Open in app Get started

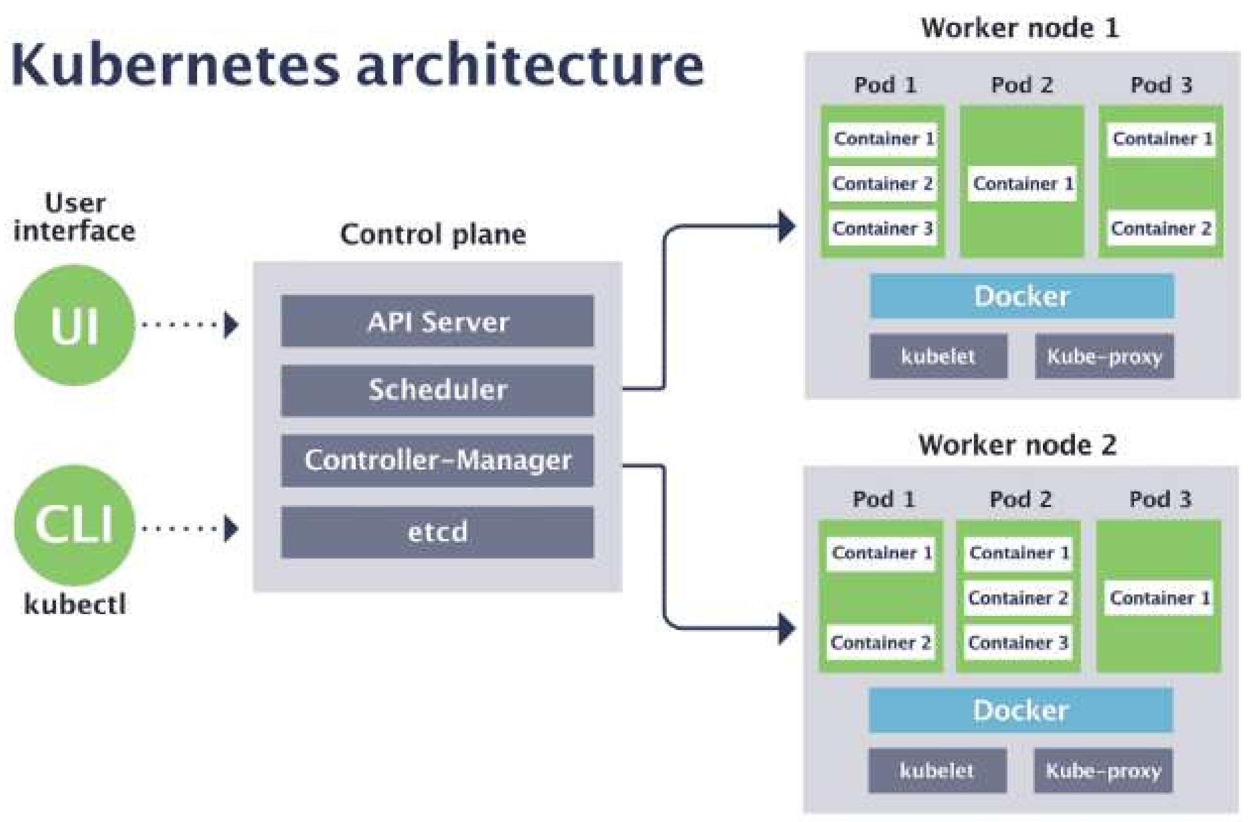
Published in techbeatly

Nived Velayudhan Follow

Jan 14 · 7 min read · Listen

Save

Kubernetes Architecture — Deep Dive



CONTROL PLANE COMPONENTS:

metadata. The only component that talks to etcd directly is the Kubernetes API server. All other components read and write data to etcd indirectly through the API server.

Etcd also implements a watch feature, which provides an event-based interface for asynchronously monitoring changes to keys. Once a key is changed, its watchers get notified. The API Server component heavily relies on this to get notified and to move the current state of etcd towards the desired state.

What should the number of etcd instances be an odd number?

In an HA environment, you would typically have 3,5, or 7 instances of etcd instances running, but why? Since etcd is a distributed data store, it is possible to scale it horizontally but also you need to ensure that the data in each instance is consistent and for this, your system needs to reach a consensus on what the state is. Etcd uses RAFT consensus algorithm for this.

The algorithm requires a majority (or quorum) for the cluster to progress to the next state. If you have only 2 instances of ectd, if either of them fails, the etcd cluster can't transition to a new state because no majority exists and in the case of 3 instances, one instance can fail and quorum can be reached as a majority of instances are still available.

API Server:

The API server is the only component in Kubernetes that directly interacts with etcd. All other components in Kubernetes and also by the clients (kubectl) must go through the API server to work with the cluster state. API server provides the following functionalities:

- Provides a consistent way of storing objects in etcd.
- Performs validation of those objects so clients can't store improperly configured objects which could have happened if they were writing to the etcd datastore directly.
- Provides a RESTful API to create, update, modify, or delete a resource.
- Provides optimistic concurrency locking so changes to an object are never overridden

to and determine whether the authenticated user can perform the requested action on the requested resource.

- Responsible for admission control if the request is trying to create, modify or delete a resource. Example: AlwaysPullImages, DefaultStorageClass, ResourceQuota, etc.
- Implements a watch mechanism (similar to etcd) for clients to watch for changes. This allows components such as the Scheduler and Controller Manager to interact with the API Server in a loosely coupled manner.

Controller Manager:

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state. Controller tracks at least one Kubernetes resource type and these objects have a spec field that represents the desired state.

Examples of controllers:

- Replication Manager (a controller for ReplicationController resources)
- ReplicaSet, DaemonSet, and Job controllers
- Deployment controller
- StatefulSet controller
- Node controller
- Service controller
- Endpoints controller
- Namespace controller
- PersistentVolume controller

themselves, but because using watches doesn't guarantee the controller won't miss an event, they also perform a re-list operation periodically to make sure they haven't missed anything.

The Controller Manager also performs lifecycle functions such as namespace creation and lifecycle, event garbage collection, terminated-pod garbage collection, cascading-deletion garbage collection, node garbage collection, etc

Also, read — Cloud Controller Manager

Scheduler:

The scheduler is a control plane process that assigns pods to nodes. It watches for newly created pods that have no nodes assigned and for every pod that the scheduler discovers, the scheduler becomes responsible for finding the best node for that pod to run on.

Nodes that meet the scheduling requirements for a Pod are called feasible nodes. If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it. Once it finds a feasible node, it runs a set of functions to score the nodes and the node with the highest score is selected. It then notifies the API server about the selected node and this process is called binding.

The selection of nodes is a 2-step process:

1. **Filtering** the list of all nodes to obtain a list of acceptable nodes the pod can be scheduled to. (eg. the PodFitsResources filter checks whether a candidate Node has enough available resource to meet a Pod's specific resource requests)

2. **Scoring** the list of nodes obtained from step-1 & ranking them to choose the best node. If multiple nodes have the highest score, a round-robin is used to ensure pods are deployed across all of them evenly.

Factors that need to be taken into account for scheduling decisions can include:

- If the pod requests to be bound to a specific host port, is that port already taken on this node or not?
- Does the pod tolerate the taints of the node?
- Does the pod specify node affinity or anti-affinity rules? etc.

The Scheduler doesn't instruct the selected node to run the pod. All the Scheduler does is update the pod definition through the API server. The API server through the watch mechanism then notifies the Kubelet that the pod has been scheduled. Then the kubelet service on the target node sees the pod has been scheduled to its node, it creates and runs the pod's containers.

WORKER NODE COMPONENTS:

Kubelet:

Kubelet is an agent that runs on each node in the cluster and is the component responsible for everything running on a worker node. It makes sure that containers are running in a Pod.

The main functions of kubelet service are:

1. Register the node it's running on by creating a node resource in the API server.
2. Continuously monitor the API server for Pods that have been scheduled to the node.
3. Start the pod's containers by using the configured container runtime.
4. Continuously monitors running containers and reports their status, events, and resource consumption to the API server.
5. Runs the container liveness probes, restart containers when the probes fail, terminate containers when their Pod is deleted from the API server and notifies the server that the pod has terminated.

for watching the API Server for changes on services and pods definitions to maintain the entire network configuration up to date. When a service is backed by more than one pod, the proxy performs load balancing across those pods.

The kube-proxy got its name because it was an actual proxy server that used to accept connections and proxy them to the pods, but the current implementation uses iptables rules to redirect packets to a randomly selected backend pod without passing them through an actual proxy server.

High-Level view of it works:

1. When a service is created, a virtual IP address is assigned immediately.
2. API server notifies the kube-proxy agents running on worker nodes that a new service has been created.
3. Each kube-proxy makes the service addressable by setting up iptables rules, ensuring each service IP/Port pair is intercepted and the destination address modified to one of the pods that back the service.
4. Watches the API server for changes to services or its endpoint objects.

Container runtime:

Container runtimes that focus on running containers, setting up namespace, and cgroups for containers are called lower-level container runtimes and the ones that focus on formats, unpacking, management, and sharing of images and provide APIs for developers needs are called higher-level container runtimes (container engine).

Container runtime takes care of:

1. Pulling the container image that is required by the container from an image registry, if its not available locally.
2. Extracting the image onto a copy-on-write filesystem and all the container layers

4. Setting the metadata from container image like overriding CMD, ENTRYPOINT from user inputs, setting up SECCOMP rules, etc to ensure container runs as expected.
5. Altering the kernel to assign some sort of isolation like process, networking & filesystem to this container.
6. Alerting the kernel to assign some resource limits like CPU or memory limits.
7. Pass system call (syscall) to the kernel to start the container.
8. Making sure SElinux/AppArmor is set up properly.

Image Credits: CNCF