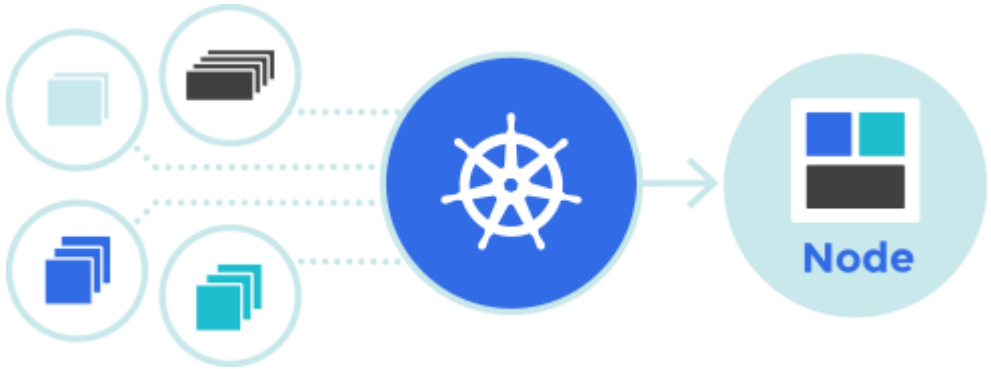
Kubernetes

What is Kubernetes



- Kubernetes is an open-source system for automating deployment, operations, and scaling of containerized applications.
- The Kubernetes project was started by Google in 2014
- It groups containers that make up an application into logical units for easy management and discovery
- Kubernetes builds upon a decade and a half of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

Features of Kubernetes

- Planet Scale
  - Designed on the same principles that allows Google to run billions of containers a week, Kubernetes can scale without increasing your ops team.

- Never Outgrow
  - Whether testing locally or running a global enterprise, Kubernetes flexibility grows with you to deliver your applications consistently and easily no matter how complex your need is.
- Run Anywhere
  - Kubernetes is open source giving you the freedom to take advantage of on-premise, hybrid, or public cloud infrastructure, letting you effortlessly move workloads to where it matters to you.

Kubernetes is:

- **portable**: public, private, hybrid, multi-cloud
- **extensible**: modular, pluggable, hookable, composable
- **self-healing**: auto-placement, auto-restart, auto-replication, auto-scaling

Kubernetes container scheduler

- Kubernetes can schedule and run application containers on clusters of physical or virtual machines.
- In order to take full advantage of the potential benefits of containers and leave the old deployment methods behind, one needs to cut the cord to physical and virtual machines.
- However, once specific containers are no longer bound to specific machines, **host-centric** infrastructure no longer works: managed groups, load balancing, auto-scaling, etc. One needs **container-centric** infrastructure. That’s what Kubernetes provides

Why Kubernetes

- Most clustering technologies strive to provide a uniform platform for application deployment. The user should not have to care much about where work is scheduled.
- The unit of work presented to the user is at the "service" level and can be accomplished by any of the member nodes.
- However, in many cases, it *does* matter what the underlying infrastructure looks like.
- When scaling an app out, an administrator cares that the various instances of a service are not all being assigned to the same host.

Why Kubernetes

- On the other side of things, many distributed applications build with scaling in mind are actually made up of smaller component services.
- These services must be scheduled on the same host as related components if they are going to be configured in a trivial way.
- This becomes even more important when they rely on specific networking conditions in order to communicate appropriately.

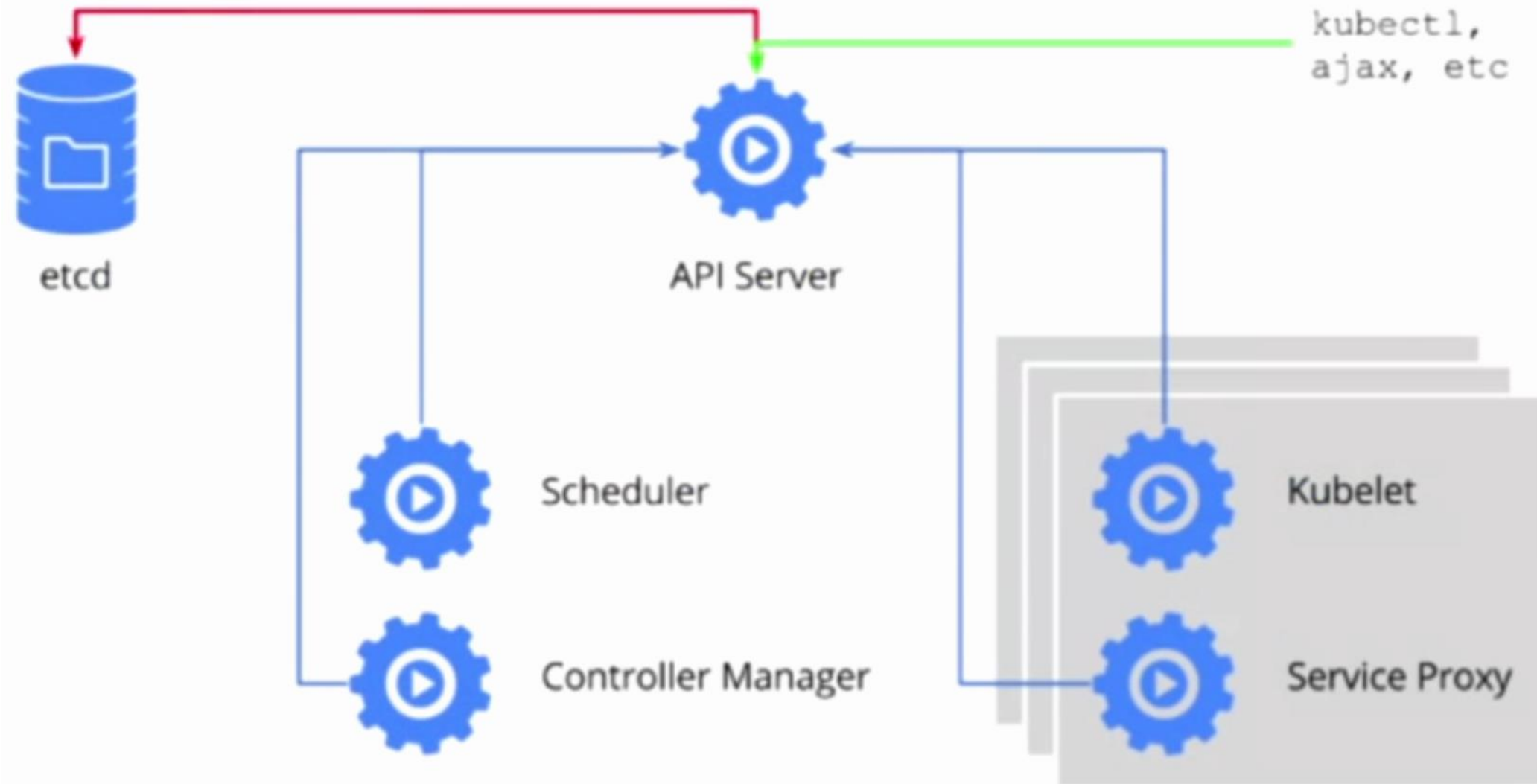
Why Kubernetes

- While it is possible with most clustering software to make these types of scheduling decisions, operating at the level of individual services is not ideal.
- Applications comprised of different services should still be managed as a single application in most cases.
- Kubernetes provides a layer over the infrastructure to allow for this type of management.

Master Server Components

- The controlling unit in a Kubernetes cluster is called the **master** server.
- It serves as the main management contact point for administrators, and it also provides many cluster-wide systems for the relatively dumb worker nodes.
- The master server runs a number of unique services that are used to manage the cluster's workload and direct communications across the system.

Kubernetes Architecture



Etcd

- One of the fundamental components that Kubernetes needs to function is a globally available configuration store.
- The `etcd` project, developed by the CoreOS team, is a lightweight, distributed key-value store that can be distributed across multiple nodes.
- Kubernetes uses `etcd` to store configuration data that can be used by each of the nodes in the cluster.
- This can be used for service discovery and represents the state of the cluster that each component can reference to configure or reconfigure themselves.
- By providing a simple HTTP/JSON API, the interface for setting or retrieving values is very straight forward.
- The implementation of `etcd` on a Kubernetes cluster is a bit more flexible than CoreOS.

API Server

- One of the most important services that the master server runs is an API server.
- This is the main management point of the entire cluster, as it allows a user to configure many of Kubernetes' workloads and organizational units.
- It also is responsible for making sure that the `etcd` store and the service details of deployed containers are in agreement.
- The API server implements a RESTful interface, which means that many different tools and libraries can readily communicate with it.
- A client called `kubecfg` is packaged along with the server-side tools and can be used from a local computer or by connecting to the master server.

Controller Manager Server

- The controller manager service is used to handle the replication processes defined by replication tasks.
- The details of these operations are written to `etcd`, where the controller manager watches for changes.
- When a change is seen, the controller manager reads the new information and implements the replication procedure that fulfills the desired state.
- This can involve scaling the application group up or down.

Scheduler Server

- The process that actually assigns workloads to specific nodes in the cluster is the scheduler.
- This is used to read in a service's operating requirements, analyze the current infrastructure environment, and place the work on an acceptable node or nodes.
- The scheduler is responsible for tracking resource utilization on each host to make sure that workloads are not scheduled in excess of the available resources.
- The scheduler must know the total resources available on each server, as well as the resources allocated to existing workloads assigned on each server.

Minion Server Components

- In Kubernetes, servers that perform work are known as **minions**.
- Minion servers have a few requirements that are necessary to communicate with the master, configure the networking for containers, and run the actual workloads assigned to them.

Kubelet Service

- The main contact point for each minion with the cluster group is through a small service called `kubelet`.
- This service is responsible for relaying information to and from the master server, as well as interacting with the `etcd` store to read configuration details or write new values.
- The `kubelet` service communicates with the master server to receive commands and work.
- Work is received in the form of a "manifest" which defines the workload and the operating parameters.
- The`kubelet` process then assumes responsibility for maintaining the state of the work on the minion server.

Proxy Service

- In order to deal with individual host subnetting and in order to make services available to external parties, a small proxy service is run on each minion server.
- This process forwards requests to the correct containers, can do primitive load balancing, and is generally responsible for making sure the networking environment is predictable and accessible, but isolated.

Kubernetes Work Units

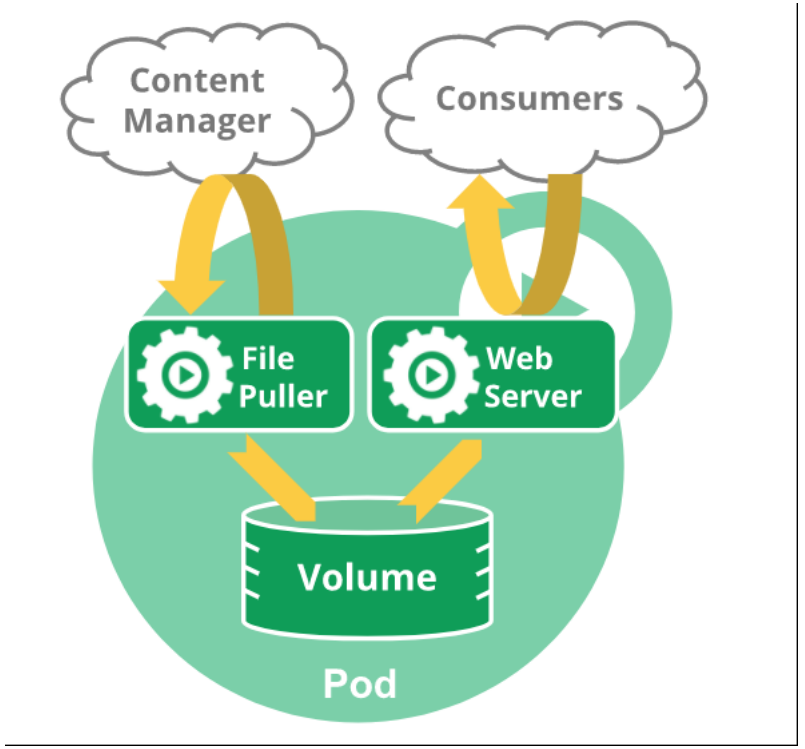
- While containers are the used to deploy applications, the workloads that define each type of work are specific to Kubernetes. We will go over the different types of "work" that can be assigned below.
  - **Pods**
  - **Services**
  - **Replication Controllers**
  - Labels

Pods

- A *Pod* is the basic building block of Kubernetes
- the smallest and simplest unit in the Kubernetes object model that you create or deploy.
- A Pod represents a running process on your cluster.
- A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run.

Pods

- Pod represents a unit of deployment: *a single instance of an application in Kubernetes*, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.



Pods in a Kubernetes cluster can be used in two main ways:

- **Pods that run a single container**. The “one-container-per-Pod” model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.
- **Pods that run multiple containers that need to work together**. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers might form a single cohesive unit of service–one container serving files from a shared volume to the public, while a separate “sidecar” container refreshes or updates those files. The Pod wraps these containers and storage resources together as a single manageable entity.

POD Networking

- Each Pod is assigned a unique IP address.
- Every container in a Pod shares the network namespace, including the IP address and network ports.
- Containers *inside a Pod* can communicate with one another using localhost.
- When containers in a Pod communicate with entities *outside the Pod*, they must coordinate how they use the shared network resources (such as ports).

POD Storage

- A Pod can specify a set of shared storage *volumes*.
- All containers in the Pod can access the shared volumes, allowing those containers to share data.
- Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted.

POD Lifecycle

- The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle:
- Pending: The Pod has been accepted by the Kubernetes system, but one or more of the Container images has not been created. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while.
- Running: The Pod has been bound to a node, and all of the Containers have been created. At least one Container is still running, or is in the process of starting or restarting.

POD Lifecycle

- Succeeded: All Containers in the Pod have terminated in success, and will not be restarted.
- Failed: All Containers in the Pod have terminated, and at least one Container has terminated in failure. That is, the Container either exited with non-zero status or was terminated by the system.
- Unknown: For some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Pod.

POD Restart policy

- A PodSpec has a restartPolicy field with possible values
  - Always,
  - OnFailure, and
  - Never.
- The default value is Always.
- restartPolicy applies to all Containers in the Pod.
- restartPolicy only refers to restarts of the Containers by the kubelet on the same node.
- Failed Containers that are restarted by the kubelet are restarted with an exponential back-off delay (10s, 20s, 40s …) capped at five minutes, and is reset after ten minutes of successful execution.
- Pods once bound to a node, a Pod will never be rebound to another node

POD Controllers

Replication Sets

- A ReplicaSet ensures that a specified number of pod replicas are running at any given time.
- A Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to pods along with a lot of other useful features.
- Therefore, we recommend using Deployments instead of directly using ReplicaSets,

POD Deployments

- A *Deployment* controller provides declarative updates for Pods and ReplicationSets
- You describe a *desired state* in a Deployment object,
- the Deployment controller changes the actual state to the desired state at a controlled rate.
- You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Kubernetes Service

- A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service.
- The set of Pods targeted by a Service is (usually) determined by a Label Selector.

Services

- A **service**, when described this way, is a unit that acts as a basic load balancer and ambassador for other containers.
- This allows you to deploy a service unit that is aware of all of the backend containers to pass traffic to.
- External applications only need to worry about a single access point, but benefit from a scalable backend or at least a backend that can be swapped out when necessary.
- Services are an interface to a group of containers so that consumers do not have to worry about anything beyond a single access location.
- By deploying a service, you easily gain discover-ability and can simplify your container designs.

Replication Controllers

- A more complex version of a pod is a **replicated pod**. These are handled by a type of work unit known as a**replication controller**.
- A replication controller is a framework for defining pods that are meant to be horizontally scaled.
- The work unit is, in essence, a nested unit.
- A template is provided, which is basically a complete pod definition.
- This is wrapped with additional details about the replication work that should be done.
- The replication controller is delegated responsibility over maintaining a desired number of copies.
- This means that if a container temporarily goes down, the replication controller might start up another container. If the first container comes back online, the controller will kill off one of the containers.

Labels

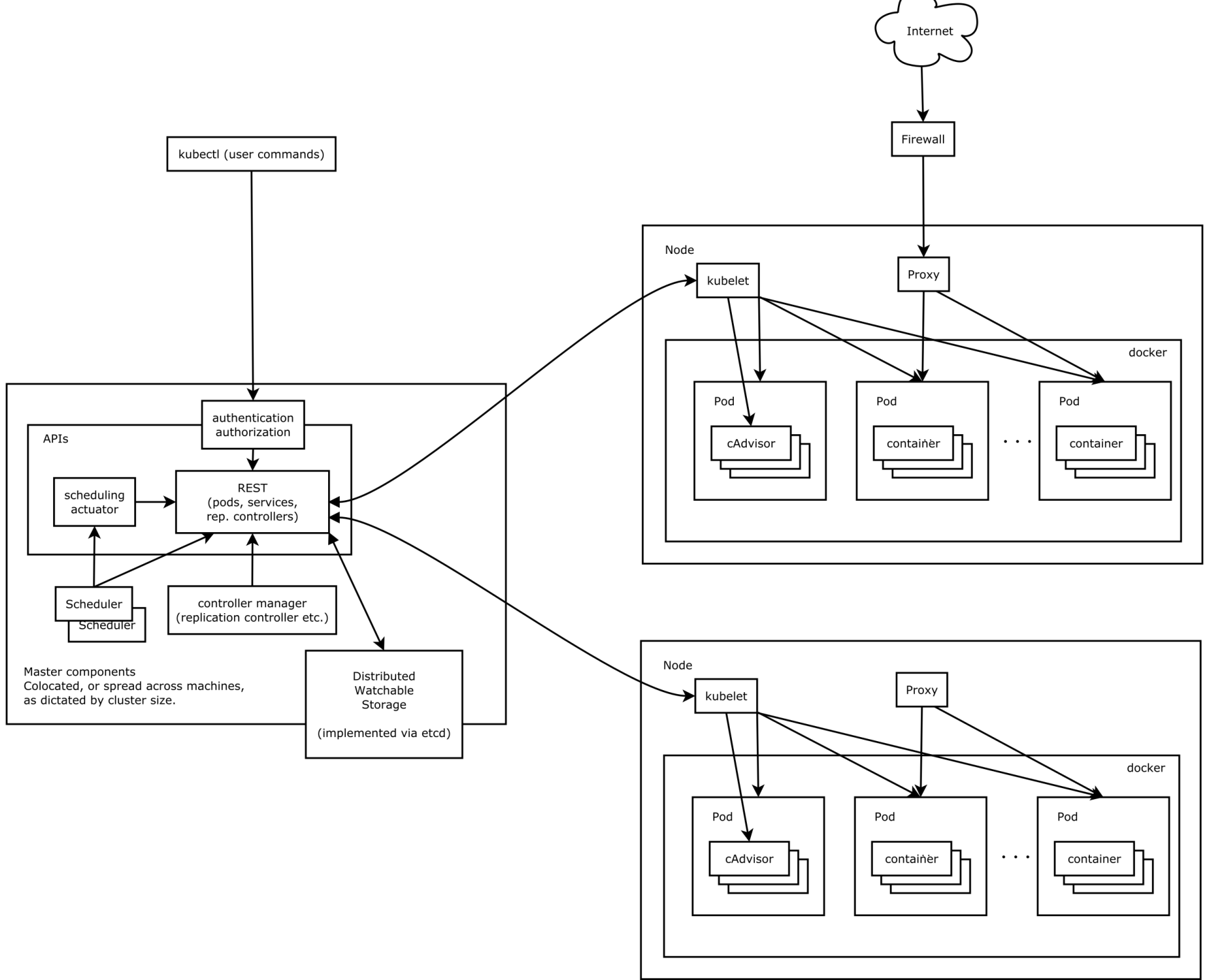
- A Kubernetes organizational concept outside of the work-based units is labeling.
- A **label** is basically an arbitrary tag that can be placed on the above work units to mark them as a part of a group.
- These can then be selected for management purposes and action targeting.
- Labels are fundamental to how both services and replication controllers function. To get a list of backend servers that a service should pass traffic to, it selects containers based on labels that have been given to them.

Labels

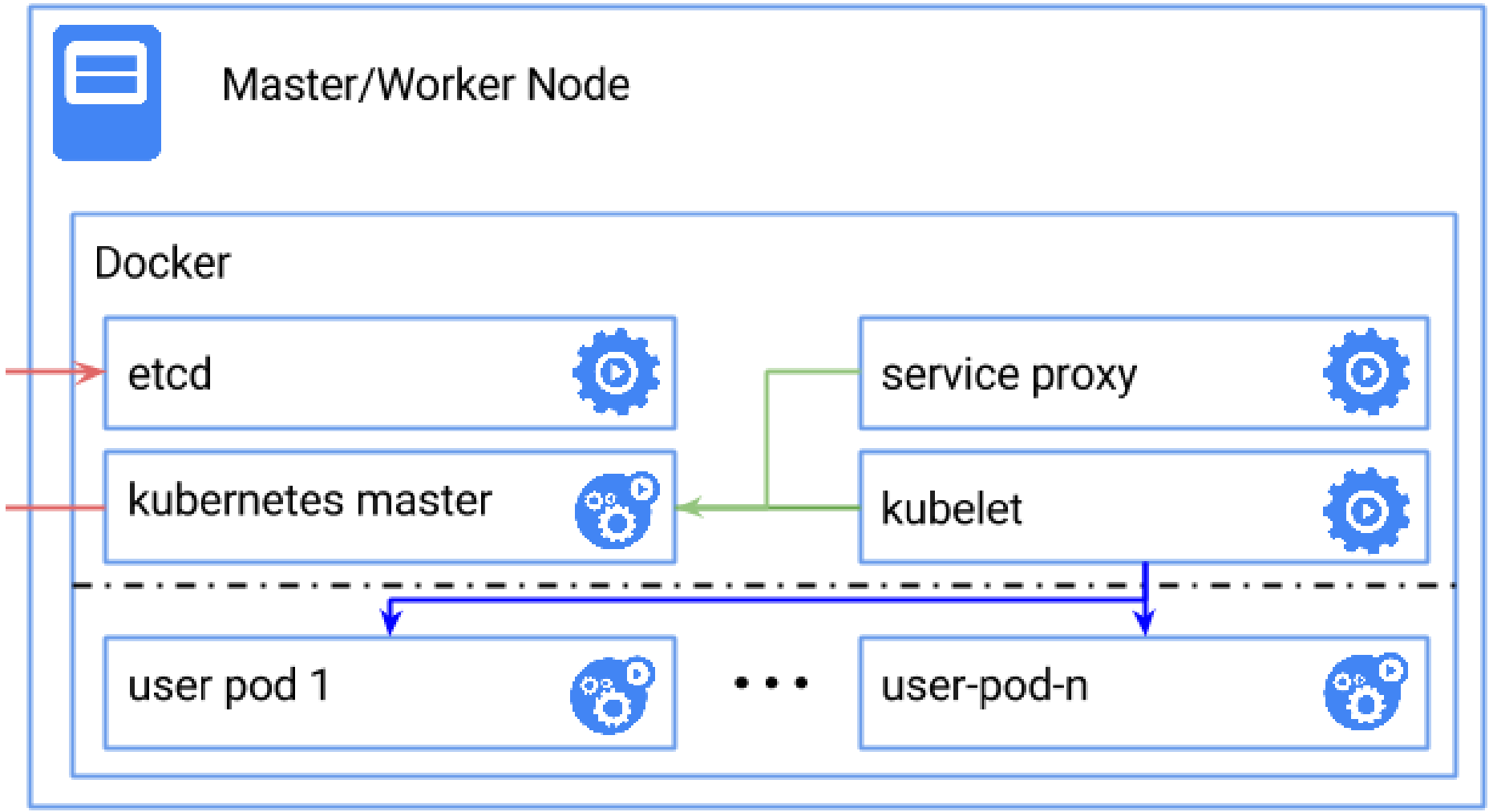
- Similarly, replication controllers give all of the containers spawned from their templates the same label.
- This makes it easy for the controller to monitor each instance.
- The controller or the administrator can manage all of the instances as a group, regardless of how many containers have been spawned.
- Labels are given as key-value pairs.
- Each unit can have more than one label, but each unit can only have one entry for each key. You can stick with giving pods a "name" key as a general purpose identifier, or you can classify them by various criteria such as development stage, public accessibility, etc.
- In many cases you'll want to assign many labels for fine-grained control. You can then select based on a single or combined label requirements.

Kubectl CLI

- The easiest way to interact with Kubernetes is via the kubectl command-line interface.
- Installing Kubernetes Command line


Internet
Firewall
kubectl (user commands)
Node
kubelet
Proxy
docker
Pod
cAdvisor
Pod
container
Pod
container
authentication
authorization
APIs
scheduling
actuator
REST
(pods, services,
rep. controllers)
Scheduler
Scheduler
controller manager
(replication controller etc.)
Master components
Colocated, or spread across machines,
as dictated by cluster size.
Distributed
Watchable
Storage
(implemented via etcd)
Node
kubelet
Proxy
docker
Pod
cAdvisor
Pod
container
Pod
container

Install Kubernetes in Docker



Install Kubernetes

- Copy yum repos and yum.conf
- Install docker on the machine

#yum install docker-engine

Start the docker service

#systemctl start docker; systemctl enable docker

Start the Kubernetes

```
docker run \
    --volume=/:/rootfs:ro \
    --volume=/sys:/sys:ro \
    --volume=/var/lib/docker/:/var/lib/docker:rw \
    --volume=/var/lib/kubelet/:/var/lib/kubelet:rw \
    --volume=/var/run:/var/run:rw \
    --net=host \
    --pid=host \
    --privileged=true \
    --name=kubelet \
    -d \
    gcr.io/google_containers/hyperkube-amd64:v${K8S_VERSION} \
    /hyperkube kubelet \
        --containerized \
        --hostname-override="127.0.0.1" \
        --address="0.0.0.0" \
        --api-servers=http://localhost:8080 \
        --config=/etc/kubernetes/manifests \
        --cluster-dns=10.0.0.10 \
        --cluster-domain=cluster.local \
        --allow-privileged=true --v=2
```

Download and install Kuberctl commandline

- #wget http://storage.googleapis.com/kubernetes-release/release/v${K8S_VERSION}/bin/linux/amd64/kubectl
- #chmod 755 kubectl
- #PATH=$PATH:`pwd`

Create kubernetes cluster configuration:

- #kubectl config set-cluster test-doc --server=http://localhost:8080
- #kubectl config set-context test-doc --cluster=test-doc
- #kubectl config use-context test-doc

Test it

Check the list of nodes running in the Cluster

#kubectl get nodes

Run a application in Kubernetes

- #kubectl run nginx --image=nginx --port=80
- **#docker ps**

Expose it as a service

#kubectl expose deployment nginx --port=80

- Run the following command to obtain the cluster local IP of this service we just created:

#ip=$(kubectl get svc nginx --template={{.spec.clusterIP}})

#echo $ip

Hit the IP in Firefox and try

Kubernetes Pod

- In Kubernetes, a group of one or more containers is called a *pod*. Containers in a pod are deployed together, and are started, stopped, and replicated as a group.
- Pod Definition
  - A pod definition is a declaration of a *desired state*.
  - Desired state is a very important concept in the Kubernetes model.
  - Many things present a desired state to the system, and it is Kubernetes' responsibility to make sure that the current state matches the desired state.
  - For example, when you create a Pod, you declare that you want the containers in it to be running. If the containers happen to not be running

Create POD Definition File

- Vi pod-nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Create and Run a Pod

- #kubectl create -f docs/user-guide/walkthrough/pod-nginx.yaml
- #kubectl get pods
- #curl http://$(kubectl get pod nginx -o go-template=)

Volumes

1.Define a volume:

volumes: - **name**: **redis-persistent-storage emptyDir**: {}

1.Define a volume mount within a container definition:

volumeMounts: *# name must match the volume name below* - **name**: **redis-persistent-storage** *# mount path within the container* **mountPath**: **/data/redis**

Volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-persistent-storage
      mountPath: /data/redis
  volumes:
  - name: redis-persistent-storage
    emptyDir: {}
```

Volumes

Volume Types

- **EmptyDir**: Creates a new directory that will exist as long as the Pod is running on the node, but it can persist across container failures and restarts.
- **HostPath**: Mounts an existing directory on the node's file system (e.g. **/var/logs**)