Cluster maintenance

Operating system upgrade

If a node goes down, the pods on the node become unavailable, this causes the applications on those pods to become unavailable to the users.

If there are no other pods on any live nodes running that applications, users of that application are impacted.

If the node comes back-up quickly, and the kubelet on the node goes to running state, then the down-time of the application may not be so significant to impact any user.

If the pods were part of a replicaset then the new pods will be created and assigned to other live nodes. The time kubernetes waits before creating pods for a replica is called "pods eviction timeout" that is set as argument for Kubernetes controller manager.

"Pods eviction timeout" is the time a master node waits before considering a node dead.

If a pod is not part of a replicaset, the pod will be deleted and not recovered until the pod is created again.

So during upgrades of nodes, we need to be mindful and make sure that the pods that were running on a node will be able to function after the upgrade is complete for this, theoretically, we must ensure that the upgrade process is complete within 'pods eviction timeout".

But there is no way to ensure that.

Thus the safer way is to drain the nodes, so that the pods running on the node are safely transferred/migrated to another node..

The way how drain works is by gracefully terminating the pod on current node and starting the same pod on a different node.

Following are some details on "kubectl drain node" command:

It will mark the node as unschedulable.

drain evicts or deletes all the pods, except the mirror pods.

daemonset (DS) managed pods will only be processed if "ignore-daemons set" flag is supplied

DS managed pods are not deleted, because DS controller ignores unschedulable marking and will recreate the pods if deleted

the node should only be worked on after the drain operation is complete; this is because drain waits for graceful termination of the pods and could take some time to complete.

after the working on the node maintenance is completed, the node should be uncordoned.

if there is a standalone pod, the drain command will fail. Drain command will not evict pod that is not managed by replicaset, replication Controller, daemanset, job or statefulset, thus deleting or evicting a standalone pod fails.

the standalone pod can be deleted using "-force" flag; but the pod and its data will be lost forever

the best way to handle this is by deploying the standalone pod as a deployment before deleting the pod.

Cluster upgrade process:

If kube API server is at version X, the controller manager and kube scheduler can be at version (X-1), Or higher.

Similarly, kubelet and kube proxy can be at version (x-2), Or higher.

But none of the components should be higher than Kube API server.

Kubectl however can be at (X+1) or (X-1)

The recommended upgrade approach is to upgrade 1 minor version at a time, incrementally.

Upgrade example from verson X to (X+1).

The usual approach is to upgrade master nodes first and then upgrade worker nodes.

When the master node is being upgraded, the components such as API server, scheduler, controller manager are down briefly. The workers will keep on serving, but since the master nodes and their components are down, the management functions are also down.

You cannot access cluster components using kubectl. No new pods or deployments can be created either.

Once the upgrade on master node is completed, the nodes will come back up, and all the management task will start to work.

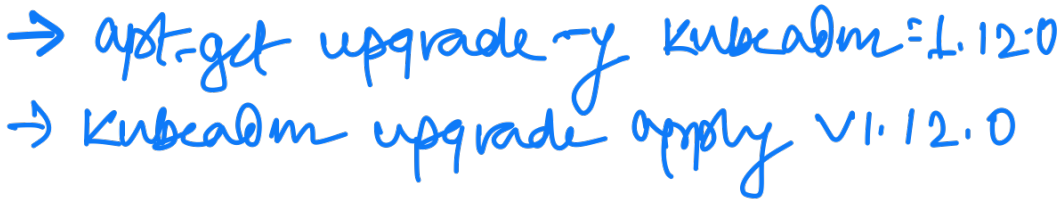
At this point the master modes have version (x +1) whereas the worker nodes are at version (x)

To upgrade the worker node, if we upgrade all worker nodes at the same time, there will be some down time. So, to prevent the down time, we can drain and upgrade worker nodes one at a time; or in a batch.

“kubeadm upgrade plan” lists details and important info on current version and new version that are available. During upgrade, you can follow the steps listed in the upgrade plan.

Upgrade using kube adm tool:

First, upgrade kubeadm in the master node/s. Then upgrade control plane components from the master node/s. Following commands to run on each master node :



This *kubeadm upgrade apply* will upgrade all control plane components. However, the node will still show the version as old version. This is because the kubelet on the nodes are not upgraded yet.

To upgrade kubelet on the master node, do the following after draining the master node.

```
→ apt-get upgrade -y kubelet=1.12.0-00
```

Restart kubelet after the upgrade operation. At this point, the kubectl get nodes command should show newer version for the master node.

Now, for the worker node, do the following:

```
kubectl drain <my-worker>
apt-get upgrade -y kubeadm=1.12.0-00
apt-get upgrade -y kubelet=1.12.0-00
kubeadm upgrade node config
        --kubelet-version v1.12.0

systemctl restart kubelet
kubectl uncordon <my-worker>
```

And do the same thing to other worker nodes.

kubeadm documentations also has steps laid out to upgrade for each version.

Backups and restore:

Lets look at the backup and restore. Resource configs, ETCD database and persistent storage are some of the candidates that we need to backup, just in case we need to restore them in case of disaster recovery.

Resources can be created using imperative approach by the command line; but to save the object/resource definition for future, it is better to save item in declarative YAML file; so that they can be reused in future.

But saving an the items definitions in declarative way may not be adhered by all team members; although that is the best way to preserve the definition. So, just to be sure that we have definition of all the resources, API server can be queried as follows:

kubectl get all --all-namespaces -o yaml > all-items.yaml

But this will only get definitions of some resources, not all. There are some services that will get the definition for us, such as some external products.

For ETCD database, it stores the state of the cluster.. So, we may choose to backup ETCD server itself rather than going through the use of other tools.

ETCD servers are usually hosted on master nodes. There is a location specified for the data directory. We can configure our backup system to backup that location.

We can also create a snapshot of ETCD using ***etcdctl snapshot save*** command.

The database of ETCD can be restored using following steps:

→ Stop the apiserver using
service kube-apiservice stopped ↵
→ Restore using following
ETCDCTL_API=3 etcdctl snapshot restore \
snap.db --data-dir /var/lib/etcd ↵
→ And update etcd.service to use the new
data-dir
→ Then reload the service daemon using
systemctl daemon-reload ↵
→ And, restart etcd service using
service etcd restart
→ Finally, start kube apiserver service

In a practical case, to create a snapshot, we need to supply some other arguments also.

--endpoints ← can be found in etcd yaml manifest
--cacert ← grab "trusted-ca-cert" argument from manifest
--cert ← grab "cert-file" from manifest
--key ← grab "key-file" from manifest file

Restoring now from the above created backup is a local process which doesnot require the certificates and key files. But we do need to update the ETCD manifest file to point to the new data directory.

To point to the new data directory in the manifest YAML file, we may need to update the "host path" field is the volumes field. Also make sure that the data dir in the ETCD argument is the same as the mount path in volume mount.

Any update is manifest are applied automatically. As the etcd changes are applied and the pods are coming up, it also restarts scheduler and kube controller manager.

Useful info:

Stacked etcd: ETCD set up on the master node of the cluster.
External etcd: ETCD running on external place somewhere.
Verify which one: check the api-server config for etcd config and endpoint
Check ETCD config: when not running kubectl to check the apiserver config and for external ETCD setting the arguments can be checked using following command on the node:

```
ps -ef | grep -i etcd
```

All the arguments needed to run the etcdctl are found on the output produced by the ps command above like endpoint, cert, cacert, and key.

When running as external cluster, the eyed is run as a systemd process, so to modify the arguments we need to use following:

```
vi /etc/systemd/system/etcd.service
```

This above file contains the description of the ETCD to run in an external cluster. After updating the file above, we need to reload systemd daemon. In this case, we will update the —data-dir argument in the system file

(unlike in manifest case where we updated the host file path in volumes of the yam file).