Cloud computing - Lightweight virtualization & containers

Date @November 26, 2024

Lightweight virtualization

The goal of lightweight virtualization is to create a system that can guarantee the nice properties of computer virtualization (scalability, elasticity, isolation) but that consumes less resources. Lightweight virtualization is appropriate when we cannot accept the overhead of a classical virtualization and we would like to have an isolated environment that is quick to deploy, migrate, scale and dispose with possibly little or no overhead at all.

It uses operating system-level virtualization instead of full hardware virtualization. In case of OS-level virtualization, the hypervisor is the Linux kernel itself, so a dedicated hypervisor is no longer required. Virtual environments can replace classical VMs; they are also called jails or containers, and in them you can have some features for resource management and isolation and apps are executed inside them.

What can we use for lightweight virtualization?

Two main goals are to assign and limit the resources for each virtual environment, and to provide some kind of isolation, such different levels of visibility. This can be achieved by leveraging cgroups and namespaces.

Linux cgroups

A control group (cgroup) is a collection of processes that are bound by the same criteria and associated with a set of parameters or limits. Cgroups were not born for virtualization but for processes management. In fact in general, in OSs processes enter in competition in using resources, such as CPU or memory. Cgroups try to limit the resources that processes can use in advance.

Their functions are:

- resource limiting: group can be set to not exceed a configured memory limit, which also includes the file system cache
- accounting: measures a group’s resource usage which may be used, for example, for billing purposes
- prioritization: some groups may get a larger share of CPU utilization or disk I/O throughput, network bandwidth

- control: freezing groups of processes, their checkpointing and restarting

Linux namespaces

The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. They enable to create distinct virtual environment, such as two namespaces have completely independent networking stacks, virtual interfaces, IP addresses, or two namespaces have visibility on completely independent file systems, such as different /etc folder.

Currently linux implements 7 different types of namespaces:

- mount namespaces: create new mount namespace
- UTS namespaces: create new UTS namespace
- IPC namespace: create new IPC namespace
- PID namespace: create new PID namespace
- user namespaces: create new USR namespace
- CGROUP namespace: create a new cgroup namespace

Limitations of cgroups and namespaces

Cgroups and namespaces have some limitations:

- management complexity: they are very flexible but very difficult to use due to a lot of commands to turn them on
- portability: VMs can be packaged and started on any server, provided that the virtual hardware environment is replicated; but it s not easy to package an isolated app and make it running on another server
- lack of support for data center virtualization: they cannot be used to handle an entire datacenter.

So them can become a nice start, but they should be extended to handle a datacenter at scale.

Containers

LXC containers

Containers are executable units of software in which application code is packaged along with its libraries and dependencies, in common ways so that the code can be run anywhere. To do this, containers take advantage of a form of operating system virtualization in which features of the OS kernel can be leveraged to isolate processes and control the amount of CPU, memory and disk that those processes can access.

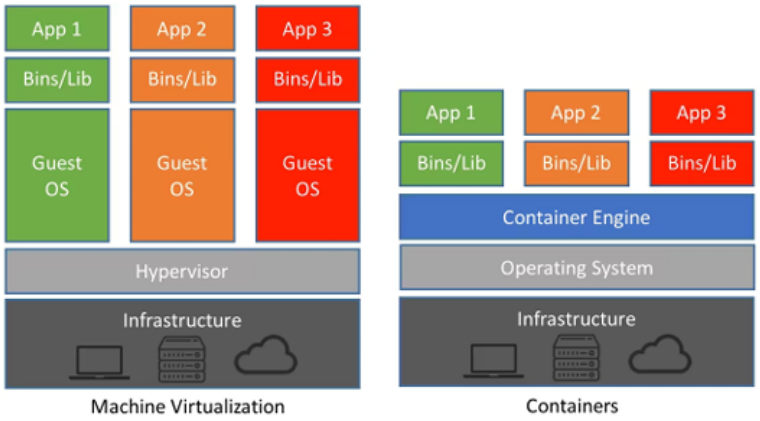
Containers are small, fast and portable because unlike a virtual machine, containers do not need to include a guest OS in every instance and can instead simply leverage the features and resources of the host OS.

Containers group processes together inside isolated containers (to which different resources can be assigned) and they share the same operating system as the host.

On the other end they do not emulate hardware nor run different kernels or OSs.

Containers vs VMs

Containers are faster than real VMs but lighter than VM, because they have less memory, no virtualization overhead etc. VMs provide better isolation, better security due to the limited points of attacks and enable the usage of different OSs.



Use cases for containers

- microservices: containers are small and lightweight which makes them a good match for microservice architectures where applications are cosntructed of many, loosely coupled and independently deployable smaller services
- DevOps: the combination of microservices as an architecture and containers as a platform is a common foundation for many teams that embrace DevOps as the way they build, ship and run software
- hybrid, multicloud: because containers can run consistently anywhere, they are an ideal underlying architecture for hybrid cloud and multicloud scenarios in which organizations find themselves operating across a mix of multiple public clouds in combination with their own data center
- application modernizing and migration: one of the most common approaches to application modernization is to containerize applications in preparation for cloud migration

Microservices

From monoliths to microservices

In the old days of the Internet, we used to build monolith applications but there was no functional distinction between different logical components, it involved difficult software development and long testing process and it was not scalable.

Microservices

Microservices are a software development technique that arranges an application as a collection of loosely coupled services, breaking the monolith used in the past into a more manageable fully decoupled components.

They follow the single-responsibility principle (SRP) of object-oriented design: a software module should be responsible to one, and only one, actor, where actor refers to a group that requires a change in the module.

New challenges arise. One of them is that the traditional mechanism for inter-process communication does not support this paradigm. Another one is about orchestration; we need to simplify how other microservices can be reached and to simplify scaling operations.

To address the latter problem, there are different solutions:

- manual deployment and orchestration: it is simple, available everywhere but not automated and not scalable
- via scripting: it integrates with existing environments but it is not scalable and manual scheduling is needed
- get rid of human intervention, using a dedicated orchestrator, so it is automated, reproducible, self-healing, scalable but it brings some overhead and requires new tooling and training

Orchestration goals

An orchestrator must:

- handle the container
- scheduling, which is about matching containers to machines depending on the resource needs, affinity requirements or labels
- replication, which is about running N copies
- handle machine failures
- discovery, that is about finding peers and services in other containers

- virtual networking

It would be nice if they had:

- built-in load-balancers
- automated updates
- cluster auto-scaling
- provisioning storage
- automatic restart
- late-binding configuration

Docker

When we talk about Docker, we do not refer specifically to the orchestrator, but we refer to a complete platform and tool called container management. The term docker means a virtual environment (a container) but it is something more. However, dockers are a more advanced version of the traditional LXC containers and at the same time the docker platform tries to simplify the way the user can instantiate and interact with the running containers.

Dockers offer improved and seamless container portability, running without modification across any desktop, data center and cloud environment. Moreover, they offer even lighter weight and more granular updates. A docker also include an orchestrator, which provides automating container creation, container versioning and shared container libraries.

Docker file

The docker file is a sort of script which automates the processo of Docker image creation. It is a list of command-line interface (CLI) instructions that Docker engine will run in order to assemble the image.

Docker images

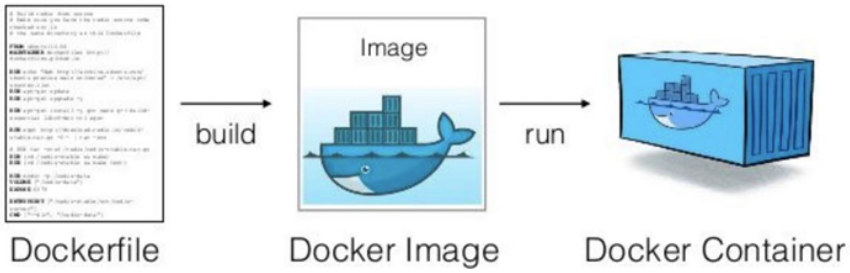
Docker images contain executable application source code as well as the tools, libraries and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance of the container.

It is possible to build a Docker image from scratch, but most developers pull them down from common repositories called docker registries. Multiple docker images can be created from a single base image, and they'll share the commonalities of their stack.

Docker images are made up of layers and each layer corresponds to a version of the image. Whenever a developer makes changes to the image, a new top layer is created, and this top layer replaces the previous top layer as the current version of the image. Previous layers are saved for rollbacks or to be re-used in other projects.

Docker containers

Docker containers are the live, running instances od Docker images. While Docker images are read-only files, containers are executable content. Users can interact with them and administrators can adjust their settings and conditions using Docker commands.

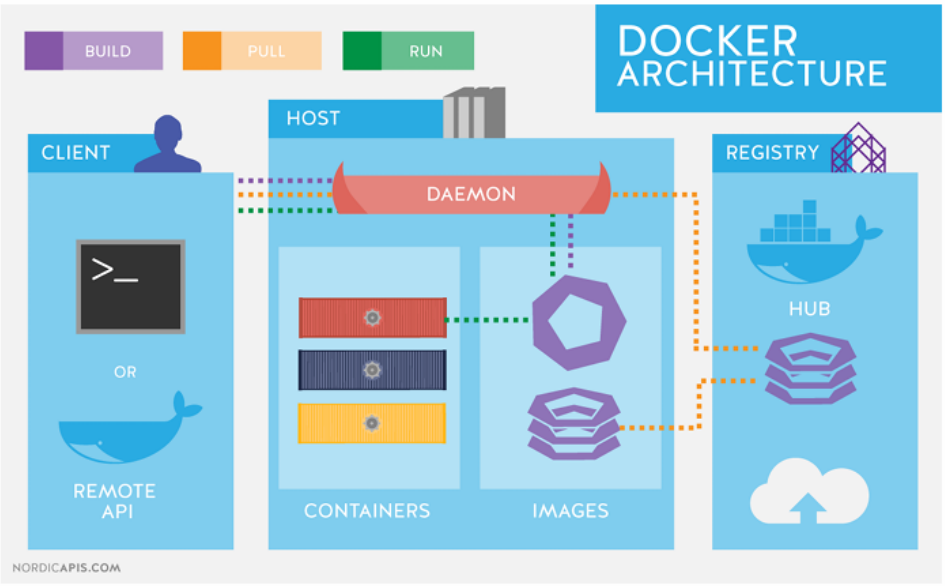


Docker daemon

The Docker daemon is a service that creates and manages Docker images, using the commands from the client. Essentially it servers as the control center of you Docker implementation. The server on which Docker daemon runs is called the Docker host.

Docker registry

A Docker registry is a scalable open-source storage and distribution system for Docker images. The registry enables you to track image versions in repositories using tagging for identification. this is accomplished using git, a version control tool.



Docker Swarm

Docker Swarm is an orchestrator, it is integrated with Docker and mainly developed by that company. It is not used so much nowadays and it is being replaced by Kubernetes.

Kubernetes

Kubernetes is a container orchestrator and it is able to schedule the containers on the different machines, to manage and to scale the containerized applications.

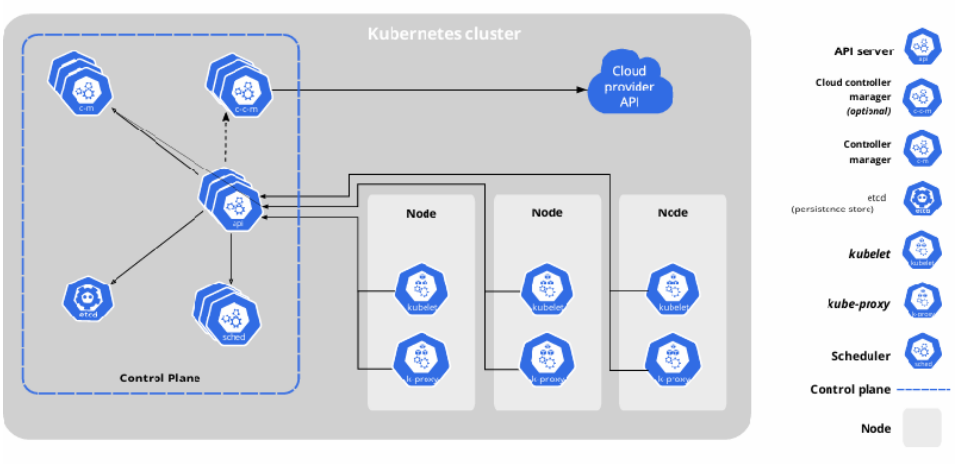
It is actually the standard for containers orchestrator, it is more complex but more features as well.

It is a dynamic system that manages the deployment, management and interconnection of containers on on a fleet of worker servers. The Kubernetes architecture is based on clusters.

Clusters

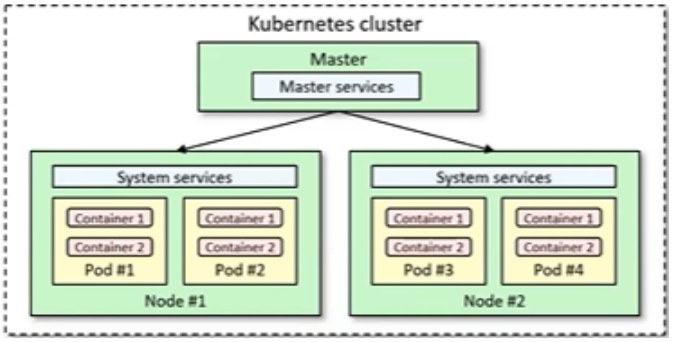
Clusters are the building blocks of Kubernetes architecture, they're made up of nodes, each of which represents a single compute host. Developers manage cluster operations using kubectl, a command-line interface that communicates directly with the Kubernetes API.

Each node consists of a master node that servers as the control plan for the cluster, and multiple worker nodes that deploy, run and manage containerized applications. The master node runs a scheduler service that automates when and where the containers are deployed based on developer-set deployment requirements and available computing capacity. Each worker node includes the tool that is being used to manage the containers and a software agent called a Kubelet that receives and executes orders from the master node



Pod

A Pod is the smallest and simplest unit in the Kubernetes object model that you can deploy or create. It may be composed of one or more containers that work together to perform a given function, and a pod is mandatorily executed on a single node. In Kubernetes a pod is the same as a process in an operating system.



A user can ask request that some containers allocated in a pod can be deployed thanks of a fundamental principle of Kubernetes that is the principle of declarative configuration.

```yaml
apiVersion: v1
kind: Pod
metadata:
   labels:
    app: nginx
   name: nginx-5f78746595
   namespace: default
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
      - containerPort: 8080
        name: http
        protocol: TCP
      volumeMounts:
      - mountPath:/var/log/
  volumes:
      - emptyDir: {}
        name: logs
```

Kubernetes deployments

Pods are typically rolled out using Deployments, which are objects defined by YAML files that declare a particular desired state. Once created, a controller on the Control Plane gradually brings the actual state of the cluster to match the desired state declared in the Deployment by scheduling containers onto nodes as required. Deployment files can be continuously edited and updated by the user. Every time a user modify a YAML file he has to perform an API call.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 8080
```

Kubernetes services

The Kubernetes service is an entity which groups some pods (possibly with similar functions) and assigns the same static IP address and the same port to which this service can be reached.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
  type: ClusterIP
  clusterIP: 10.96.0.100 #
Assign a specific ClusterIP
to the service
```

If there are multiple pods, a Kubernetes service performs load balancing techniques in order to balance the traffic to the different ports.

Pod management: Kubernetes node agents

To start and manage pods and their containers on worker machines, nodes run an agent process called kubelet which communicates with a kube-apiserver on the Control Plane.

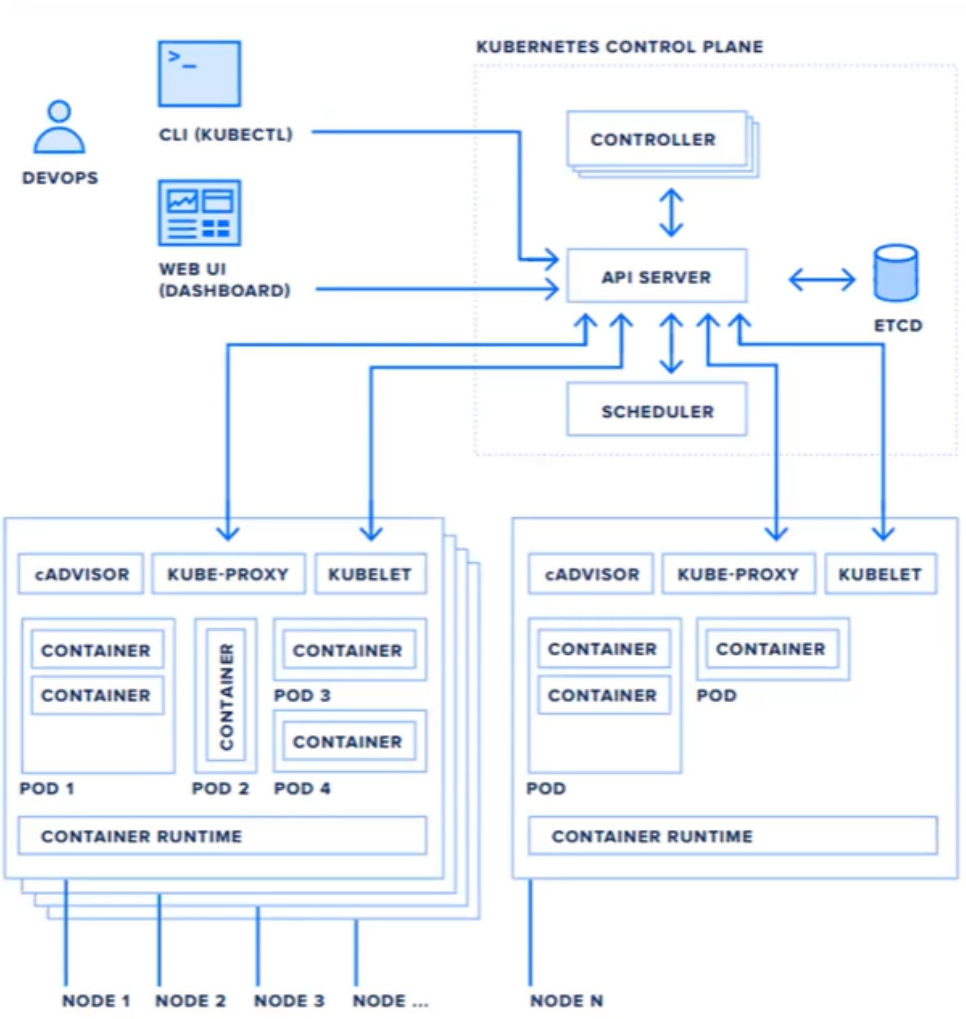
Typically a pod has a container runtime agent like a Docker daemon and a kube-proxy which manages network rules on the host.

Network policy

Network policies are entities that define what type of communication can be accepted by each pod depending on values like the IP addresses and ports, labels on pod and namespace selector.

Kubernetes control plane

The Kubernetes Control Plane oversees the nodes and manages their scheduling and maintains their workloads. It includes the kube-apiserver front-end, which is the most important element because it receives API calls and interacts with the Kubectl of the worker node. Since there is a distributed architecture of worker nodes, the Control Plane must have a state of the distributed architecture and this is achieved with the key-value store named etcd, which stores all the cluster data. There is also a kube-scheduler which schedules pods to nodes, and a set of controllers to continuously observe the state of the cluster and drive its actual state towards the desired state.



Functions of the Control Plane

Kubernetes schedules and automates container-related tasks throughout the application lifecycle, including the following.

Rolling Deployments

Rolling deployments is the capability to continuously update the state of a deployment to the state desired by the user and this can be done because the user can continuously update the deployment file and the master node will decide how to rollout the changes.

In general when the master node has to perform the rolling deployment the goal is to minimize the downtime and make the application in the Kubernetes cluster continues to be available to clients or external users. For this purpose there different techniques used by the master node and one of the most used is the blue-green deployment, where the idea is that if you have a new state to which you want to target, such as a new version of a container, you don't stop the current version in order to update the software internally but you keep the first container running as before, in the meantime you run the second container (or the second pods with the list of containers) and only when the new container is totally operative you switch the traffic from going to the previous entity to the new one. If resources are not available you can't perform this solution.

Service discovery

In Kubernetes an element of the cluster may be able to interact with another by simply using the DNS system implemented inside the Kubernetes cluster itself of some other mechanism such as local environment variables.

Load balancing

When you specify a deployment with n replicas of a pod, the load balancer (a service) will be in charge of deciding how to redirect the traffic coming outside the cluster to the different replicas of the pod.

Health checking

Self-healing (that is health checking) is employed in the event a containers fails, so that Kubernetes can restart or replace it automatically to prevent downtime. It can also take down containers that don't meet your health-check requirements.

Cluster networking

Kubernetes is in charge of creating a networking inside the cluster because clusters also need to connect running applications and containers to one another across machines, using IP addresses and assignment of network addresses to cluster members and containers.

Autoscaling

Kubernetes can decide to allocate more containers when it notices that there is a growth in demand going to a specific application. This can be done by the user by editing the

deployment file or by Kubernetes by analyzing metrics related to the already deployed containers.

Declarative configuration

In declarative configuration the user declares which desired state they would like for a given application (such as 4 running containers of an NGINX web server), and the system takes care of achieving that state by launching containers on the appropriate members, or killing running containers.

On the contrary an imperative configuration would require developers to explicitly define and manually execute a series of actions to bring about the desired cluster state, which can be error-prone, making rollbacks difficult.