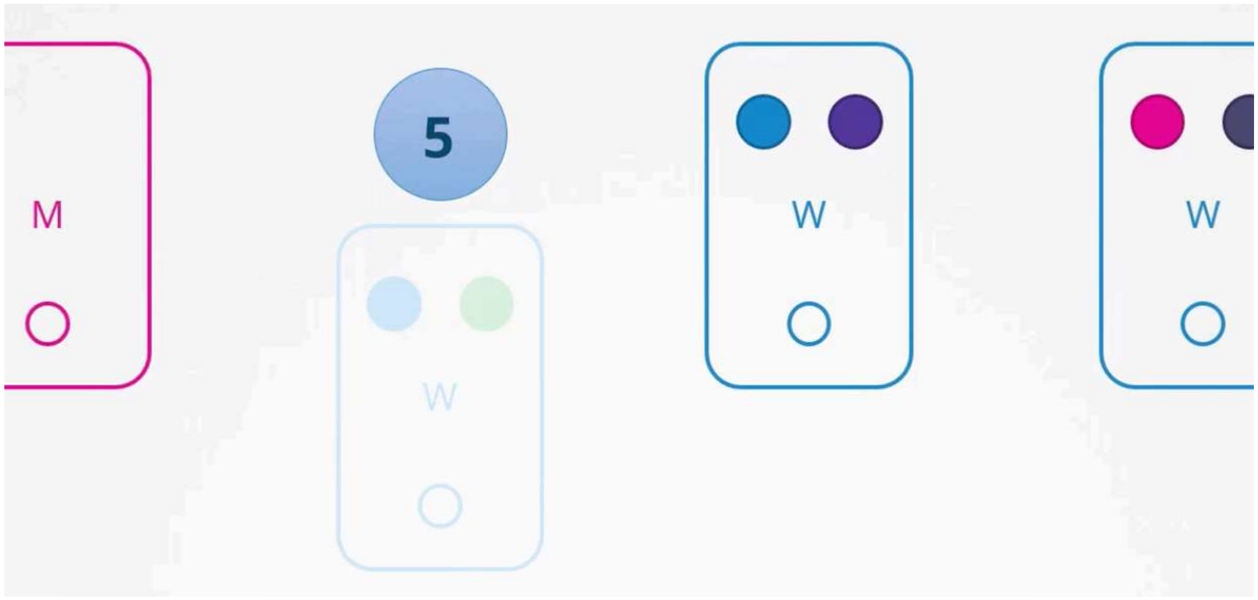
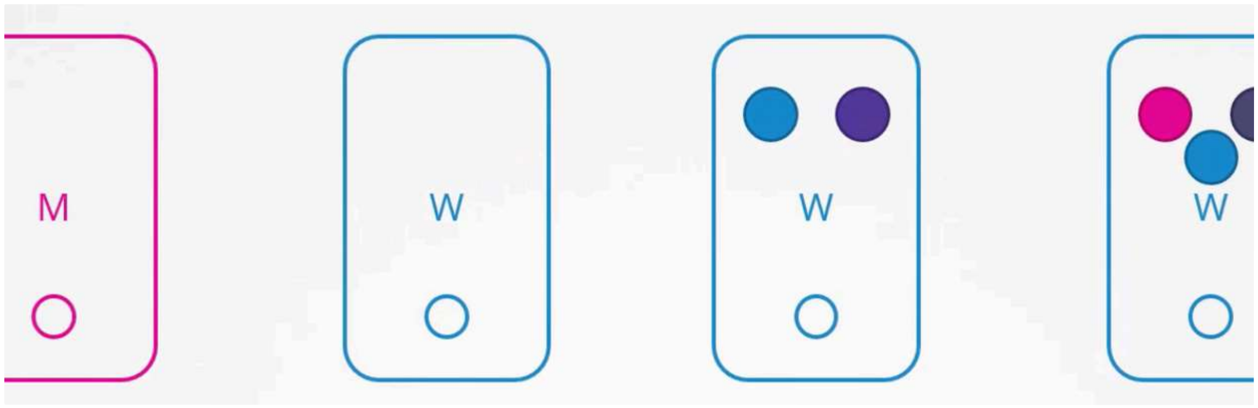
Unhealthy Nodes

If a node in the cluster goes down, the `kube-controller-manager` waits for **5 mins (default, max)** for the node to come back online. If the node comes back online within 5 mins, the pods running on it are restarted and then everything works the same way.



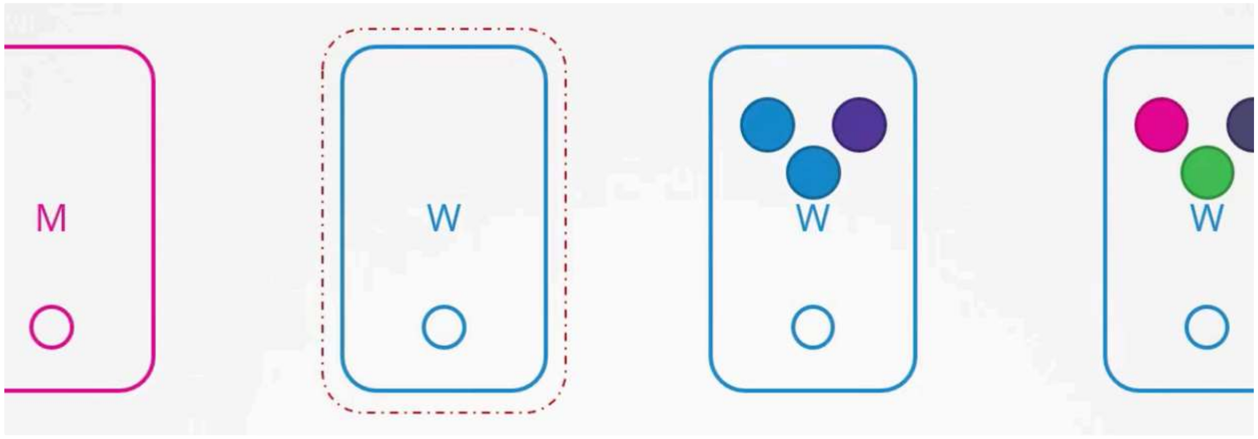
If the node doesn't come back online within 5 mins, it is considered unhealthy and all the pods running on it that were associated with replicasets are spawned on other nodes. Any pod on that node that was not associated with any replicaset dies with the node.

The time for which the `kube-controller-manager` waits before declaring a node unhealthy is called **pod eviction timeout** and it is configured in the `kube-controller-manager`. If the node comes back online after the pod eviction timeout, it has no pod running on it.

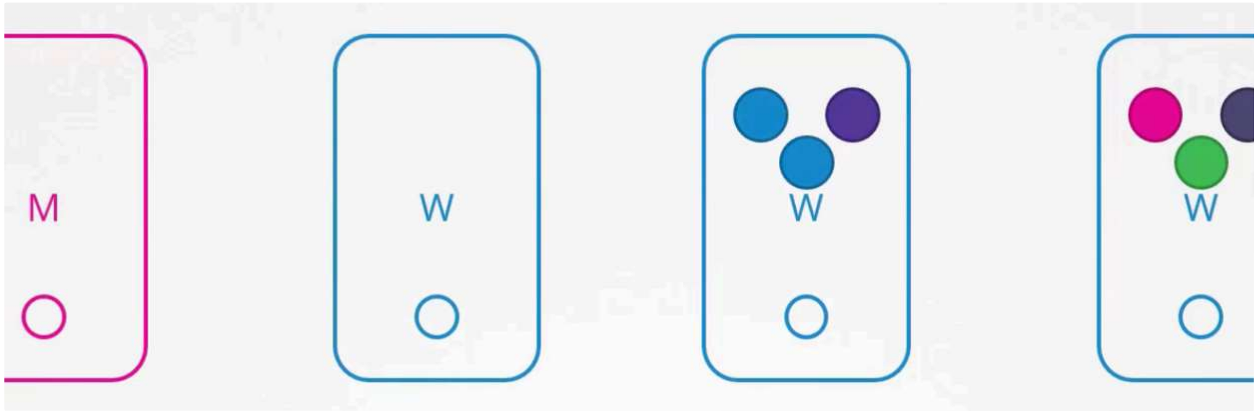


OS Upgrades

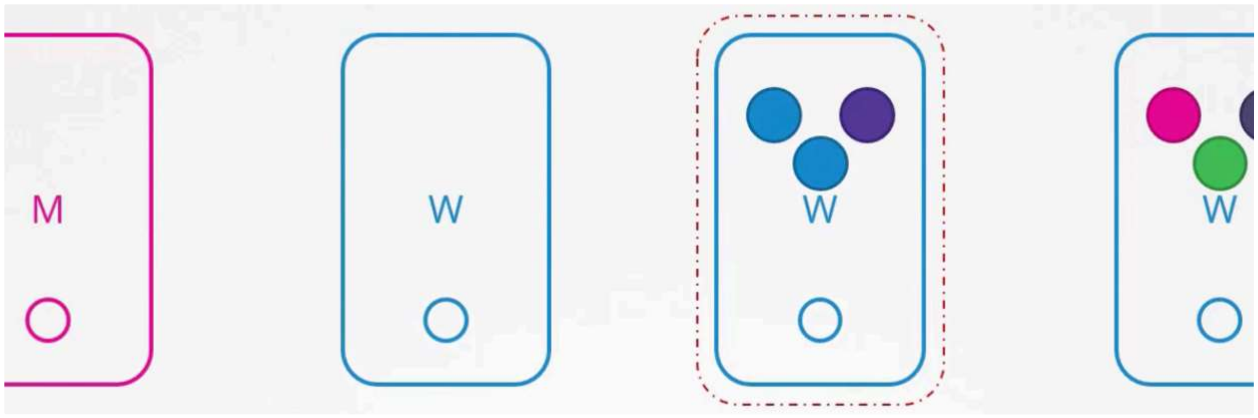
To perform OS upgrade or maintenance on a node, first drain it using `k drain <node-name>`. This will terminate the running pods on the node, spawn them on other nodes and mark the node as **cordon**, meaning no pods can be scheduled on it. The above command will not work if there are pods on the node that are not associated with any replicaset, as these pods will not be spawned on any other node. We'll have to use `--force` option to terminate such pods forcefully.



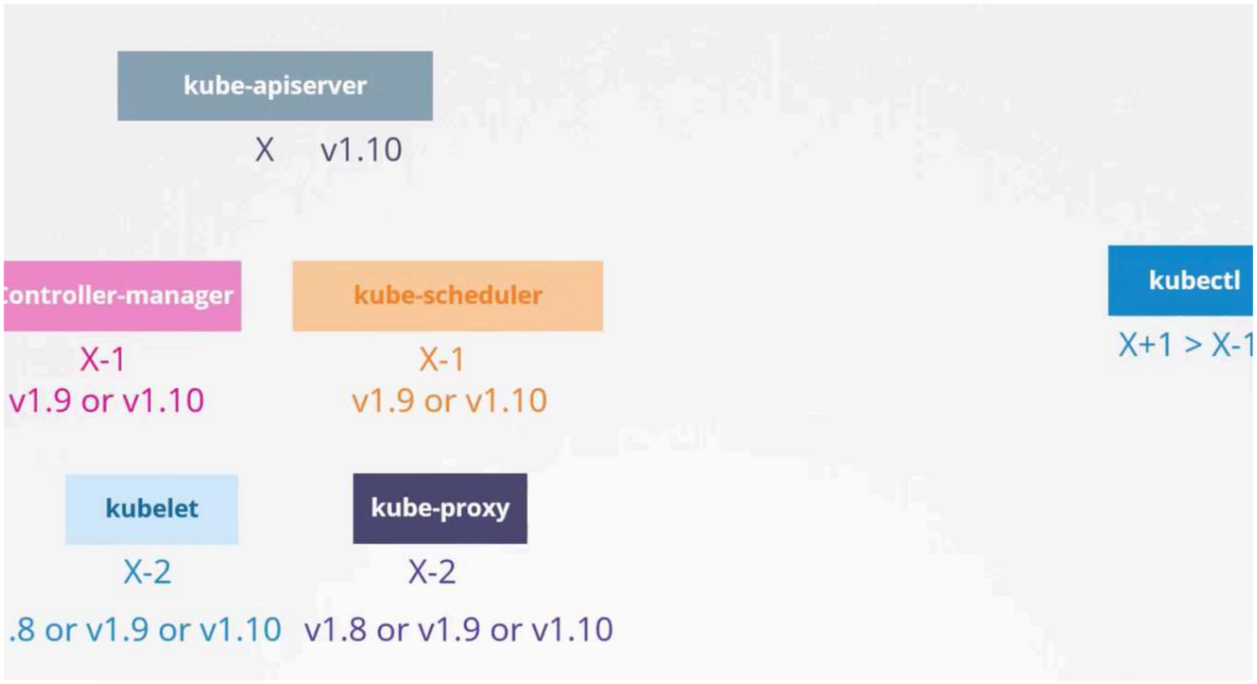
Now, perform the upgrade or maintenance. After that, uncordon the node using `k uncordon <node-name>`. This will make the node schedulable again.



We can manually cordon a node to prevent new pods from being scheduled on it by running `k cordon <node-name>`. This does not remove already running pods on the node.


M
W
W
W

Cluster Version Upgrade

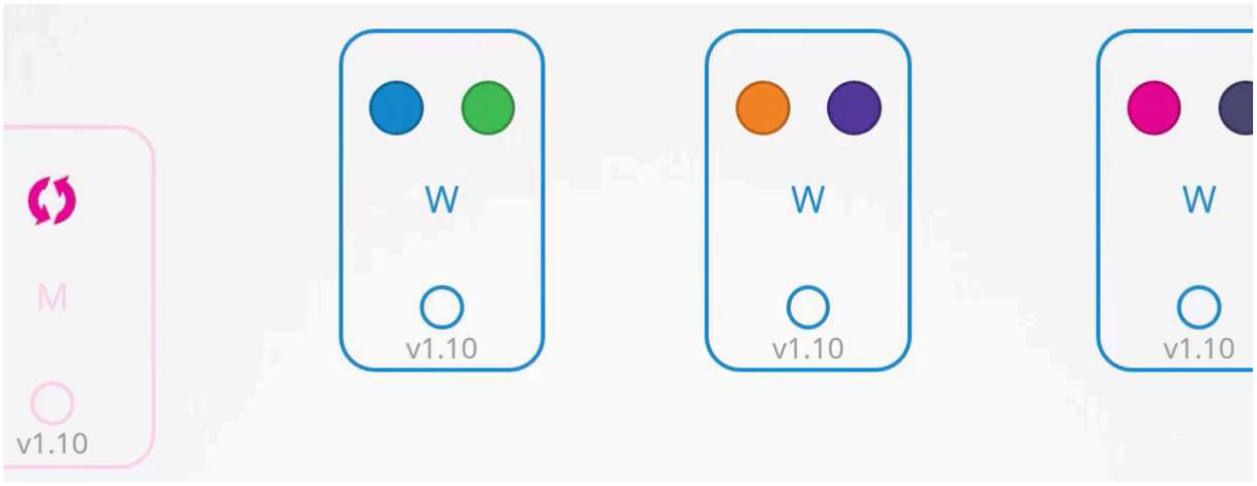


- When we download a K8s release, the 5 core components shown on the left in the image above are at the same version.
- The K8s components can be at different versions. The `kube-apiserver` must be at the highest version compared to other components, except `kubectl` which can be one minor version above.
- Upgrading the version of a cluster basically means upgrading the version of various K8s components running in the cluster.

- **K8s supports 3 latest minor versions.** If the current version of a K8s component is unsupported, we should upgrade it one minor version at a time.
- Upgrading the cluster on a managed K8s engine like GKE is very easy and takes only a few clicks.
- `k get nodes` command shows the version of `kubelet` running on each node.

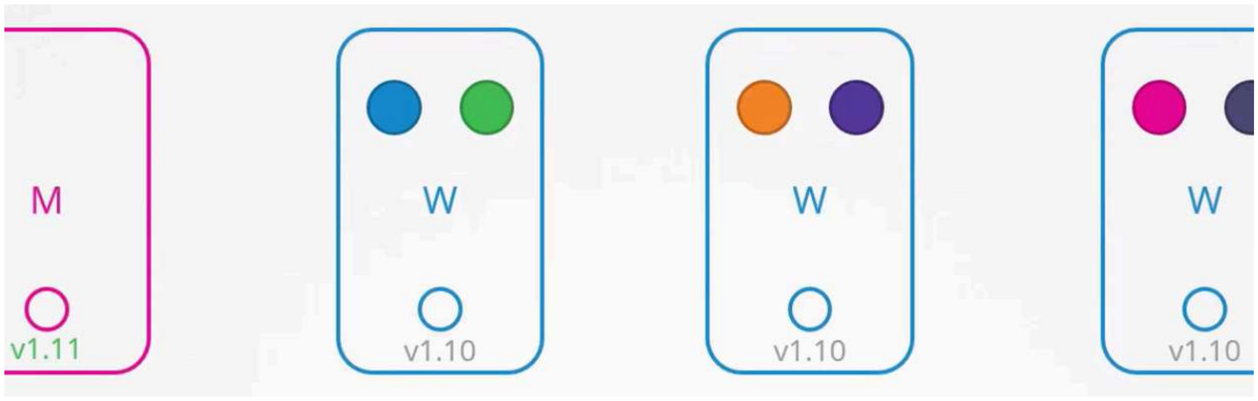
Upgrading the master node

First the master node is upgraded, during which the control plane components are unavailable. The worker nodes keep functioning and the application is up. While the master node is getting updated, all management functions are down. We cannot run `kubectl` commands as `kube-apiserver` is down. If a pod were to crash, a new one will not be spawned as the `kube-controller-manager` is down.



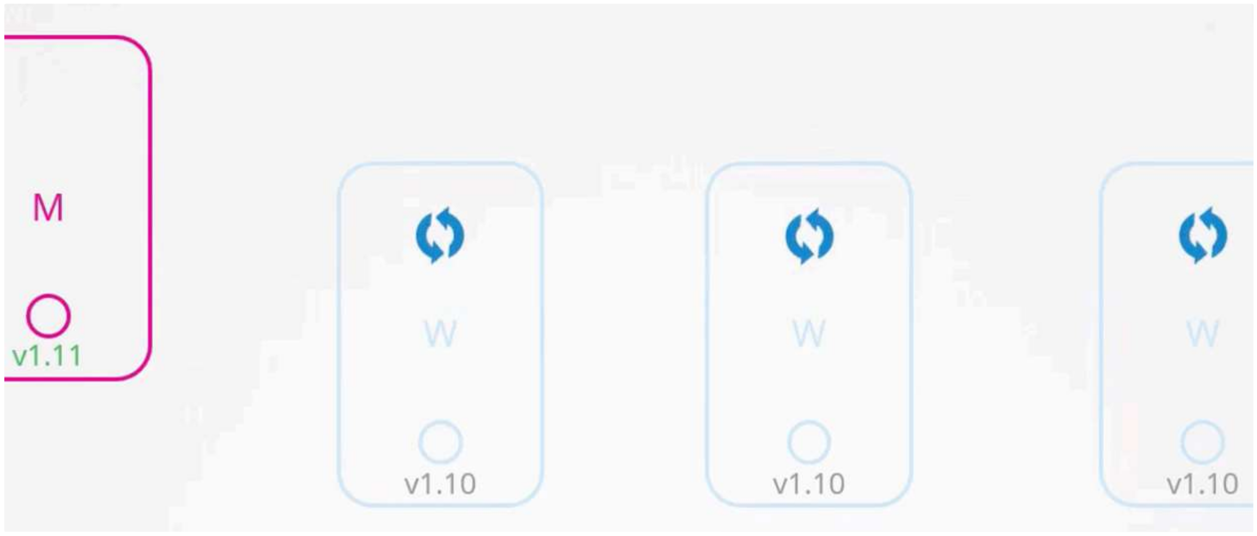
Upgrading worker nodes

Once the master node has been upgraded, we need to upgrade the worker nodes (upgrade the k8s components running on them). As the worker nodes serve traffic, there are various strategies to upgrade them.



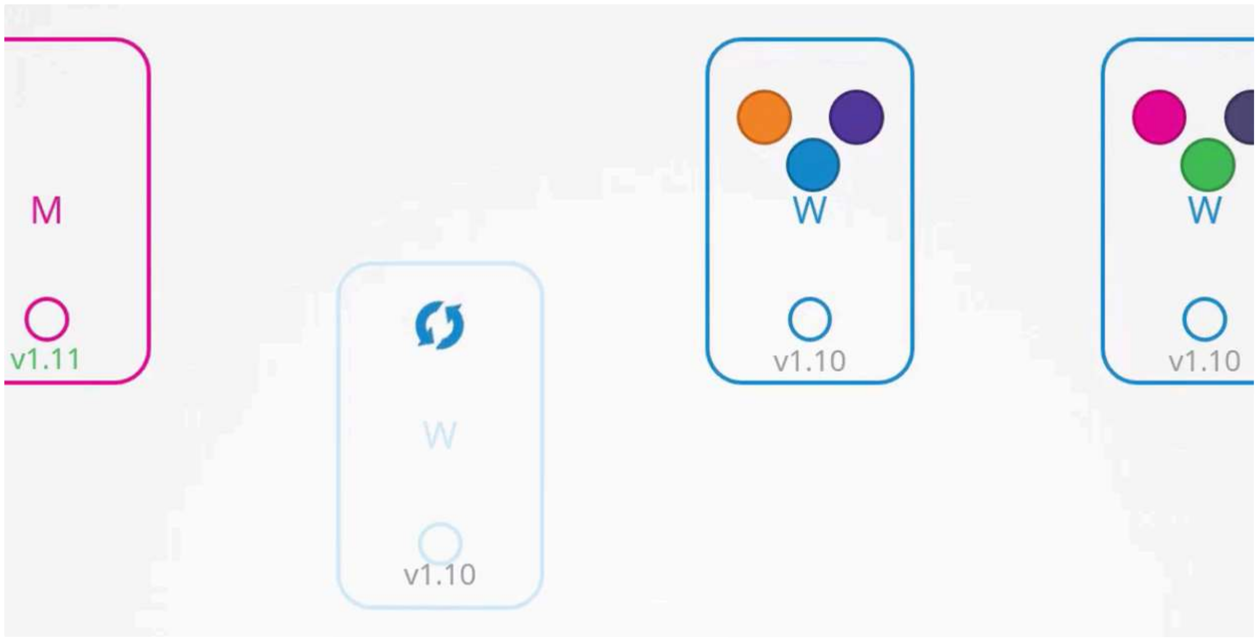
Strategy 1: All at once

Upgrade all the worker nodes at once. This will lead to downtime as no pods will be running during the upgrade.



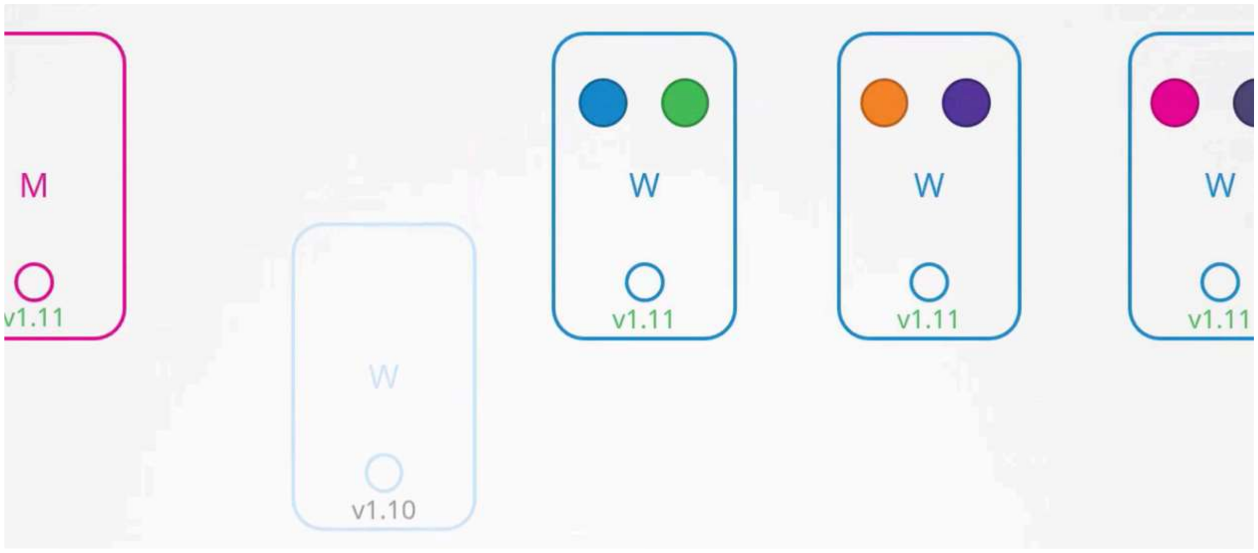
Strategy 2: One at a time

Move the pods of the first node to the remaining nodes and upgrade the first node. Then repeat the same for the rest of the nodes.



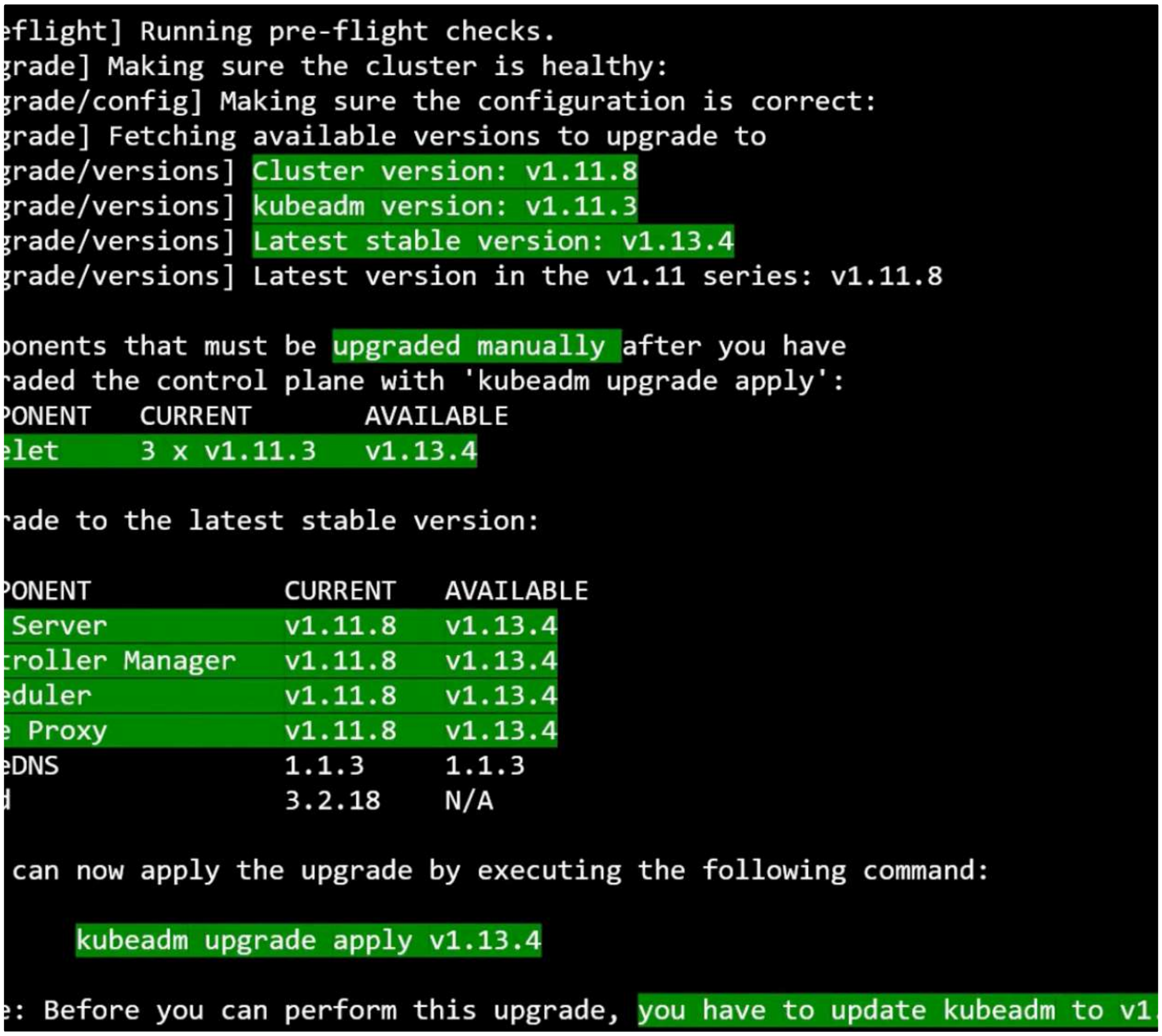
Strategy 3: Add new nodes

Add a new worker node with the latest k8s components running on it, drain the first node and then terminate it. Repeat the same for other nodes.



Upgrading a cluster managed by KubeAdmin

To view the version details of the cluster along with the latest k8s release, run `kubeadm upgrade plan` command on the master node.



```
eflight] Running pre-flight checks.
grade] Making sure the cluster is healthy:
grade/config] Making sure the configuration is correct:
grade] Fetching available versions to upgrade to
grade/versions] Cluster version: v1.11.8
grade/versions] kubeadm version: v1.11.3
grade/versions] Latest stable version: v1.13.4
grade/versions] Latest version in the v1.11 series: v1.11.8

ponents that must be upgraded manually after you have
raded the control plane with 'kubeadm upgrade apply':
PONENT    CURRENT       AVAILABLE
elet      3 x v1.11.3   v1.13.4

rade to the latest stable version:

PONENT              CURRENT    AVAILABLE
 Server             v1.11.8    v1.13.4
troller Manager     v1.11.8    v1.13.4
eduler              v1.11.8    v1.13.4
e Proxy             v1.11.8    v1.13.4
eDNS                1.1.3      1.1.3
d                   3.2.18     N/A

 can now apply the upgrade by executing the following command:

    kubeadm upgrade apply v1.13.4

e: Before you can perform this upgrade, you have to update kubeadm to v1
```

Let's say we want to upgrade the cluster version from `v1.11.0` to `v1.13.0`. Since we can only upgrade one minor version at a time, we'll first upgrade to `v1.12.0`.

Upgrading the master node

Before upgrading the cluster, update the `kubeadm` tool (`kubeadm` follows the same versioning as K8s) - `apt-get upgrade -y kubeadm=1.12.0-00`.

Now, upgrade the cluster (control plane components of the cluster), except the `kubelet` service as it is not controlled by `kubeadm` - `kubeadm upgrade apply v1.12.0`.

If the `kubelet` service is running on the master node, we need to upgrade it. Start by draining the master node by running `k drain controlplane`. If `kubelet` is installed as a service, upgrade its package version - `apt-get upgrade -y kubelet=1.12.0-00` and restart the service - `systemctl restart kubelet`. Now, un-cordon the master node - `k uncordon controlplane`. The master node has been upgraded.

Upgrading the worker nodes

Backup and Restore

The following can be backed up:

- Resource Configurations
- ETCD Cluster
- Persistent Volumes

Backing up Resource Configuration

If all the k8s resources are created using config files (declarative approach), then the configuration directory can be backed up using a version control system like Git. If all the resources are not created this way, we can generate resource configuration by running `kubectl get all --all-namespaces -o yaml > all.yaml`.

Recommended to use **Velero**, a managed tool that can take backups of the cluster configuration.

Backing up ETCD Cluster

ETCD cluster can be backed up instead of generating the resource configuration for the cluster. For this, backup the data directory of the ETCD cluster.

In managed k8s engine, ETCD data directory is not accessible. In such cases, backup the resource configuration.

Velero

- Open-source
- Supports various plugins to backup the cluster to different storage locations like S3, Azure Blob Storage, etc.
- Download the binary and run the `velero install` command along with the storage plugin and credentials to create the Velero pod looking for backups in the storage destination.
- Works in CLI only
- Runs a velero container in the cluster
- We can define a TTL for the backups stored in the storage location.
- Kubernetes Backup & Restore using Velero | ADITYA JOSHI | - YouTube

- Kubernetes Cluster Migration | Migrating Data Across Clusters | ADITYA JOSHI | - YouTube
- Kubernetes Backups, Upgrades, Migrations - with Velero - YouTube

Cluster Design

Basics

- K8s is not supported on Windows. You need to use a virtualization software to run a linux OS on it to support K8s.
- **Minikube** uses a virtualization software to run VMs that host k8s components. This allows us to deploy single node clusters easily.
- **KubeAdm** deploys single or multi node cluster on provisioned VMs for development purposes.

Deploying production-grade clusters

Production grade clusters can be either setup from scratch (turnkey solution) or they can be managed by a cloud provider (hosted solution).

Turnkey Solution

- You provision VMs
- You configure VMs
- You use scripts to deploy cluster
- You maintain VMs yourself
- Eg: OpenShift, Vagrant, etc.

Hosted Solution

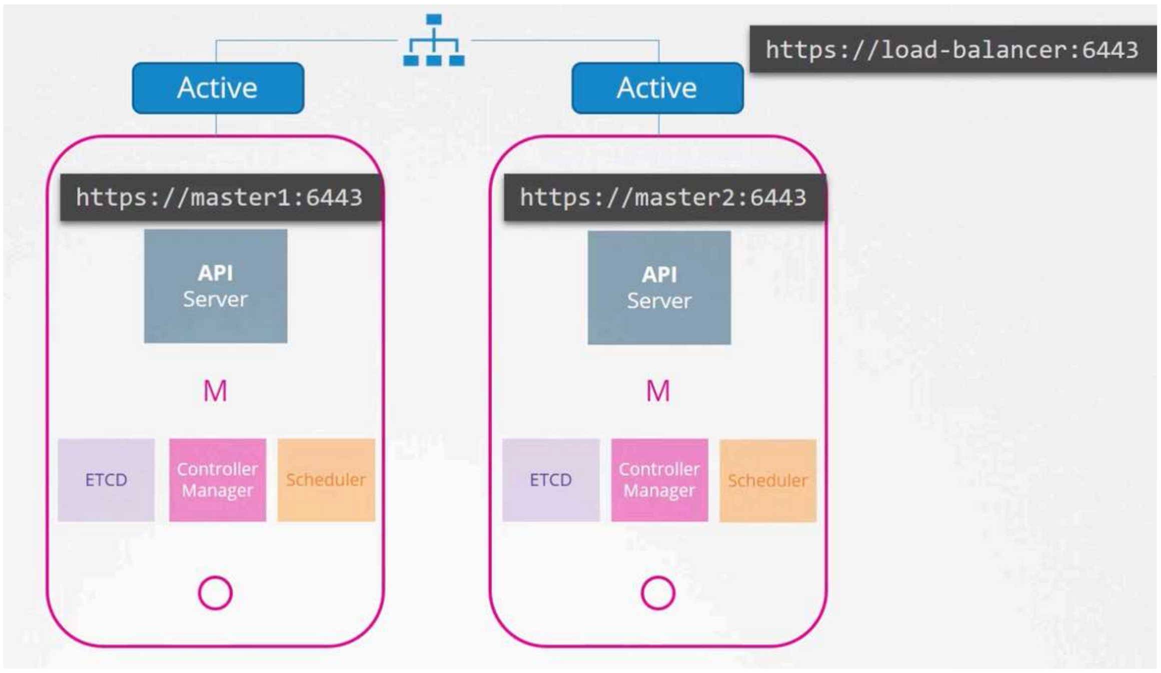
- Kubernetes-as-a-Service
- Provider provisions and configures VMs
- Provider installs Kubernetes
- Provider maintains VMs
- Eg: EKS, GKE, OpenShift Online, AKS

High Availability

To avoid single point of failure, production grade clusters should have at least 3 master nodes. Consider a cluster with 2 master nodes.

API Server

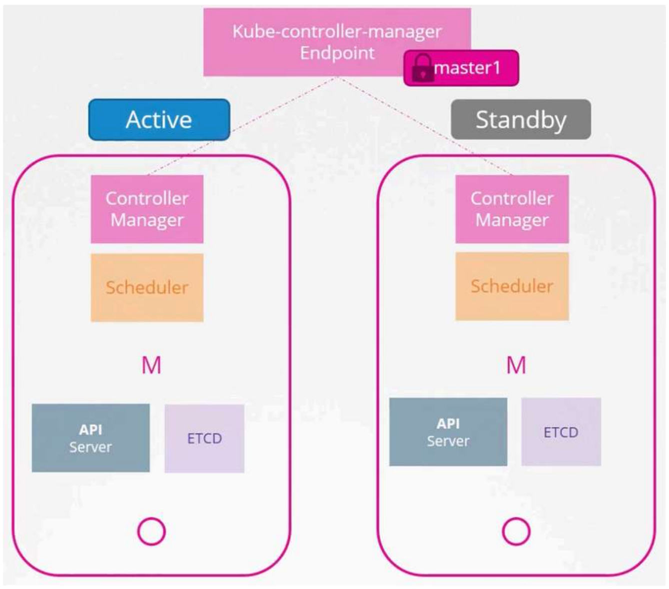
The **API Server** is just a REST API server, it can be kept running on both the masters in an **active-active mode**. To distribute the incoming requests to both the KubeAPI servers, use a load balancer like `nginx` in front and point the `kubectl` utility to the load balancer (by configuring KubeConfig).



Controller Manager and Scheduler

The **Controller Manager** and the **Scheduler** look after the state of the cluster and make necessary changes. Therefore to avoid duplicate processing, they run in an **active-passive mode**.

The instance that will be the active one is decided based on a **leader-election** approach. The two instances compete to lease the endpoint. The instance that leases it first gets to be the master for the lease duration. The active instance needs to renew the lease before the deadline is reached. Also, the passive instance retries to get a lease every `leader-elect-retry-period`. This way if the current active instance crashes, the standby instance can become active.

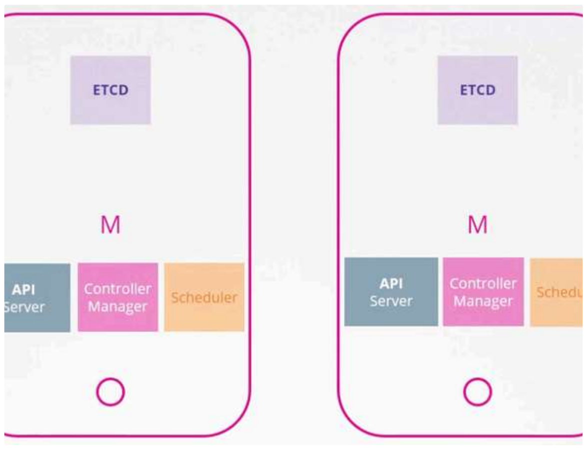


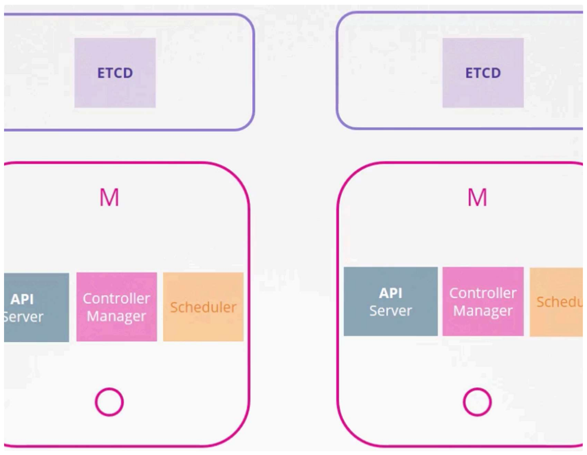
The `kube-controller-manager` and the `kube-scheduler` have leader election turned on by default.

```
kube-controller-manager --leader-elect true [other options] --leader-elect-le
ase-duration 15s --leader-elect-renew-deadline 10s --leader-elect-retry-perio
d 2s kube-scheduler --leader-elect true [other options] --leader-elect-lease-
duration 15s --leader-elect-renew-deadline 10s --leader-elect-retry-period 2s
```

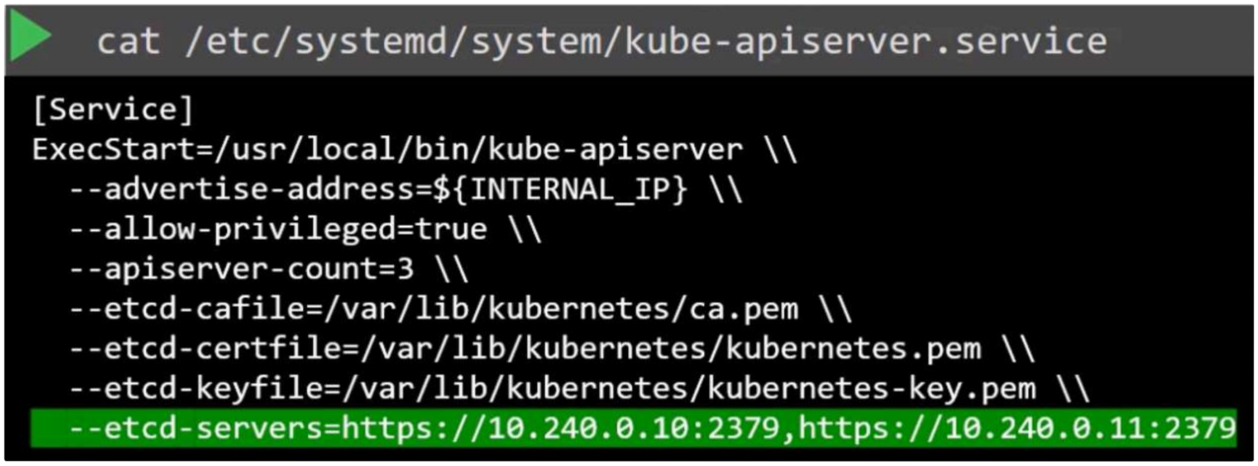
ETCD

The **ETCD Server** can be present in the master node (**stacked topology**) or externally on other servers (**external ETCD topology**). Stacked topology is easier to setup and manage but if both the master nodes are down, then all the control plane components along with the state of the cluster (ETCD servers) will be lost and thus redundancy will be compromised.





Regardless of the topology, both the API server instances should be able to reach both the ETCD servers as it is a distributed database.

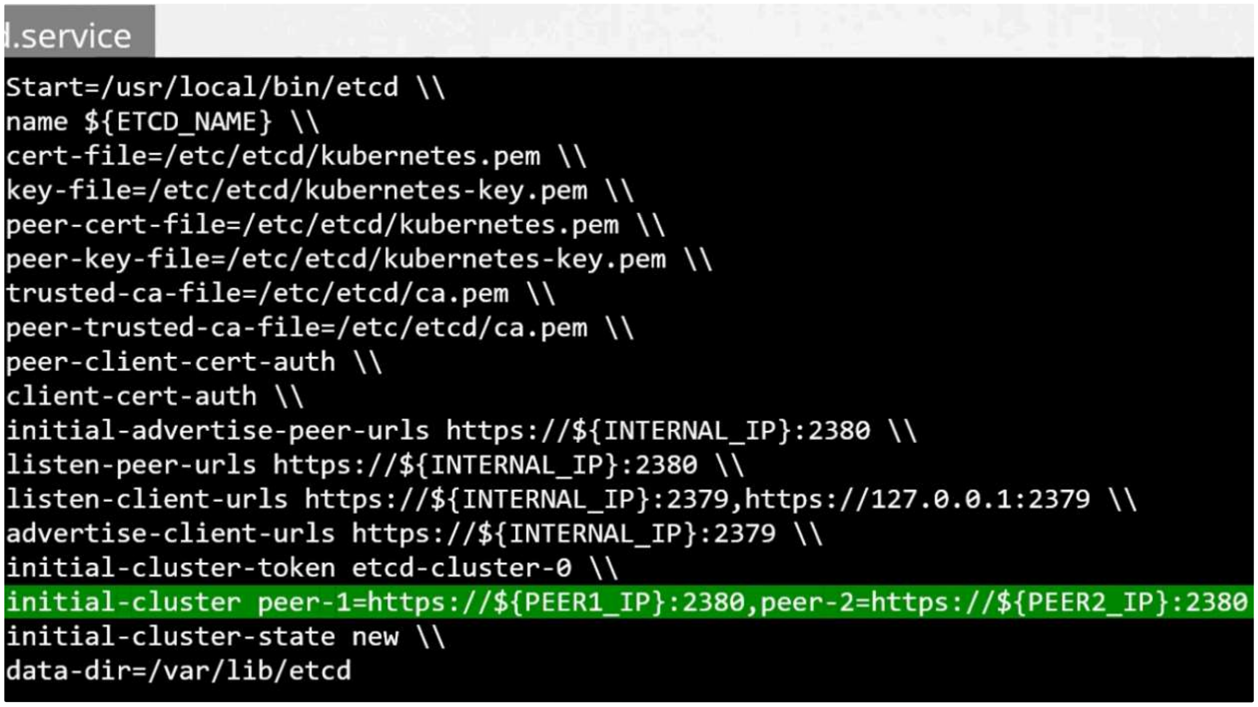


```
cat /etc/systemd/system/kube-apiserver.service

[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --etcd-servers=https://10.240.0.10:2379,https://10.240.0.11:2379
```

ETCD is distributed datastore, which means it replicates the data across all its instances. It ensures **strongly consistent writes by electing a leader ETCD instance (master) which processes the writes**. If the writes go to any of the slave instances, they are forwarded to the master instance. Whenever writes are processed, the master ensures that the data is updated on all of the slaves. **Reads can take place on any instance.**

When setting up the ETCD cluster for HA, use the `initial-cluster` option to configure the ETCD peers in the cluster.



```
.service
Start=/usr/local/bin/etcd \\
name ${ETCD_NAME} \\
cert-file=/etc/etcd/kubernetes.pem \\
key-file=/etc/etcd/kubernetes-key.pem \\
peer-cert-file=/etc/etcd/kubernetes.pem \\
peer-key-file=/etc/etcd/kubernetes-key.pem \\
trusted-ca-file=/etc/etcd/ca.pem \\
peer-trusted-ca-file=/etc/etcd/ca.pem \\
peer-client-cert-auth \\
client-cert-auth \\
initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
listen-peer-urls https://${INTERNAL_IP}:2380 \\
listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \\
advertise-client-urls https://${INTERNAL_IP}:2379 \\
initial-cluster-token etcd-cluster-0 \\
initial-cluster peer-1=https://${PEER1_IP}:2380,peer-2=https://${PEER2_IP}:2380
initial-cluster-state new \\
data-dir=/var/lib/etcd
```

ETCD instances elect the leader among them using **RAFT protocol**. If the leader doesn't regularly notify the other instances of its role regularly, it is assumed to have died and a new leader is elected.

A write operation is considered complete only if it can be written to a majority (**quorum**) of the ETCD instances (to ensure that a write goes through if a minority of ETCD instances goes down). If a dead instance comes back up, the write is propagated to it as well.

`Quorum = floor(N/2 + 1)`, where N is the total number of ETCD instances, is the minimum number of ETCD instances that should be up and running for the cluster to perform a successful write.

`N = 2 => Quorum = 2`, which means quorum cannot be met even if a single instance goes down.

Managers	Majority	Fault Tolerance
1	1	0
2	2	0
3	2	1