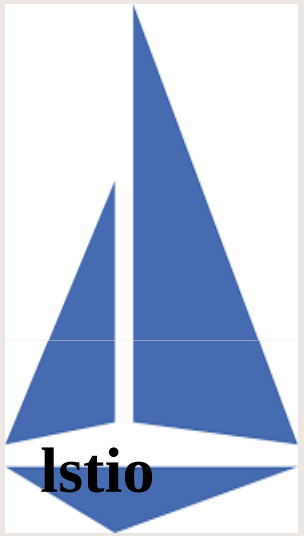
Copyright Notice

Overview on Docker, Kubernetes and Istio



Prakash Badhe

prakash.badhe@vishwasoft.in

Agenda

- Docker introduction and installations
- Docker usage
- Docker architecture
- Deploy applications with Docker
- Docker applications to share resources
- Manage Docker images with Docker Hub
- Clustering with Docker Swarm
- Docker Monitoring and management
- Kubernetes architecture and usage
- Kubernetes Cluster
- Deployment in the Cloud
- Service mesh Istio overview

Setup : Ubuntu Linux

- Linux Ubuntu 16.0 onwards
- Hardware virtualization enabled in BIOS
- Oracle VM Box for Kubernetes.
- Firefox Mozilla browser
- Adobe PDF reader.
- Live internet connection with download permissions

Setup : Windows10 64 bit

- Windows Build version latest
- Hardware virtualization enabled in BIOS
- Windows 10 Pro or Enterprise 64 bit
- Docker for Windows
- Google Chrome/Firefox Mozilla browser
- Adobe PDF reader.
- Live internet connection with download permissions

Windows 7 and Mac

- Hardware virtualization enabled in BIOS
- Windows7/Mac 64 bit
- Docker ToolBox for Windows/Mac
  - Installs VM and Docker on VM.
- Google Chrome/Firefox Mozilla browser
- Adobe PDF reader.
- Live internet connection with download permissions

It works on My Machine..

The application developed and tested in development environment fails in production number of times.

Issues in deployment

- Application debug/release versions
- OS version/patches/service pack not matching
- Memory availability
- Port conflict
- CPU and hardware configuration
- Dependent libraries/Runtime environment,. net framework, class lib etc. absent or mismatch versions.
- Database missing or not matching
- Application/Network settings
- User rights and permissions

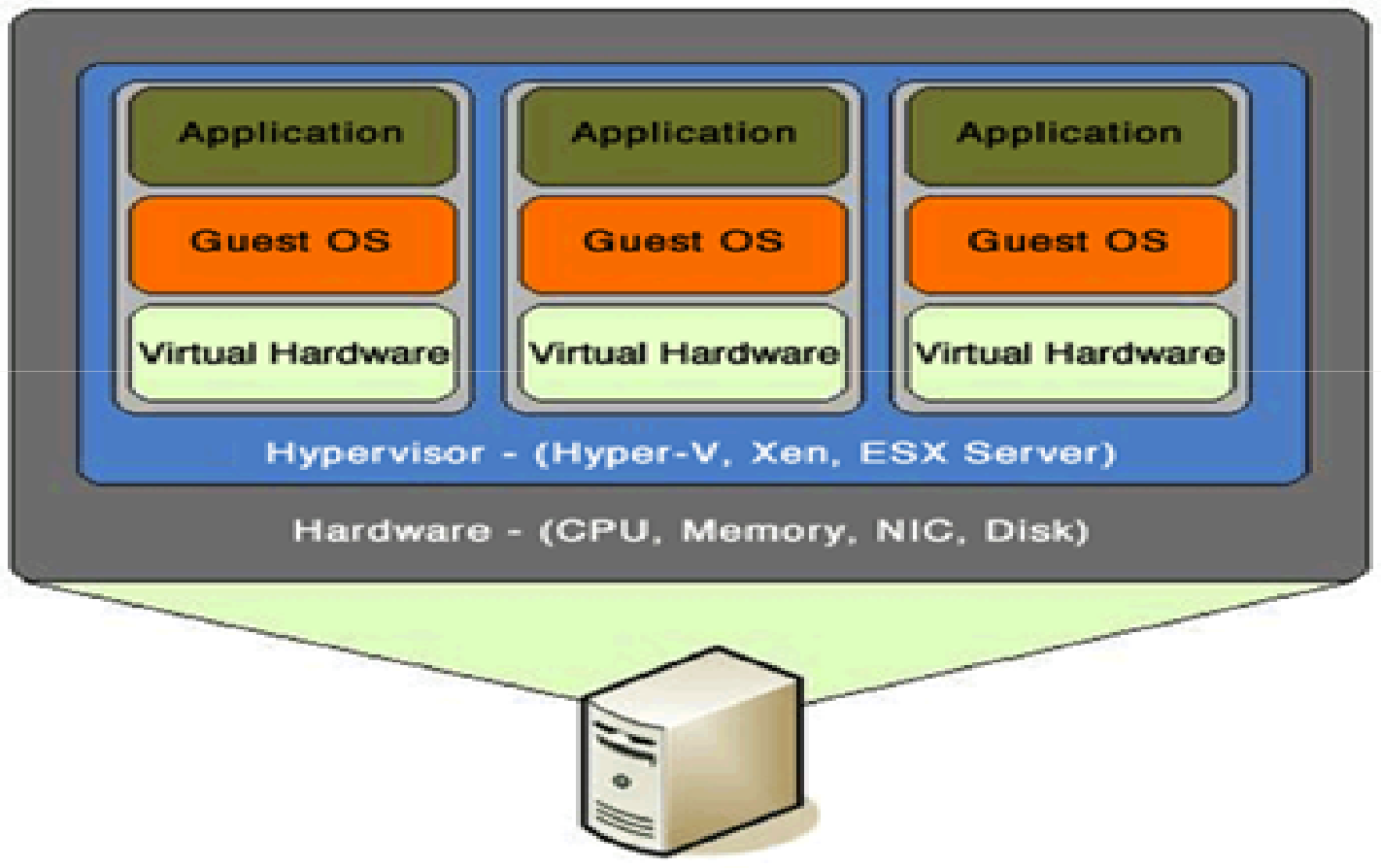
Matching the Deployment

- Can we have the matching environment for development as well as production.
  - Debug/Release, Runtime and OS are different in production.
- Can we have the SAME environment for development as well as production.
  - Visual Studio, JDK etc.not needed in production
  - Higher capacity hardware needed in production.
- Simulate the hardware
  - CPU,Memory ,Hard disk space etc.

Virtual Environment

- Hardware Virtualization enabled in the CPU
- Software Virtualization by Virtual Machine(Hypervisor)
- Examples
  - Linux on top of Windows
  - Windows on Solaris

Create different environment with the same hardware and OS



Virtual machine on top of base OS

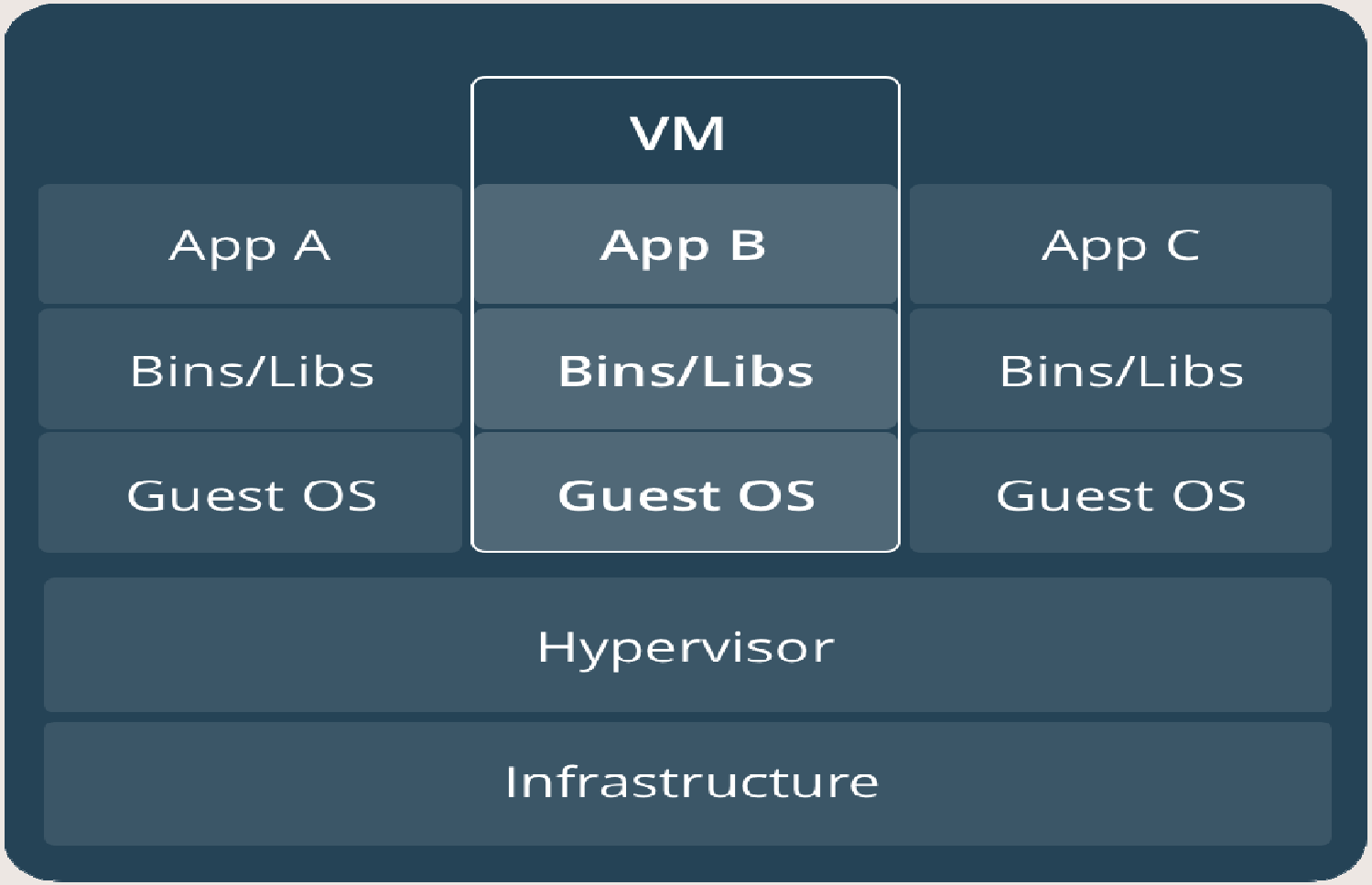
Virtual Machine

- Oracle VirtualBox is one of the native application that creates the virtual OS on top of windows and Linux.
- VMWare is another one.

VM and applications

- Virtual machines run on top of base operating systems with additional OS layer in each box.
- The VMs needs more resources and memory.
- **The VMs are isolated from other VMs on the same OS and base OS.**
- **Inter-communication and sharing is not possible**
- The application disk image and application state depends on OS settings, system-installed dependencies, OS security patches, and other hard-to-replicate issues.

Applications on VM



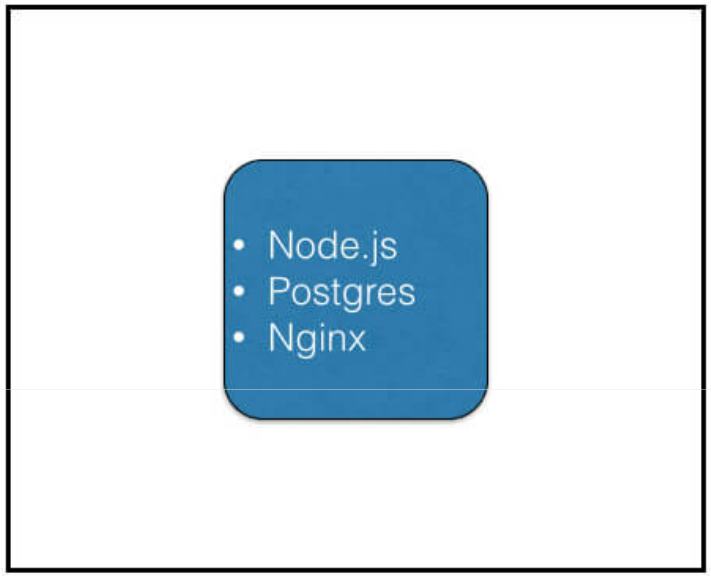
VM limitations

- Performance is slow
- Data sharing is limited.
- Costly in terms of hardware and resources.
- Not convenient for end users
- Security can be compromised.
- Compatibility with base OS and hardware
- Maintaining applications and environment needs skilled manpower and resources.

Shipping Containers

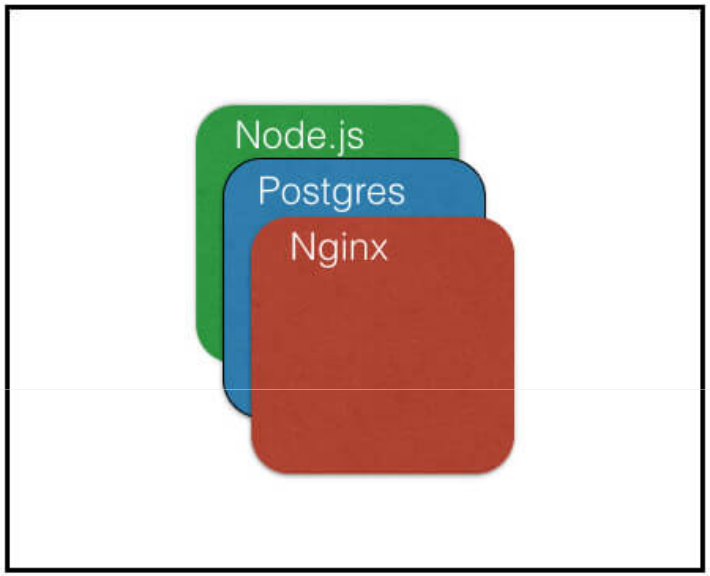
This container can contain any different types of items.

Software Application Containers



OS containers

- Meant to used as an OS - run multiple services
- No layered filesystems by default
- Built on cgroups, namespaces, native process resource isolation
- Examples - LXC, OpenVZ, Linux VServer, BSD Jails, Solaris Zones



App containers

- Meant to run for a single service
- Layered filesystems
- Built on top of OS container technologies
- Examples - Docker, Rocket

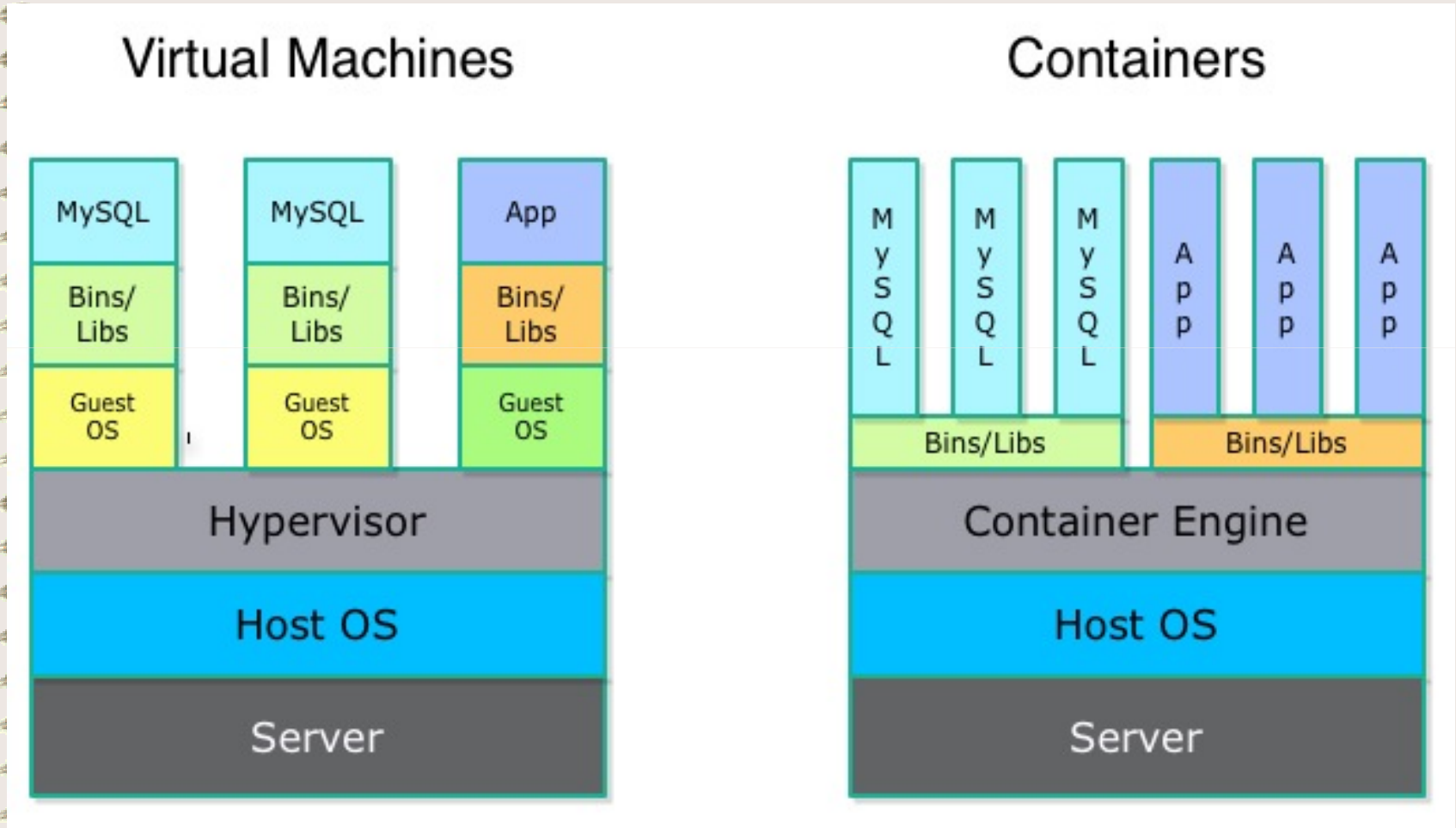
Application Container

- Containers are a way to package software in a format that runs completely isolated on a shared operating system.
- Unlike VMs, containers do not bundle a full operating system - only libraries and settings required to make the software work are needed.
- This makes for efficient, lightweight, self-contained systems and guarantees that software will always run the same, regardless of where it’s deployed.

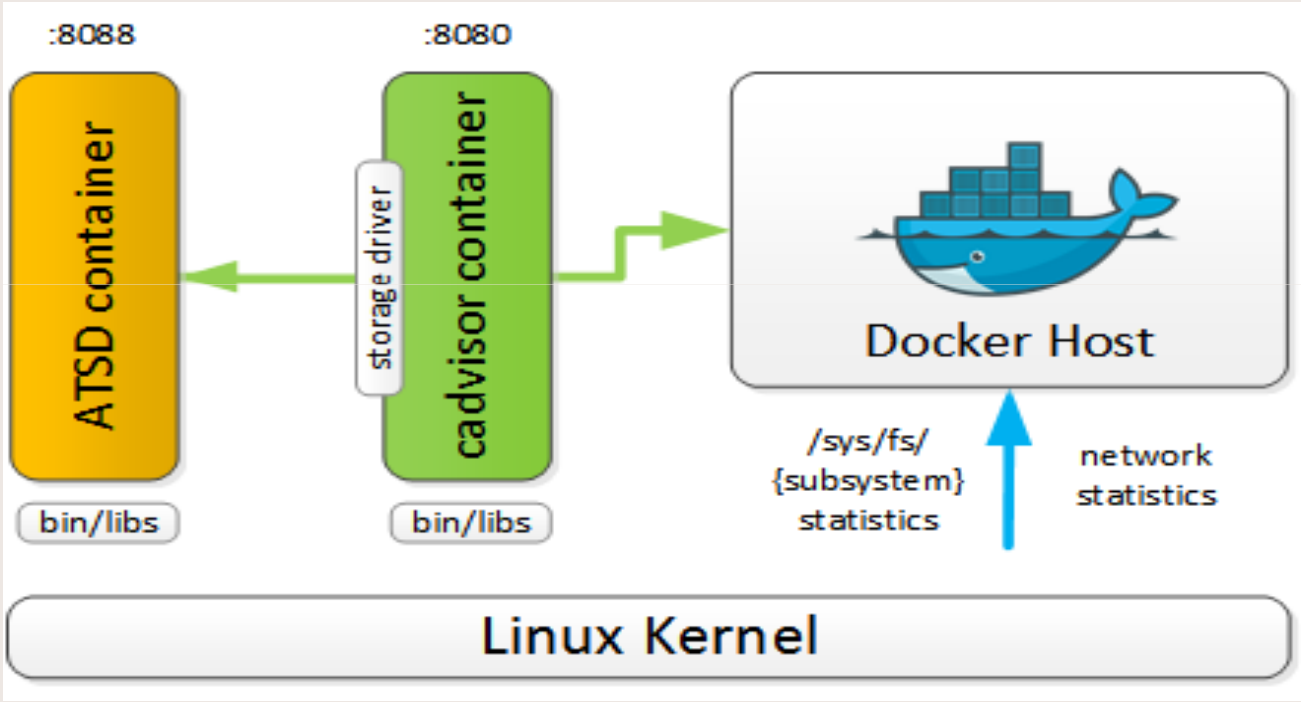
Container on Unix

- Containers can share a single kernel, and the only information that needs to be in a container image is the executable and its package dependencies, which **never need to be installed** on the host system.
- These container processes run like native processes, and you can manage them individually by running , just like you would run commands on Unix/Linux terminals.
- Since the containers contain all their dependencies, there is **no configuration conflicts and issues** with host OS.
- **A containerized app “runs anywhere.”**

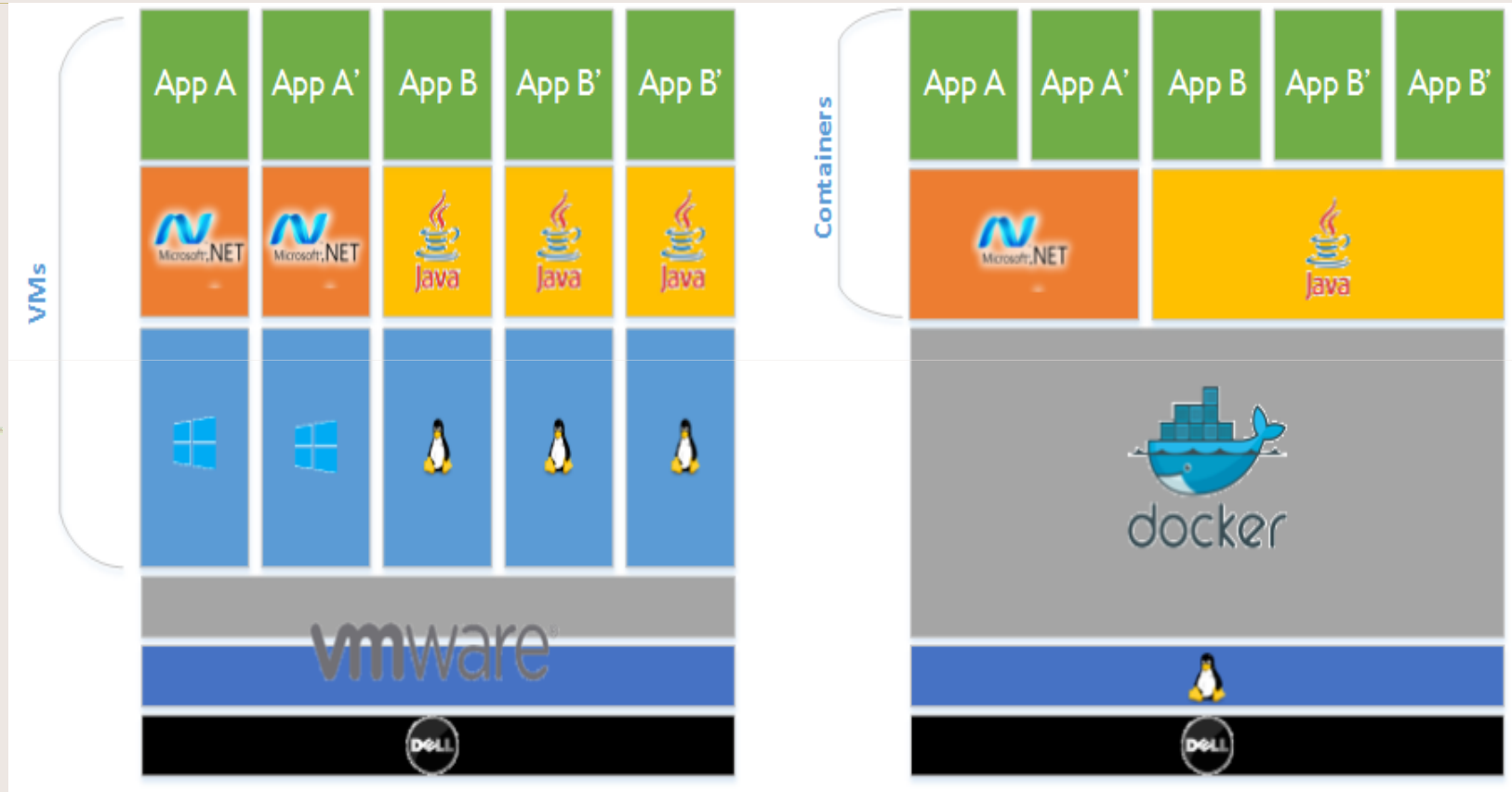
VM vs. Container Engine



Docker Container Engine



Apps on Docker vs. VM



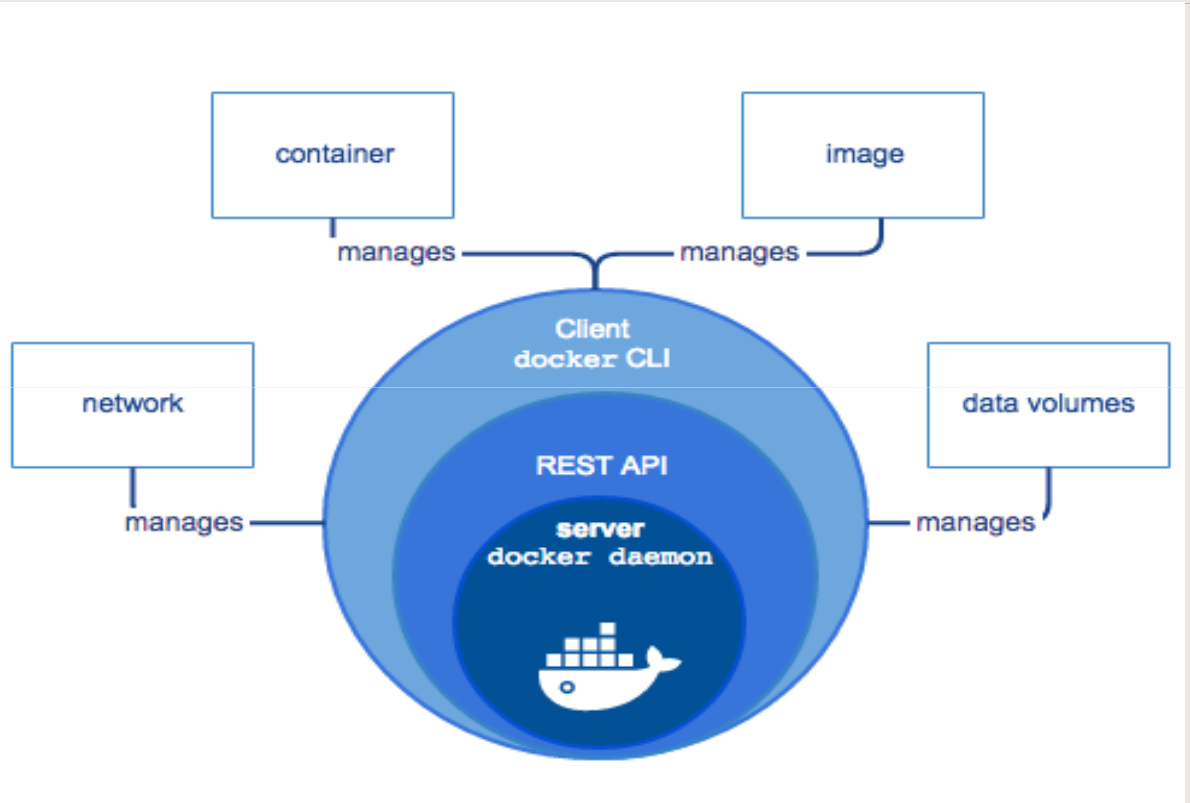
Docker Features

- Docker has the ability to package and run an application in a loosely isolated environment called a container.
- The isolation and security allow to run many containers simultaneously on a given host.
- Containers are lightweight because they don’t need the extra load of a hypervisor, but run directly within the host machine’s kernel.
- This means you can run more containers on a given hardware combination than if you were using virtual machines.
- You can even run Docker containers within host machines that are actually virtual machines!

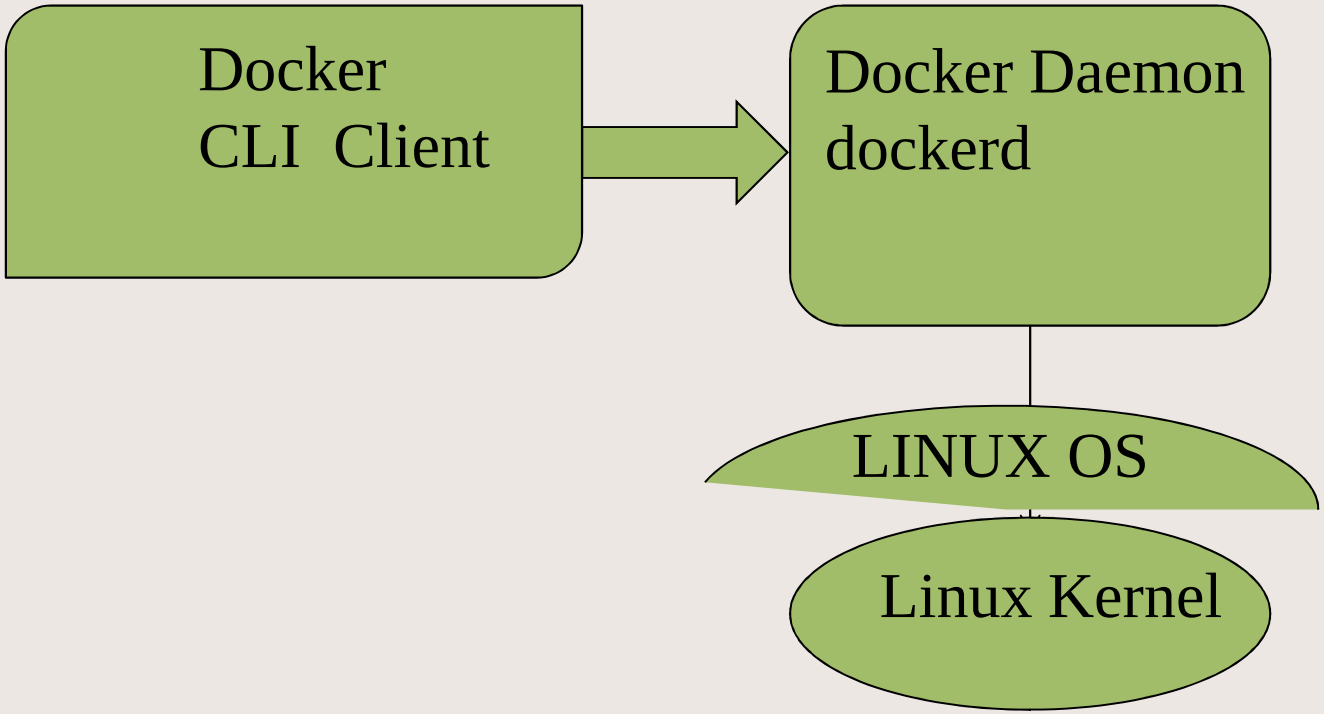
Docker Container Services

- Container instance management
- Cluster with Docker containers
- Load balancing across containers
- Monitor and Manage the container operations
- Sharing the data with containers

Docker engine



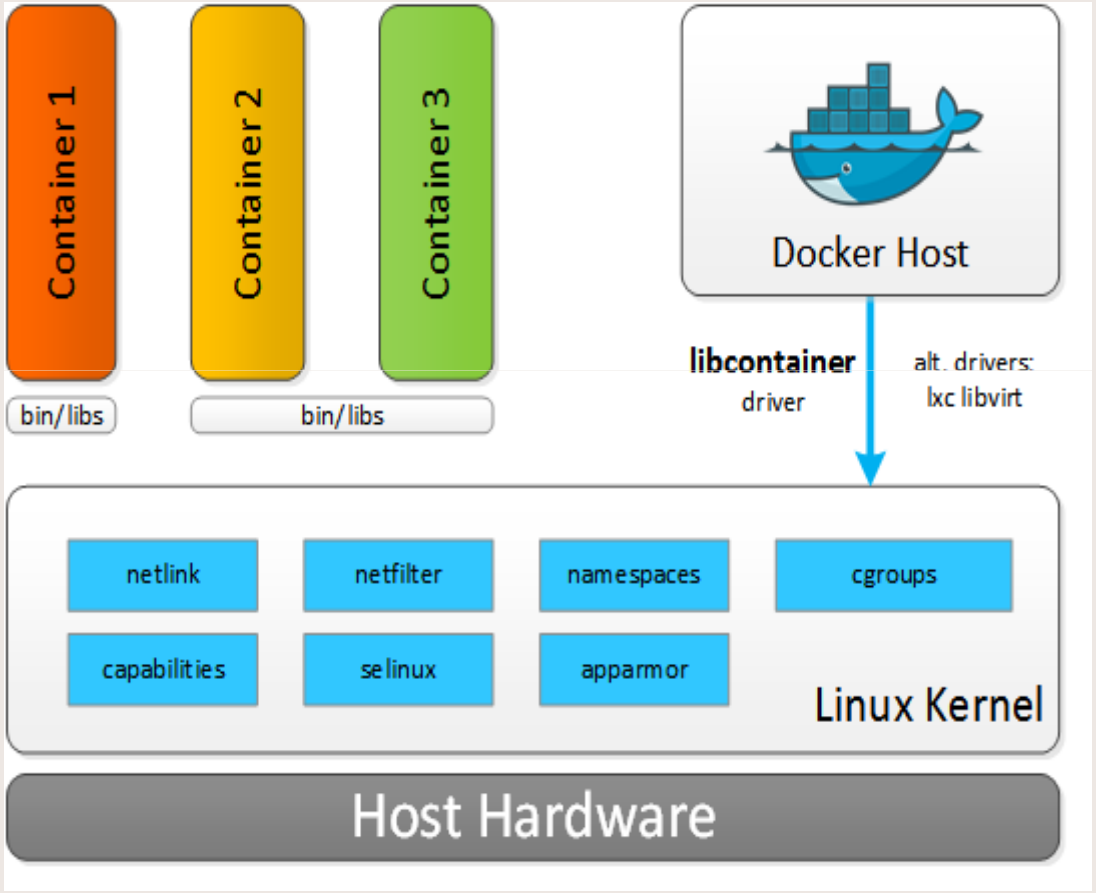
Docker Process



Docker CLI

- The “Docker” mean **Docker Engine**, the client-server application made up of the Docker daemon, a REST API that specifies interfaces for interacting with the daemon.
- It is a command line interface (CLI) client that talks to the daemon (through the REST API wrapper).
- Docker Engine accepts docker commands from the CLI, such as **docker run <image>, docker ps** to list running containers, **docker image ls** to list images, and so on.

Docker Container apps



Applications in Docker

- Applications are run inside docker as **containers**.
- The container is a mini-virtualized environment created in-memory by docker engine.
- The container contains all the dependent parts required to run the application, including libraries, files and Operating System(in optimized format).
- The container is analogous to a small machine running the application in self contained manner.
- One container is completely isolated from other containers.
- If required, one container can connect to another container like front-end container and database container.

Docker Container image

- The applications deployed in Docker are based on configuration called as images.
- The container is created based on image configuration and runs application as separate instance.
- Multiple instances of the container can be created from the same image.
- The docker image configuration is defined in a file Dockerfile.
- These images can be located on local machine or pulled from docker registry server and placed locally.

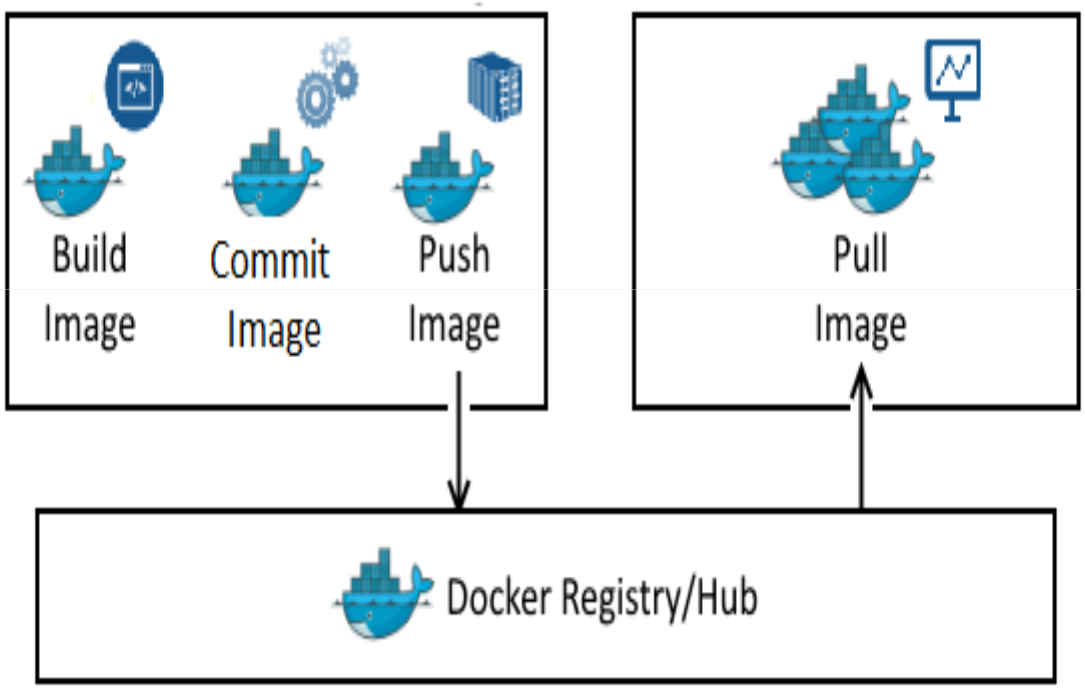
Sharing the Images

- The configuration for container is defined as image which includes the application, dependencies and network configuration.
- This image is used to start the container.
- Multiple instances of this image forms multiple isolated containers.
- These images are stored at central place and downloaded and maintained by docker server on local machine.

Docker Hub Registry

- The ‘https://hub.docker.com/ ‘ is the registry/repository that hosts thousands of docker application/container images.
- This is the storage and content delivery system, holding named **Docker** images, available in different tagged versions.
- The docker allows to pull and push the images to and from the registry.

Docker Image registry



Docker usage

- Docker is the world's leading open source software container platform.
- Developers use Docker to eliminate "works on my machine" problems when collaborating on code with co-workers.
- Operators use Docker to run and manage apps side-by-side in isolated containers to get better compute density.
- Enterprises use Docker to build agile software delivery pipelines to ship new features faster, more securely and with confidence for both Linux, Windows Server, and Linux-on-mainframe apps.

Apps Delivery with Docker

- Docker streamlines software delivery.
- Develop and deploy bug fixes and new features without roadblocks.
- Scale applications in real time
- Docker comes with built-in swarm clustering that’s easy to configure.
- Test and debug apps in environments that mimic production with minimal setup.

Universal deployments

- Docker is the universal container management system that helps to deploy the applications with any dependencies, any os settings without any issues and conflicts.
- Docker save up to 10X in personnel hours in app maintenance and support.
- Docker makes it easy to deploy, identify, and resolve issues and reduce overall IT operational costs.
- It reduces the downtime when deploying updates or quickly roll back with minimal disruption

Dev and Production

- Docker provides tooling and a platform to manage the lifecycle of the containers:
- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you’re ready, deploy your application into your production environment, as a container or an orchestrated service.
- **This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two**.

Applications with Docker

- Docker automates the repetitive tasks of setting up and configuring development environments so that developers can focus on what matters: building great software.
- Developers using Docker don’t have to install and configure complex databases nor worry about switching between incompatible language toolchain versions.
- When an app is dockerized, that complexity is pushed into containers that are easily built, shared and run.

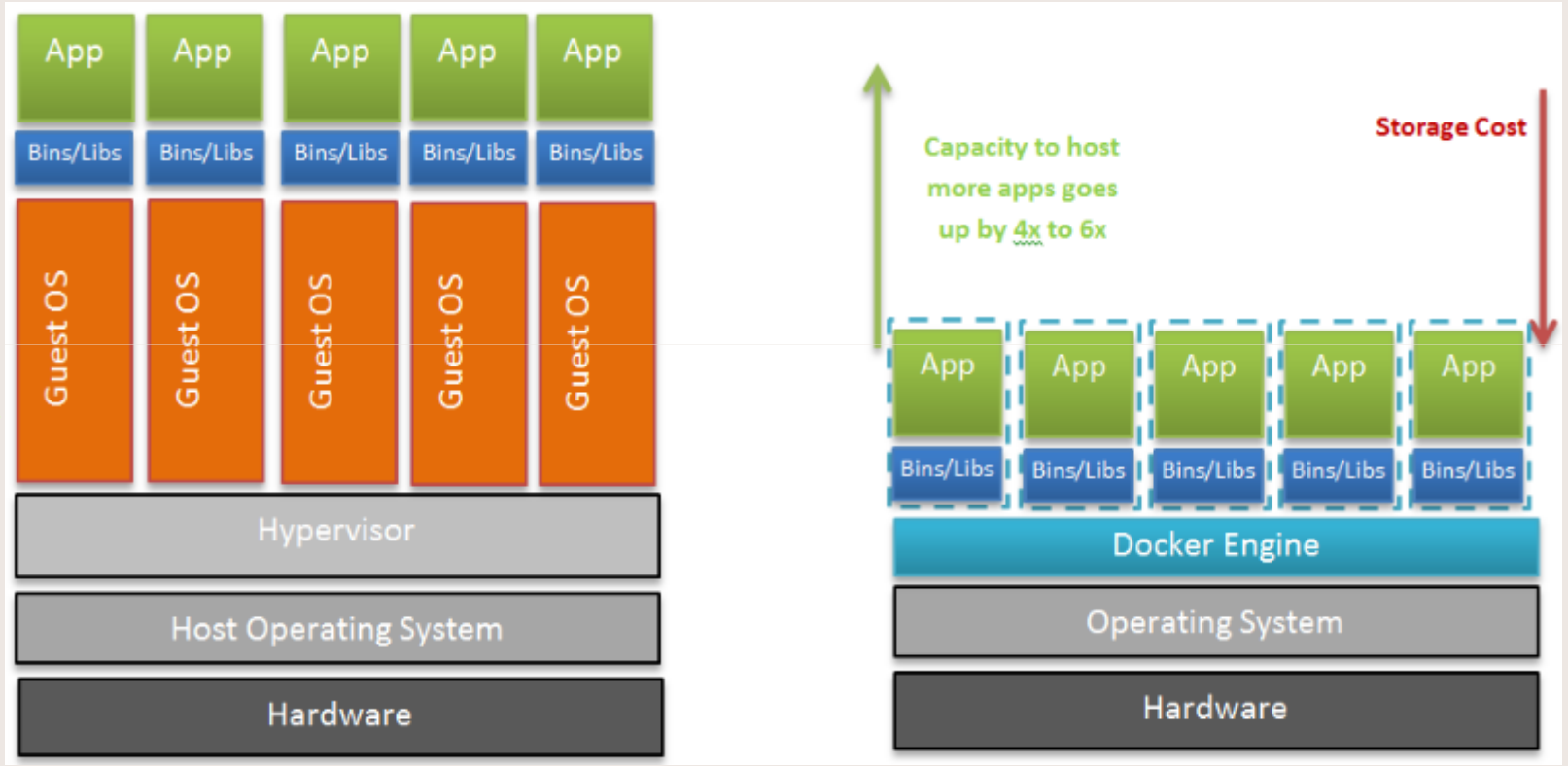
Add Support Team

- On boarding a new co-worker to a new codebase no longer means hours spent installing software and explaining setup procedures.
- Code that ships with Dockerfile images is simpler to work on: Dependencies are pulled as neatly packaged Docker images and anyone with Docker and an editor installed can build and debug the app in minutes.

Simplify with Docker

- Docker enables the developers and IT ops teams everywhere, allowing them to build, ship, test, and deploy apps automatically, securely, and portably with no surprises.
- No more wikis, READMEs, long run-book documents and post-it notes with stale information.
- Teams using Docker know that their images work the same in development, staging, and production.
- New features and fixes get to customers quickly without hassle, surprises, or downtime.

Application scaling with docker



Scaling the Containers

- Applications can be scaled up to thousands of nodes and containers.
- Docker containers spin up and down in seconds, making it easy to scale application services to satisfy peak customer demand, and back down when demand reduces.

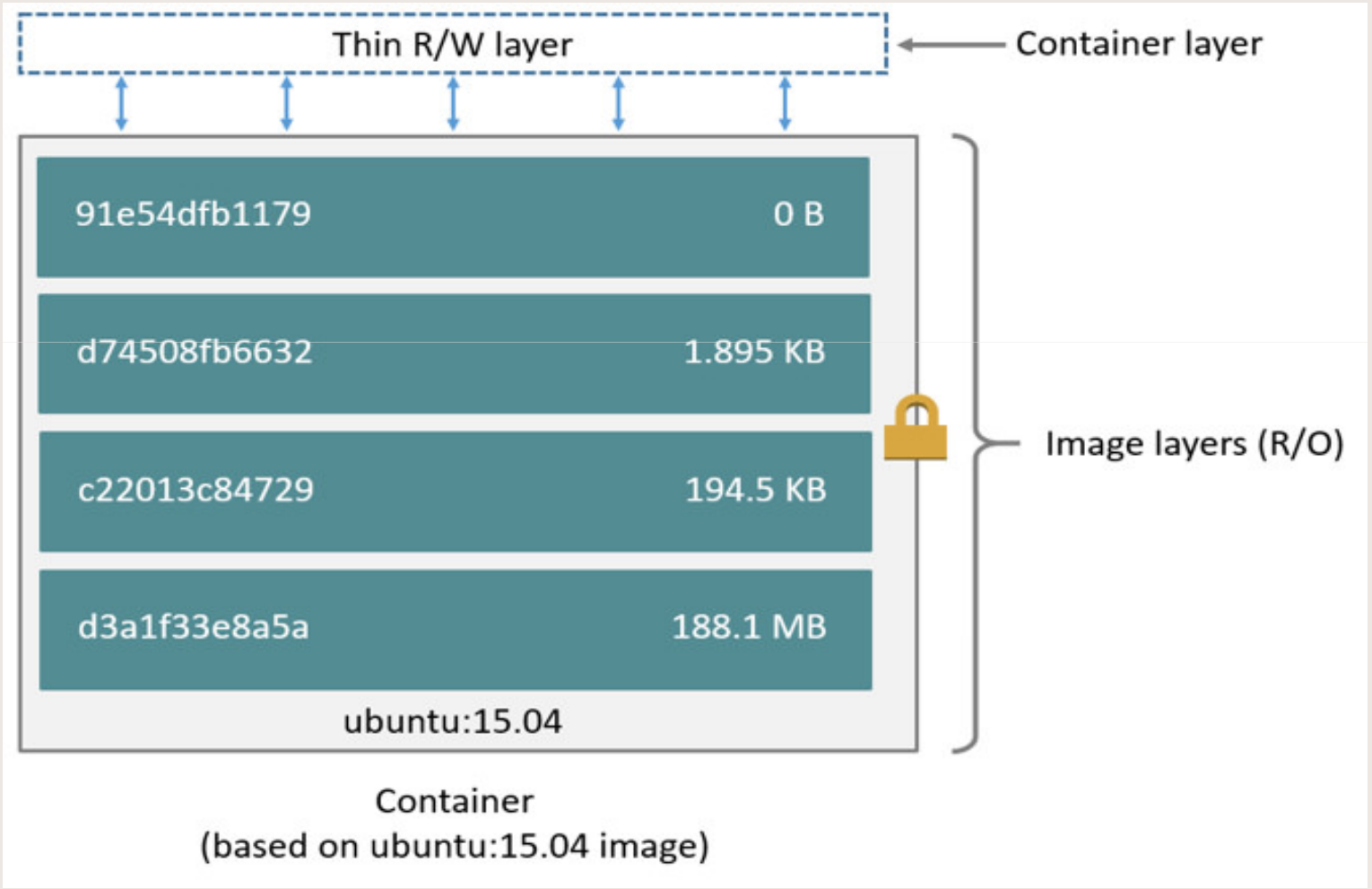
The layers

- The four commands, the FROM statement starts out by creating a layer from the ubuntu:16.04 image.
- The COPY command adds some files from the Docker client's current directory.
  The RUN command builds the application using the make command.
- Finally, the last layer specifies what command to run within the container.

Layers on Stack

- The layers are stacked on top of each other.
- When you create a new container, you add a new writable layer on top of the underlying layers.
- This layer is often called the “container layer”.
- All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

Layers in Container



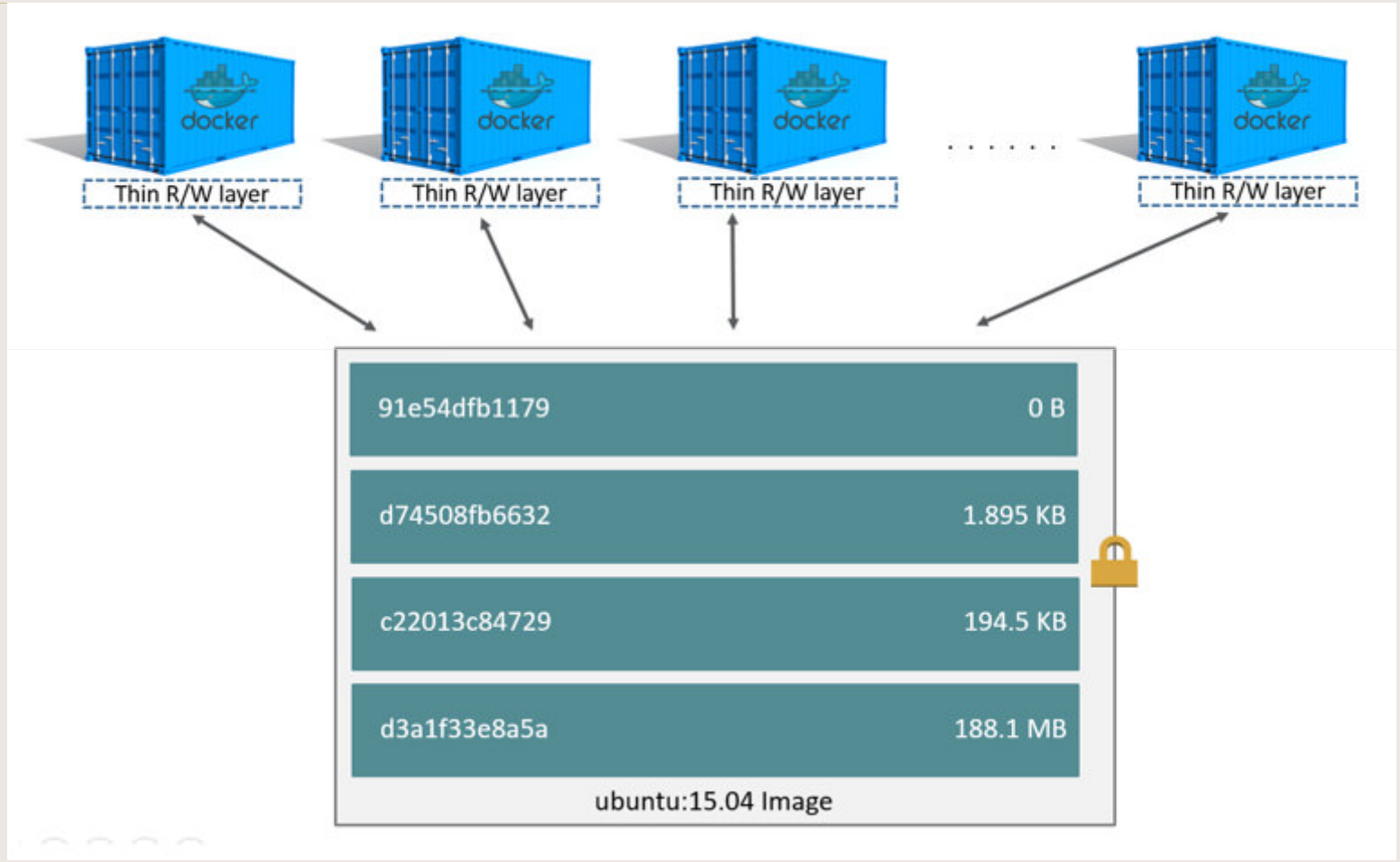
Container and layers

- The major difference between a container and an image is the top writable layer.
- All writes to the container that add new or modify existing data are stored in this writable layer.
- When the container is deleted, the writable layer is also deleted.
- The underlying image remains unchanged.

Sharing layers

- Each container has its own writable container layer, and all changes are stored in this container layer.
- The multiple containers can share access to the same underlying image and yet have their own data state.
- The diagram next shows multiple containers sharing the same Ubuntu 15.04 image.

Sharing across containers



Storage Driver

- The *storage driver* handles the details about the way these layers interact with each other.
- Different storage drivers are available as per the OS configurations.

Dockerfile

- Dockerfile defines what goes on in the environment inside the container.
- Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of the system, so you have to map ports to the outside world, and be specific about what files you want to “copy in” to that environment.
- The build of the app defined in this Dockerfile behaves exactly the same wherever it runs.

Docker Image File

- The Dockerfile content

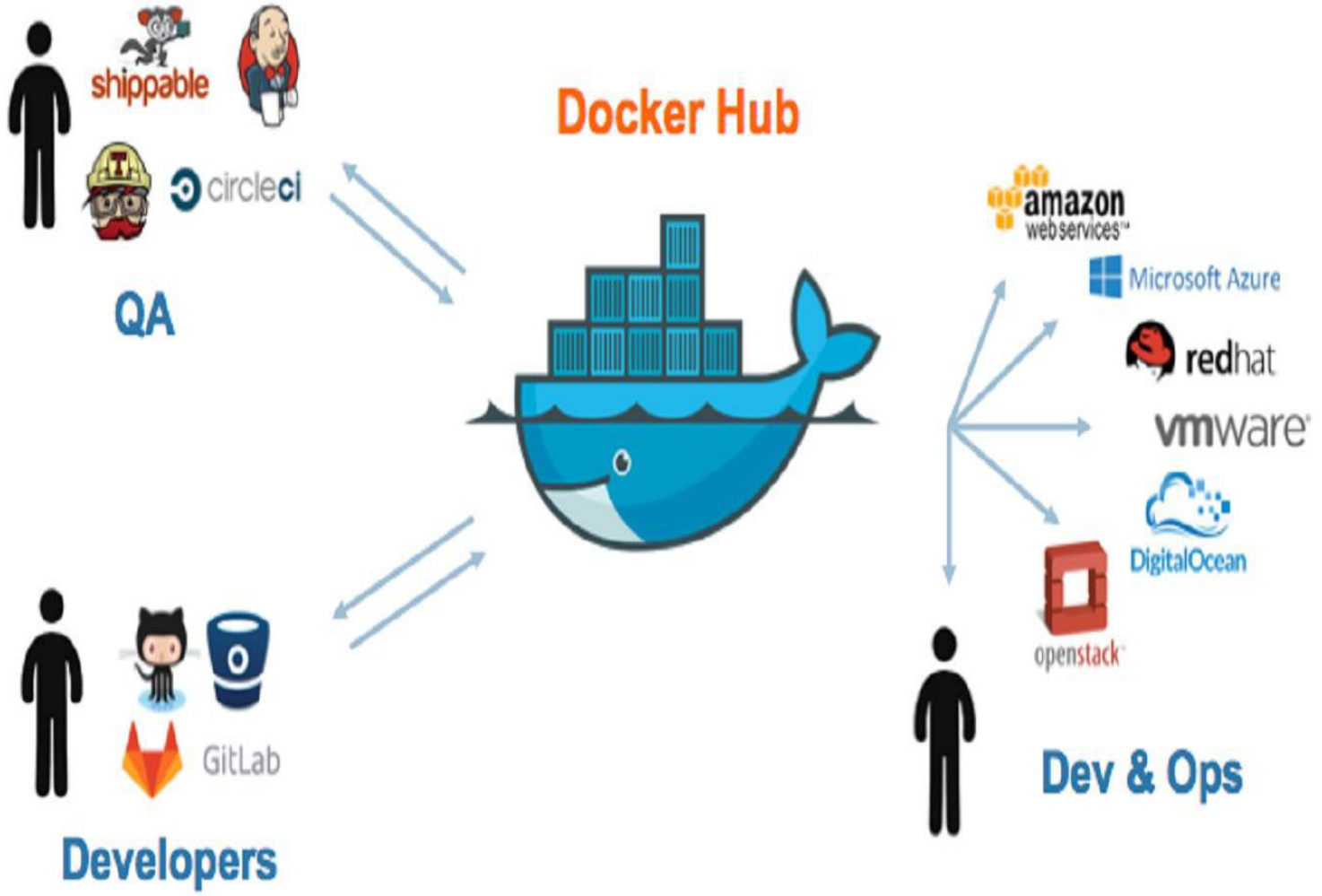
- FROM ubuntu:16.04
- COPY . /app
- RUN make /app
- CMD python /app/app.py

- The four commands create layers on the docker host system.

Sharing images

- Build locally new image
  - Test it locally
  - Push to docker Hub
- Pull exiting image from Docker Hub
  - Modify and build it.
  - Push back to Docker Hub
- Commit the existing Container to an image
  - Push the new image.

Images Sharing



More Docker Tools

- Docker-compose
- Docker-Machine
- For installation refer the installation reference.

Container configuration

- Instead of running all the commands from console one by one we can define the container configuration in the **docker-compose.yml** file.
- The docker-compose is the tool used to deploy the applications in the containers defined in the docker-compose.yml files.

Docker-compose

- The Docker-Compose is a tool for defining and running multi-container Docker applications.
- Define your container application configuration in a YAML/YML file to configure your application’s services.
- With a single command, you create ,start and stop all the services from the compose yml configuration.
- The same compose yml file is used as configuration **docker swarm** cluster configuration.

Container services

- Services are just “containers in production.”
- A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on.
- Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.
- Easy to define, run, and scale services with the Docker platform with-docker-compose.yml file.

Service Object

- The service is the image for a microservice within the context of some larger application.
- Examples of services might include an HTTP server, a database, or any other type of executable program that you wish to run in a distributed environment.
- Create a service, specify which container images to use and which commands to execute inside running containers.

Access to the containers

- All containers running in the same compose configuration are accessible to each other by name.
- The IP address is dynamically assigned to every new container.
- Additional groups can be defined based on the network definition and configuration in the compose yml file.

Port Forwarding

- The applications running in the container are isolated from host services.
- To make them sometimes directly accessible from host, port mapping with host is defined.
- **When number of instances of same container are created and mapped to the same host port, there is hoist port conflict.**
- The host port once mapped, it cannot be used for other container or other native services.
- The container restricted inside docker can be accessible from external port forwarding.

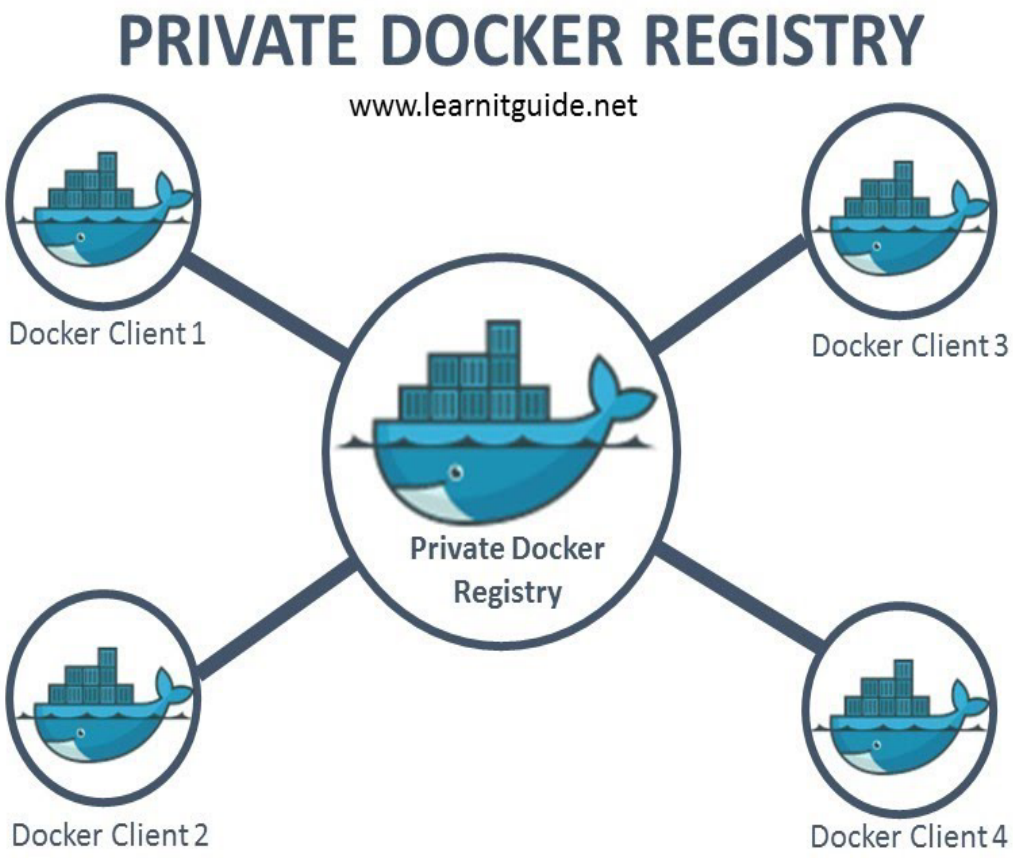
NGINX Proxy

- The **nginx** running in the same compose configuration has access to all the containers defined in the same config-compose yml file.
- The **nginx** application container acts as forwarding agent and load balancer proxy to other containers which are isolated from the host address..
- The nginx acts as proxy load balancer to redirect the requests to the available containers.

Local Docker Registry

- Instead of docker hub registry, can we have a local registry in the network?
  - Reduce the connectivity issues
  - Make our images private to us only.
  - Save time on building and sharing the images.

Docker Private Registry



Docker Registry Image

- The registry is the Docker Registry implementation for storing and distributing Docker images
- Run registry as docker container.
  - docker run -d -p 5000:5000 --restart=always --name registry **registry:2**
- Now pull an image from Docker Hub and push it to your private registry.
  - docker pull ubuntu:16.04.
- Tag the image as localhost:5000/my-Ubuntu.
- When first part of the tag is a hostname and port, Docker interprets this as the location of a registry, when pushing.

Deploy a Registry server

- The ‘registry’, is an instance of the registry image, and runs within Docker.
- Run a local registry to start the registry container:
- $ docker run -d -p 5000:5000 --restart=always --name registry registry:2

Push to our registry

- Tag the new image downloaded
- docker tag ubuntu:16.04 localhost:5000/my-ubuntu.
- Verify the local images
- Push the tagged image to the local registry running at localhost:5000.
- docker push localhost:5000/my-ubuntu.

Clear and pull from Local

- Remove the locally-cached *ubuntu:16.04* and l*ocalhost:5000/my-ubuntu* images
- docker image remove ubuntu:16.04
- docker image remove localhost:5000/my-ubuntu.
- This does not remove the localhost:5000/my-ubuntu image from our own registry.
- Now pull the ‘l*ocalhost:5000/my-ubuntu’ image from local registry server.*
- docker pull localhost:5000/my-ubuntu.
- **This pull works even if you donot have internet connection!**

Stop Local Registry

- To stop the registry, use the same docker container command s.
- docker container stop registry
- docker container rm -v registry

Quick way to check the local images

- docker images -all
- docker images --format "{{.ID}}: {{.Repository}} {{.Tag}}“
- To list all images in the “java” repository,
- docker images java
- To show untagged(dangling) images
- docker images --filter "dangling=true“
- Use in in conjunction with docker rmi ...
- docker rmi $(docker images -f "dangling=true" -q)
- docker images --filter=reference='busy*:*libc'
-

Format the image display

```
docker images --format "table {{.ID}} \t{{.Repository}}
    \t{{.Tag}}"
```

Docker Machine

- **Docker Machine** is a tool for provisioning and managing the Dockerized hosts (hosts with Docker Engine on them).
- Docker Machine is a tool that lets to install Docker Engine on virtual hosts, and manage the hosts with docker-machine commands.
- Docker-Machine to create Docker hosts on the local Mac or Windows box, on the company network, in the data center, or on cloud providers like Azure, AWS, or Digital Ocean.

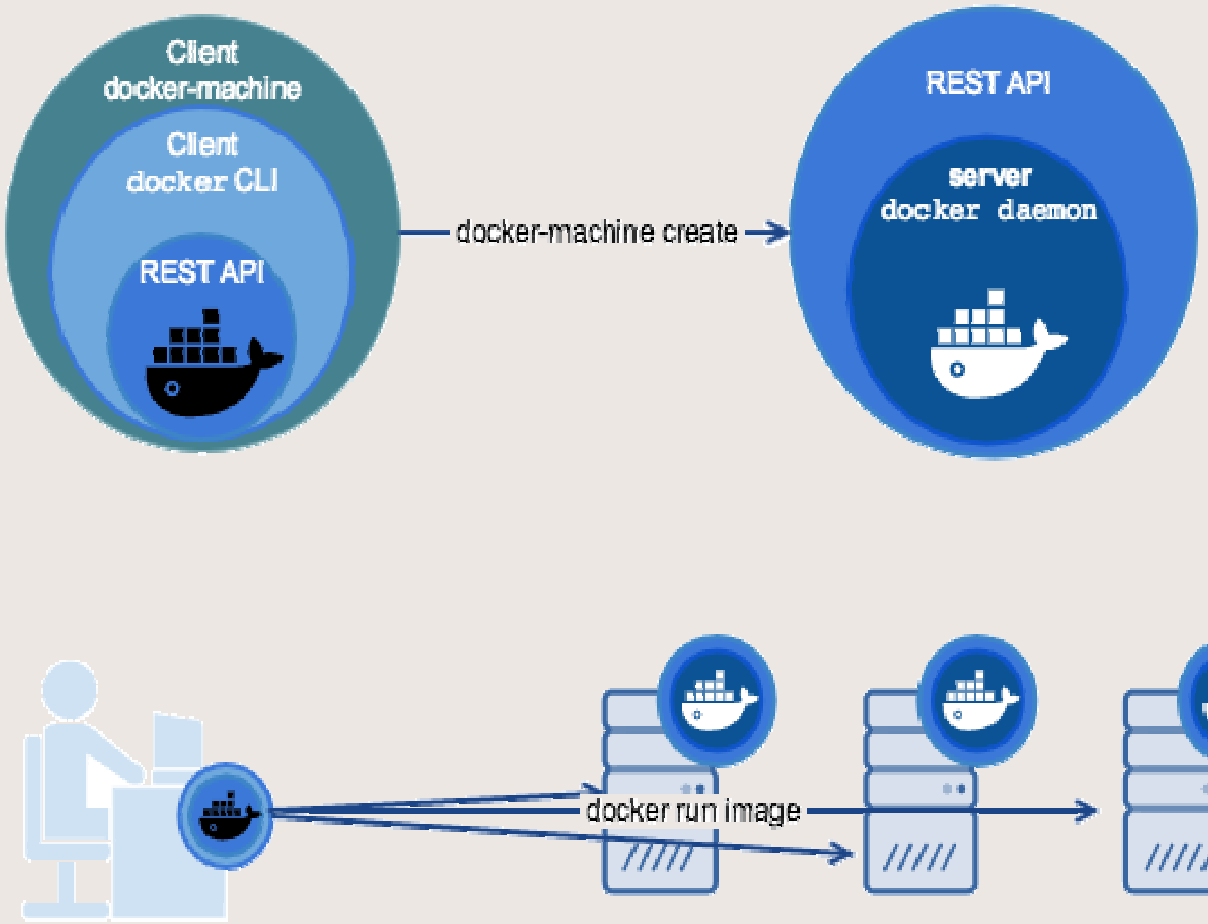
Docker-machine Tool

- The docker-machine is available with Docker toolbox and Docker for Windows installations.
- For installing on Linux,seperately download and configure it.
- Docker-Machine used to install Docker Engine on one or more virtual systems.

Docker-Machine Usage

- Using docker-machine commands, you can start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to the host.
- Install and run Docker on older Mac or Windows with Linux VM
- Provision and manage multiple remote Docker hosts
- Provision Swarm clusters with docker.
- Enables to provision multiple remote Docker hosts on various flavors of Linux machines

Create New Machine



Create Virtual Machines

- On Linux
- docker-machine create --driver virtualbox mc1
- docker-machine create --driver virtualbox mc2
- On Windows 10 with Hypervisor
- docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" vm1
- docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" vm2

Docker Machine Communication Shell

- Run docker-machine env mc1 to get the command shell to configure the shell to talk to docker machine VM instance.
- docker-machine env mc1: returns
- eval $(docker-machine env mc1) on Linux
- & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env mc1| Invoke-Expression : For Windows 10
- Run docker-machine ls to identify the active vm from console shell.
- Run from shell : docker run hello-world

Machine over SSH

- The VM ip is the IP address assigned while creating the VM.
- docker-machine ip mc1.
- docker-machine ssh mc1 "docker image ls“
- docker-machine ssh mc2 "docker ps“
- docker-machine ssh mc1"docker run hello-world“

Docker Container Logs

- The docker logs command batch-retrieves logs present at the time of execution in the container.
- To continuously monitor the logs
- The docker logs --follow command will continue streaming the new output from the container’s STDOUT and STDERR.
- docker logs –f containerName/id
- Docker attach stdout <container>

Service Logs

- When deploying with docker-compose the docker service objects are created, which manage more than one container.
- docker service logs command shows information logged by all containers participating in a service.

Process Docker Logs

- Third party tools like SysDig used to monitor and collect the logs for analyzing.
- The logging driver which sends logs to a file, an external host, a database, or another logging back-end is confiugurable.
- The nginx container process the access and error logs to Stdout and stderror separtely.

Docker networking

- Containers are smallest deployment unit with Docker.
- All the network services assigned to the container are isolated from the host machine, unless configured or mapped.
- Each time a new container is started , a new IP address and new container ID is assigned to the container.
- But the name of the container can be explicitly kept the same.

Linked Containers

- The linked containers are accessible to other containers by name even if the container is restarted or recreated with new IP address.
- Because the container name remains the same.
- The container link can be specified from command line while starting the depending container.
- The containers de
- Fined in the same docker-compose YML file are accessible by their names.

Container by Name

- If the app container is looking for database server host by name as ‘db-server’,then define the linked container as ‘db-server’ in the YML file.
- Services:
- db-server:
- image: mysql:5.6
- ports:
- -2300:3306

Cluster of containers

- All containers in the cluster should be communicating with each other.
- Even if one of the container goes down/crashed,the service should restore another instance.
- Load balancer routes the incoming requests to one of the available containers by applying Round Robin/Weighted containers or other algorithms.
- Docker Swarm mode allows to define and create cluster of containers by combining multiple physical machines as well as virtual hosts created by docker-machine commands.

SWARM Cluster

- Till now, you have been using Docker in a single-host mode on the local machine.
- The Docker also can be switched into **swarm mode**, and that enables the use of swarms.
- Enabling swarm mode instantly makes the current machine a swarm manager.
- From then on, Docker runs the commands you execute on the swarm manager you’re managing, rather than just on the current machine.
- Other nodes joining can be workers or managers.

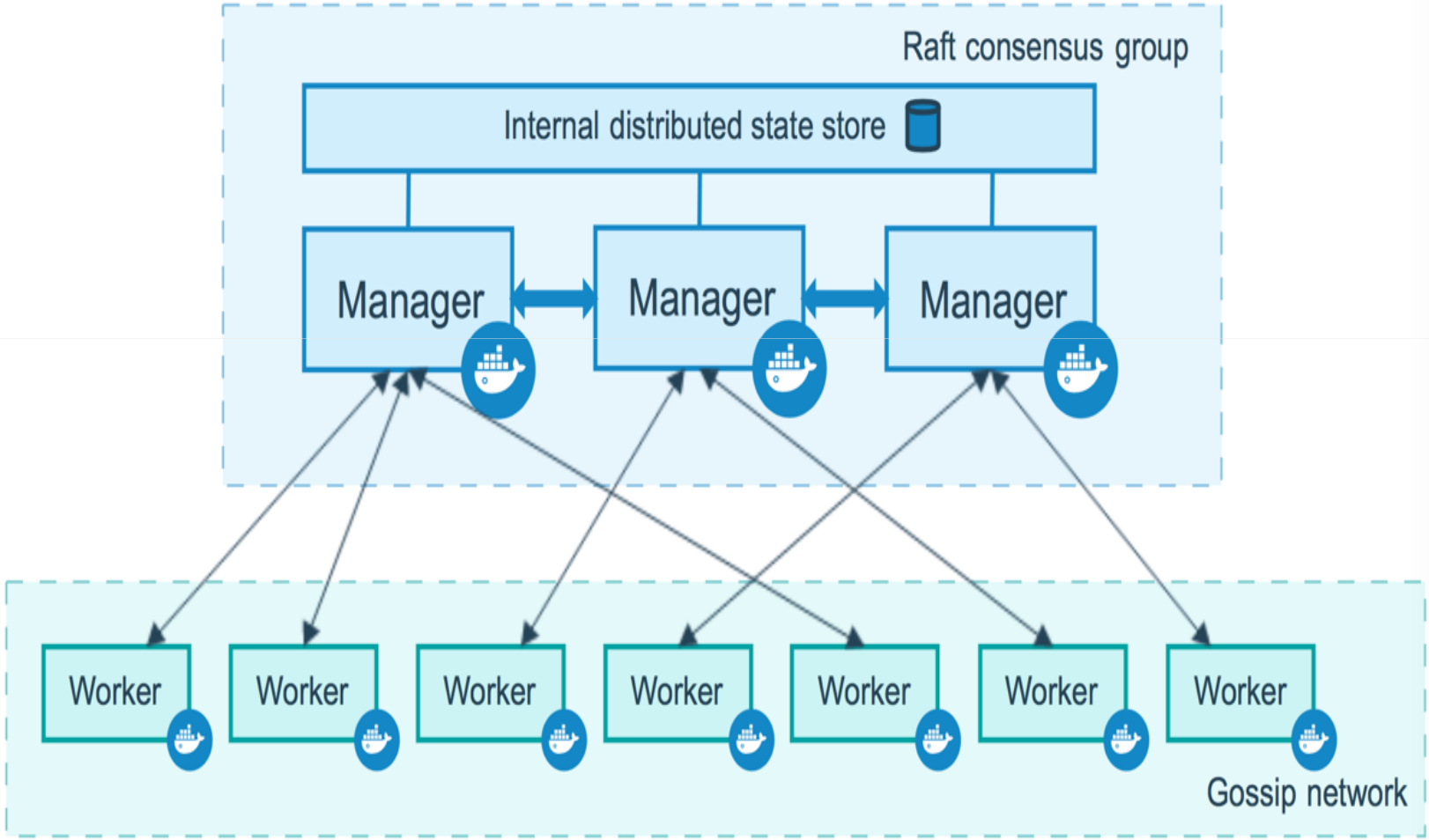
Docker Swarm Cluster

- The swarm is a group of machines that are running Docker and joined into a cluster.
- The commands on the cluster are executed on a cluster by a **swarm manager.**
- The machines in a swarm can be physical or virtual hoists.
- After joining a swarm, they are referred to as **nodes**.
- The nodes can decide to be worker or manager while joining.

SWRM Nodes

- The Swarm managers are the only machines in a swarm that execute the commands or routs the requests to the containers, or authorize other machines to join the swarm as **workers**.
- The Workers are just physical or virtual nodes to provide capacity to run containers and do not have the authority to tell any other machine what it can and cannot do.
- A given Docker host can be a manager, a worker, or perform both roles.

SWARM Nodes


Raft consensus group
Internal distributed state store
Manager
Manager
Manager
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Gossip network

Node Responsibilities

- **Manager nodes**
- Manager nodes handle cluster management tasks:
- maintaining cluster state
- scheduling services
- serving swarm mode HTTP API endpoints
- **Worker nodes**
- Worker nodes are also instances of Docker Engine whose sole purpose is to execute containers.
- Worker nodes don’t participate in the manager tasks

Services in Swarm

- The service is an object created to deploy an application image when Docker Engine is in swarm mode.
- For service creation, specify which container image to use and which commands to execute inside running containers.

Swarm Service options

- The options for the service deployments:
- The port where the swarm makes the service available outside the swarm
- An overlay network for the service to connect to other services in the swarm
- CPU and memory limits and reservations
- A rolling update policy
- The number of replicas of the image to run in the swarm

Swarm algorithm

- Swarm managers use several strategies to assign the incoming requests to the containers, such as “emptiest node” -- which fills the least utilized machines with containers.
- Or “global”, which ensures that each machine gets exactly one instance of the specified container.
- The swarm manager is configured to use the strategies defined in the Compose file.

Swarm Load Balancing

- The swarm manager uses **ingress load balancing** to expose the services you want to make available externally to the swarm.
- The swarm manager can automatically assign the service a **PublishedPort** or you can configure a PublishedPort for the service.
- You can specify any unused port. If you do not specify a port, the swarm manager assigns the service a port in the 30000-32767 range.

Swarm DNS

- Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry.
- The swarm manager uses **internal load balancing** to distribute requests among services within the cluster based upon the DNS name of the service.
- Service name as app_web is created for container named web with stack name app.
- The service app_web tracks the number of replicas for web conatiner.

Docker Machines in swarm

- Run docker-machine env mc1 to get the command shell to configure the shell to talk to docker machine VM instance.
- docker-machine env mc1: returns
- eval $(docker-machine env mc1) on Linux
- & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env mc1| Invoke-Expression : For Windows 10
- Run docker-machine ls to identify the active vm from console shell.
- Run from shell : docker swarm init..

Machine over SSH

- docker-machine ssh mc1 "docker swarm init“
- docker-machine ssh mc2 "docker swarm join“
- docker-machine ssh mc1"docker swarm init --advertise-addr <vm1 ip>“
- Where the vm1 ip is the ip assigned while creating the VM.
- docker-machine ip mc1.

Swarm Init

- A swarm is made up of multiple nodes, which can be either physical or virtual machines.
- To init the swarm mode: run docker swarm init to enable swarm mode and make the current machine a swarm manager.
- Then run docker swarm join on other machines to have them join the swarm as workers or other managers.
- Run docker swarm init and docker swarm join with port 2377 (the swarm management port), or no port at all and let it take the default.

Swarm Stack Deploy

- Once the swarm mode is initialized, the application conpose yml is deployed on swarm mamager.
- docker stack deploy -c docker-compose.yml app
- The stack is initialized with services defined for every container in the compose file.
- docker stack ls
- docker stack ps app
- The services manage the no of replicas of the images.
- Read the services with : docker service ls

Swarm deployment constraints

- Initial number of instances(replicas)
- Limit the use of cpu and memory
- Define restart policy with condition as on failure
- Update config in parallel
- Delay between restart attempts
- Maximum number of restarts before giving up
- Delay between updates
- ction on update failure
- Maximum number of tasks updated simultaneously
- The placement constraints for specific node

Containers in swarm

- You can connect an existing container to one or more networks.
- A container can connect to networks which use different network drivers.
- Once all the containers connected, the containers can communicate using another container’s IP address or name.

Cluster State

- Docker works to maintain the desired state of cluster.
- If a worker node becomes un-available,Docker schedules that node’s tasks on other nodes.
- A *task* is a running container which is part of a swarm service and managed by a swarm manager.
- If one of the node/container goes down/crashed, to maintain the specified number of replicas in the cluster, docker creates more number of containers and adds them in the cluster.

Swarm Service command

- The service commands usage
  - docker service ps app_web
  - docker service inspect app_redis
  - docker service logs app_web
  - To scale the service container instances
  - docker service scale app_web=5
  - docker service scale app_redis=2
  - To remove the stack
  - docker stack rm app

Scaling the containers

- Once the swarm is initialized and stack is deployed with initial no of container instances specified as replicas in compose YML definition.
- The docker service is used to scale to new number of replicas depending on the load of request processing.
- The number of replicas can be scaled up or down
- The command docker service scale SERVICE=REPLICAS manages the no of container instances.

Swarm service update

- Once the application is deployed in the container in production environment and now to update with new change in application .
- Update the application image
- Re-deploy the stack by removing it or restarting it.
- With Docker swarm services, you modify a service's configuration, including the networks and volumes it is connected to, without the need to manually restart the service.
- Docker will update the configuration, stop the service tasks with the out of date configuration, and create new ones matching the desired configuration.

Rolling update with image

- Deploy the stack with image and configuartion
- Now you want to update the container with update the image or other configuration.
- The rolling update allows update of the container with new image in steps.
- docker service update --image redis:3.0 redis-service

Update parameters

- The updates are done for
  - Newer image version tags
  - Config file add/update
  - Remove a config file
  - Add/update placement constatrint
  - Add/upodate container lebel
  - Add/update environment varaibales
  - Add a new generic resource
  - Update the container host name
  - Update CPU usage/memeory limits
  - Add a network

Update phases

- Once the image/external update is done, update the service
- The scheduler arranges the update tasks
  - Stop the first task.
  - Schedule update for the stopped task.
  - Start the container for the updated task.
  - If the update to a task returns RUNNING, wait for the specified delay period then start the next task.
  - If, at any time during the update, a task returns FAILED, pause the update.

Service Rollback

- Used to revert changes to a service's configuration.
- Roll back a specified service to its previous version from the swarm
  - docker service rollback -d SERVICE-Name
- This command must be run targeting a manager node.
- After executing this command, the service is reverted to the configuration that was in place before the most recent docker service update command.

Containers in Linux

- The docker containers utilize the native feature of Linux systems in kernel as **chroot jail**, that protects the containers from outside world.
- For access outside the container environment, the host address and port mapping allows to access it.
- This feature is provided internally by NAT network drivers in the Linux OS.

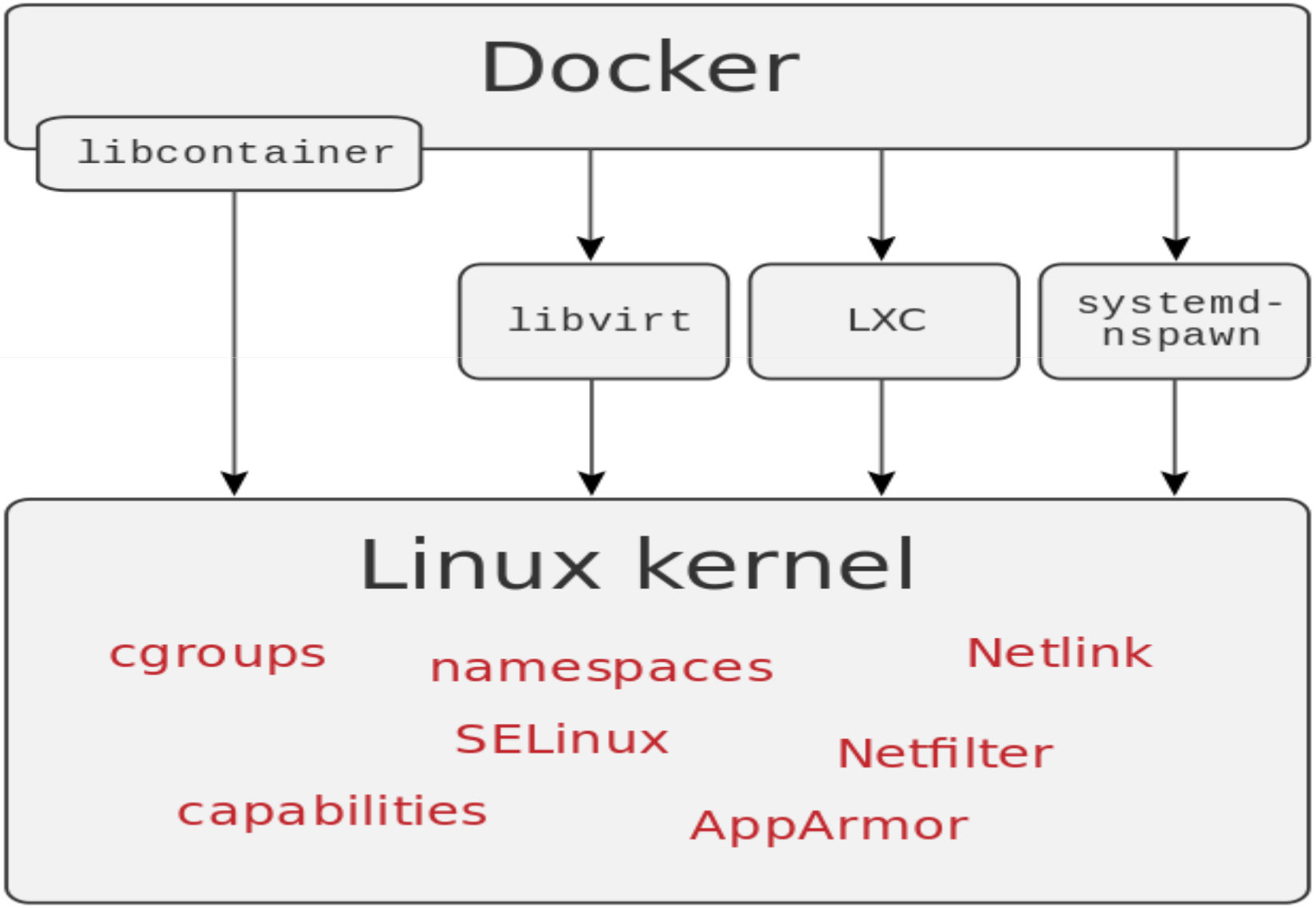
Unix Jail

- All Unix and Linux systems have change root jails" or "*chroot jails,*" feature that puts an barrier between the "jailed" software and the rest of the system.
- Because this jail is enforced by the operating system and not by an application, it provides an high level of safety and security to the application from the rest of the system and outside the system.
- A chroot jail "incarcerates" un-trusted applications, and acts like a guard, for applications that already have substantial security measures built-in.

Docker Container Support

- Image management
- Resource Isolation
- File System Isolation
- Network Isolation
- Change Management
- Sharing
- Process Management
- Service Discovery with DNS

Docker Internals



Namesapce and CGroups

- A namespace wraps a global system resource into an abstraction which will be bound only to processes within the namespace and thus providing resource isolation.
- The cgroups are a metering and limiting mechanism, they control how much of a system resource (CPU, memory) you can use.
- The namespaces limit what you can see.

Docker on Linux

- The Docker is developed in Google's Go language as native application on Linux system that takes advantage of several features of the Linux kernel.
- It as an application wrapped around features of Linux kernel that already exists in the kernel
- The cgroups to limit an applications available resources
- The namespaces to provide isolation from other containers
- The Union Filesystems to provide fast, light access to storage
-

Kernel Namespaces

- **Kernel Namespaces from Linux**
- Docker uses kernel namespaces to provide the isolated workspace called the *container*.
- When you run a container, Docker creates a set of *namespaces* for that container.
- These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Containers with Kernel

- Containers share the host kernel
- Containers use the kernel ability to group processes for resource control
- Containers ensure isolation through namespaces
- Containers feel like lightweight VMs (lower footprint, faster), but are **not Virtual Machines!**

Linux Namespaces

- Docker Engine uses following namespaces on Linux:
- **PID namespace** for process isolation.
- **NET namespace** for managing network interfaces.
- **IPC namespace** for managing access to IPC resources.
- **MNT namespace** for managing filesystem mount points.
- **UTS namespace** for isolating kernel and version identifiers.

Container NameSpaces

- Process Tree : HostName,Container name
- Volume Mounts : Source, mopunt path as destination
- Network : Network configiratioon with ip address, port no,port mapping etc.
- User Accounts : User and Group IDS
- Inter Process Communication
- To view these values for running container in json format
- Use ‘docker inspect <container name/id>’
- The inspect is also available for service objects.

Kernel Control Groups

- Kernel control groups (cgroups) allow you to do accounting on resources used by processes.
- It does a little bit of access control on device nodes and other things such as freezing groups of processes.

CGroups

- The cgroups, which are sometimes referred to as containers is used to slice an entire operating system into buckets
- It is similar to how virtual machines slice up their host system into buckets, but without having to go so far as replicating an entire set of hardware.
- The four of the system resources that cgroups can control – CPU, memory, network, and storage I/O – could be cut into slices that are then combined into two groups.

Cgroups in Linux

- A cgroup limits an application to a specific set of resources.
- Docker uses kernel control groups for resource allocation and isolation.
- The control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints.

Docker Cgroups

- **Memory cgroup** for managing accounting, limits and notifications.
- **HugeTBL cgroup** for accounting usage of huge pages by process group.
- **CPU group** for managing user / system CPU time and usage.
- **CPUSet cgroup** for binding a group to specific CPU.
- **net_cls** and **net_prio cgroup** for tagging the traffic control.
- **Devices cgroup** for reading / writing access devices.

Docker File System

- Union file systems operate by creating layers, making them very lightweight and fast.
- Docker Engine uses UnionFS to support the building blocks for container.

Container Format

- Docker Engine combines the namespaces, control groups and UnionFS into a wrapper called a container format.
- The default container format is libcontainer.

Kernel security in Docker

- Docker Engine makes use of AppArmor, Seccomp, Capabilities kernel features for security purposes.
- **AppArmor** allows to restrict programs capabilities with per-program profiles.
- **Seccomp** used for filtering syscalls issued by a program.
- **Capabilties** for performing permission checks.

Net namespace

- Network namespaces provided by the Linux kernel, are a lightweight mechanism for resource isolation
- The processes attached to a network namespace see their own network stack, while not interfering with the rest of the system’s network stack.
- The network namespace contains its own network resources: interfaces, routing tables, etc.
- Network namespaces allow only one interface to be assigned to a namespace at a time.
- If the root namespace owns *eth0*, which provides access to the external world, only programs within the root namespace could reach the Internet.

NAT Driver

- The **Network address translation** (**NAT**) is a way of re-mapping one IP address space into another by modifying network address information in the IP header of packets while they are in transit across a traffic routing device.
- **IP masquerading** is a technique that hides an entire IP address space, usually consisting of private IP addresses, behind a single IP address in another, usually public address space.
- The Docker manipulates iptables rules on Linux or it manipulates routing rules on Windows containers

Docker Network driver

- **Bridge**: Default network driver The Bridge networks are usually used when your applications run in standalone containers that need to communicate.
- **Overlay**: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other.
- The overlay networks also facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.
- This strategy removes the need to do OS-level routing between these containers.

NW Drivers for Docker

- **None**: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services.
- **Macvlan**: Macvlan networks allow to assign a MAC address to a container, making it appear as a physical device on the network.
- The Docker daemon routes traffic to containers by their MAC addresses.
- Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host’s network stack.

Container in Host Mode

- **HOST mode:**
- The container is just a process running in a host, which connects to the connect it to the “host NIC” (or “host networking namespace”).
- The container will behave from a networking standpoint just as any other process running in the host.
- Containers share the host network namespace, which may have security implications.
- Run a container in host mode on a docker host
- docker run -p 8080:80/tcp -p 8080:80/udp apache

Dynamic Port mapping

- docker run --name app-container -p 80 nginx
- This assigns dynamic port to nginx server in the container.
- Docker inspect app-container.
- To read the dynamic port mapped to the host
- docker inspect --format='{{(index (index .NetworkSettings.Ports "80/tcp") 0).HostPort}}' app-container : rreturns the value as 32769
- The container is reachable on http://192.168.99.100:32769/

Dynamic Port Issues

- With dynamic ports for the containers the external services will not be able to identify and connect.
- “Dynamic port assignment” needs to be managed by a container orchestration platform and requires specific code in the container to learn the assigned port.

NAT Usage

- In order to avoid the “port clash” in “Host” mode, a solution would be putting the container on a completely separate network namespace, internal to the host where it’s living, and then “sharing” the “external” IP address of the host amongst the many containers living in it through the use of Network Address Translation (NAT)

NAT mapping

- NAT is as your home network connecting to the broadband provider, where the public IP address of the broadband router is shared between the devices in your home network when they reach the internet.
- Your laptop and cell phone get a private address in your home network, and those get “transformed” (NAT’ed) to the public IP address that your provider assigns you as they traverse the broadband router.

Bridge Mode Host

- A separate virtual bridge can be created with a completely separate internal network namespace.
- In this mode, containers are connected to the internal “private network”, and each one gets its own IP address and a full network namespace where all TCP ports are available.
- This translating between “host” address and “internal container ” addresse is performed inside the host by iptables, a well-known linux program that enables to configure network translation rules in the kernel so that an “external” and port combination is “published” and translated to a specific “internal addrress and port ” combination.

Network mode

- docker run -d --name app-container --network bridge nginx
- docker run -d --name app-container --network host nginx
- docker run -d --name app-container --network none nginx
- docker run -d --name app-container --network host nginx
- For user defined networks
- docker run -d --name app-container --ip 192.168.12.100 --network bridge nginx

Network options

- bridge': create a network stack on the default Docker bridge
- 'none': no networking
- 'container:<name|id>': reuse another container's network stack
- 'host': use the Docker host network stack

Docker Networks

- The docker system creates three networks as default. To view it
- docker network ls
  - bride,host,none having local scope
- Once the service is initialized
  - Custom bridge network is added with local scope.
- Once the swarm is initialized
  - Overlay network named ingress with swarm scope is added into networks

Docker Security

- Docker containers are, by default, quite secure; especially if you run your processes as non-privileged users inside the container.
- You can add an extra layer of safety by enabling AppArmor, SELinux, GRSEC, or another appropriate security hardening system.
- Intrinsic security of the kernel.
- Attack surface of the Docker daemon
- Loopholes in the container configuration profile, either by default, or when customized by users.
- The security features of the kernel and how they interact with containers.
- Use trusted images

Docker Secrets

- Manage sensitive data with Docker secrets.
- The *secret* is a blob of data, such as a password, SSH private key, SSL certificate, or another piece of data that should not be transmitted over a network or stored unencrypted in a Dockerfile or in your application's source code.
- The secrets are encrypted during transit and at rest in a Docker swarm.
- A given secret is only accessible to those services which have been granted explicit access to it, and only while those service tasks are running.

Secrets Usage

- Use secrets to manage any sensitive data which a container needs at runtime but you don’t want to store in the image or in source control, such as:
  - Usernames and passwords
  - TLS certificates and keys
  - SSH keys
  - Other important data such as the name of a database or internal server
  - Generic strings or binary content (up to 500 kb in size)

Secrets Access

- The docker secrets are only available to swarm services, not to standalone containers.
- To use this feature, consider adapting the container to run as a service.
- Stateful containers can typically run with without changing the container code.
- Use secrets to manage non-sensitive data, such as configuration files.
- The secrets provide a layer of abstraction between the container and a set of credentials, where you have separate development, test, and production environments for the application.

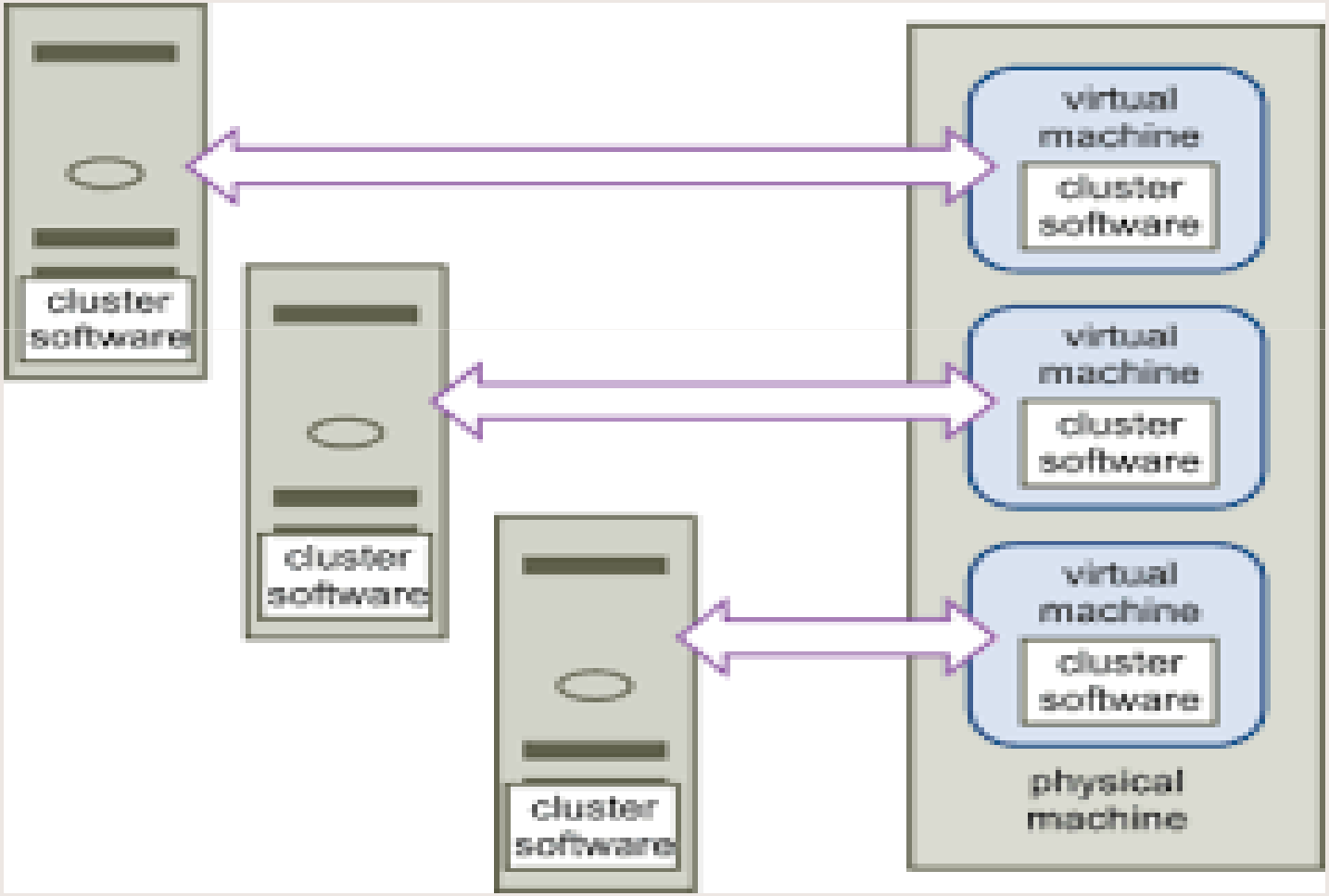
Container Management

- The Docker platform and other tools support to manage the lifecycle of a container.
- The Docker Command Line Interface (CLI) supports the following container activities:
  - Pulling a repository from the registry.
  - Running the container and optionally attaching a terminal to it.
  - Committing the container to a new image.
  - Uploading the image to the registry.
  - Terminating a running container.

Large No of Containers

- CLI meets the needs of managing one container on one host,.
- It is not useful for managing multiple containers deployed on multiple hosts.
- To go beyond the management of individual containers, we turn to container orchestration tools.
- The container orchestration tools extend lifecycle management capabilities to complex, multi-container workloads deployed on a cluster of machines.
- The orchestration tools allow users to treat the entire cluster as a single deployment target.

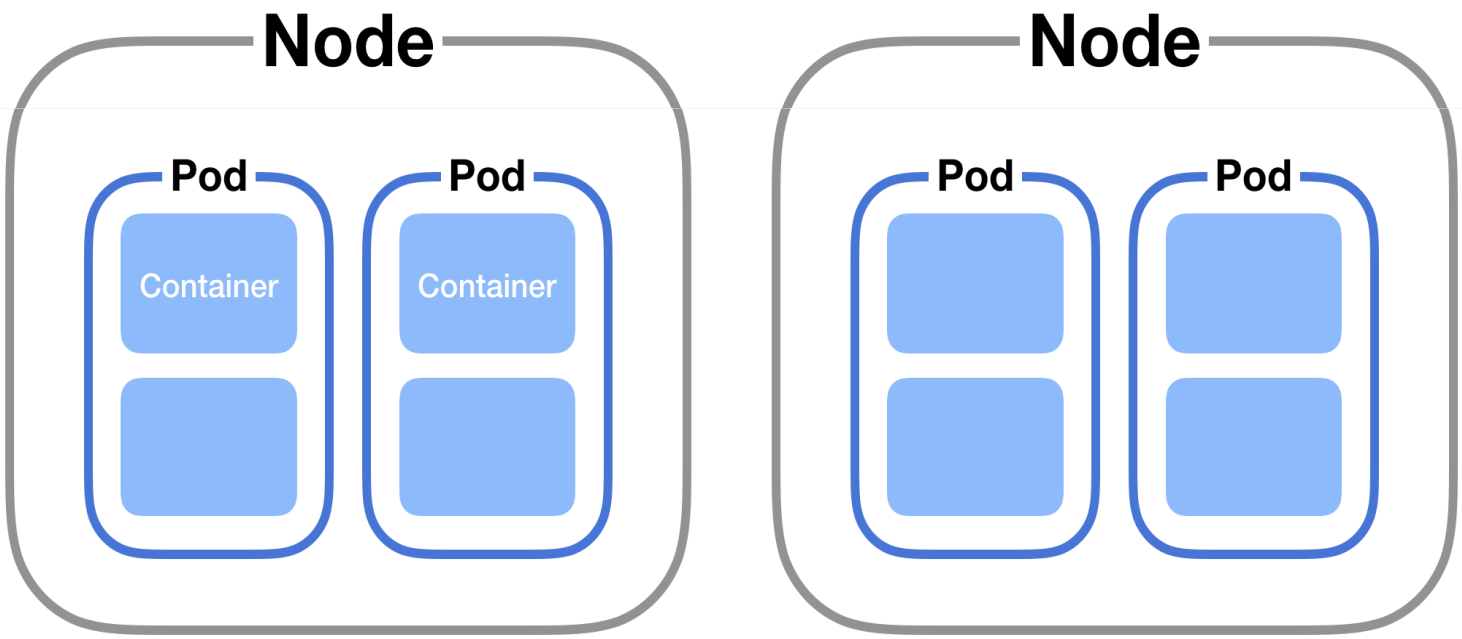
Cluster of Machines



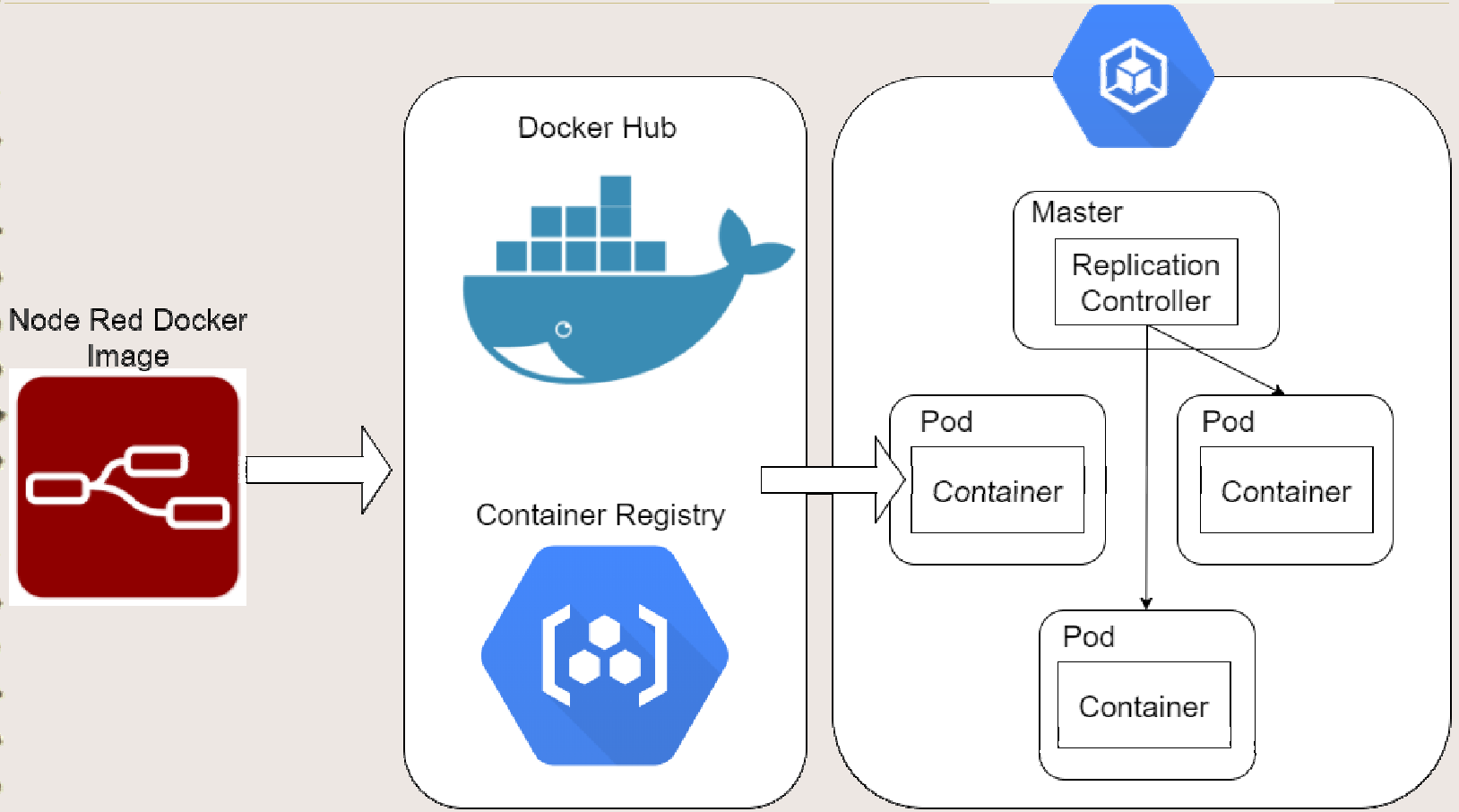
Orchestration process

- The process of orchestration involves tools that can automate all aspects of application management from initial placement, scheduling and deployment to steady-state activities such as update, deployment, update and health monitoring functions that support scaling and failover.
- These capabilities characterize some of the core features users expectations offer modern container orchestration tools.
- These tools use container configuration in a standard schema, using languages such as YAML or JSON.

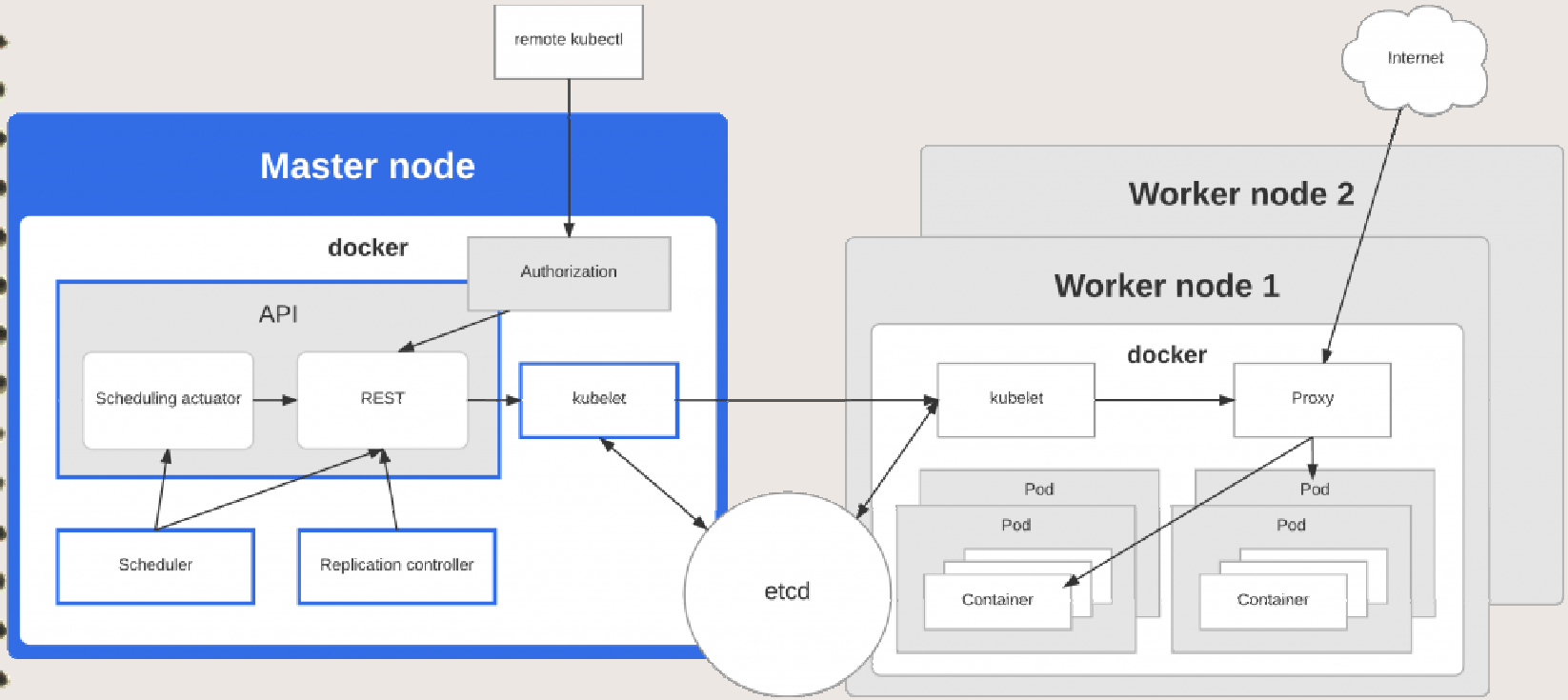
Cluster of Containers



Master Node


Node Red Docker Image
Docker Hub
Container Registry
Master
Replication Controller
Pod
Container
Pod
Container
Pod
Container

Worker and Master



Configuration options

- Declarative definition in YML
- **Rules and Constraints** for placements of containers, performance and high availability.
- Provisioning, or scheduling involves negotiating the placement of containers within the cluster and launching them. This involves selecting an appropriate host based on the configuration and load balancing algorithm.

Container Discovery

- In a distributed deployment consisting of containers running on multiple hosts, container discovery becomes critical. Web servers need to dynamically discover the database servers, and load balancers need to discover and register web servers.
- The Orchestration tools provide, or expect, a distributed key-value store, a lightweight DNS or some other mechanism to enable the discovery of containers.

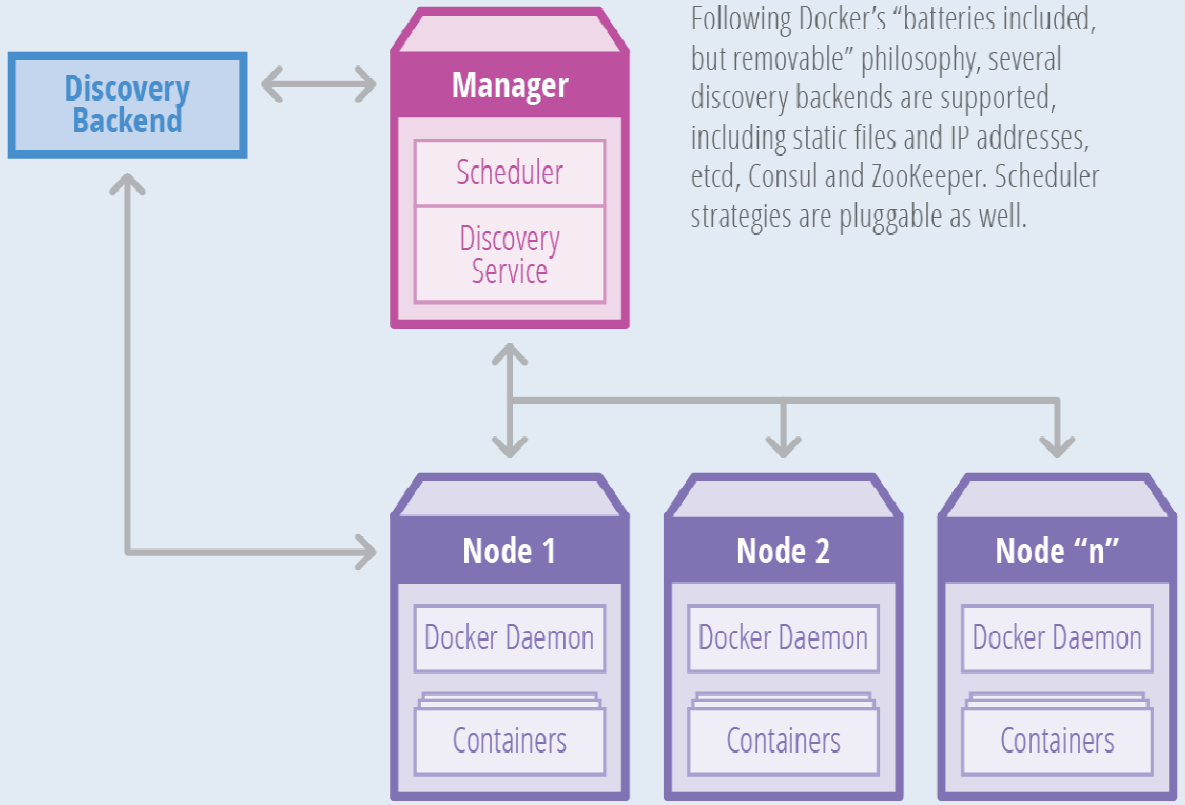
Health Monitoring

- **Health Monitoring and corrective actions**
- Since orchestration tools are aware of the desired configuration of the cluster, they should be able to track and monitor the health of the clusters's containers and hosts.
- In the event of host failure, the tools can relocate the container.
- Similarly, when a container crashes, orchestration tools can launch a replacement.
- Orchestration tools ensure that the deployment always matches the desired state declared by the developer or operator.

Container Orchestration Tools

- **Docker Swarm**
- Matches with Docker's own YML configurations or command line approaches.
- Swarm transparently deals with an endpoint associated with a pool of Docker Engines.
- The existing tools and APIs continue to work with a cluster in the same way they work with a single instance.
- Docker's tooling/CLI and Compose are how developers create their applications, and therefore, they don't have to be recoded to accommodate an orchestrator.

Swarm Architecture



Source: The New Stack.

THENEWSTACK

Swarm Usage

- Docker Swarm supports constraints and affinities to determine the placement of containers on specific hosts.
- Constraints define requirements to select a subset of nodes that should be considered for scheduling.
- They can be based on attributes like storage type, geographic location, environment and kernel version. Affinity defines requirements to collocate containers on hosts.

Discovery in Swarm

- For discovering containers on each host, Swarm uses a pluggable backend architecture that works with a simple hosted discovery service, static files, lists of IPs, etcd, Consul and ZooKeeper.
- Swarm supports basic health monitoring, which prevents provisioning containers on faulty hosts.

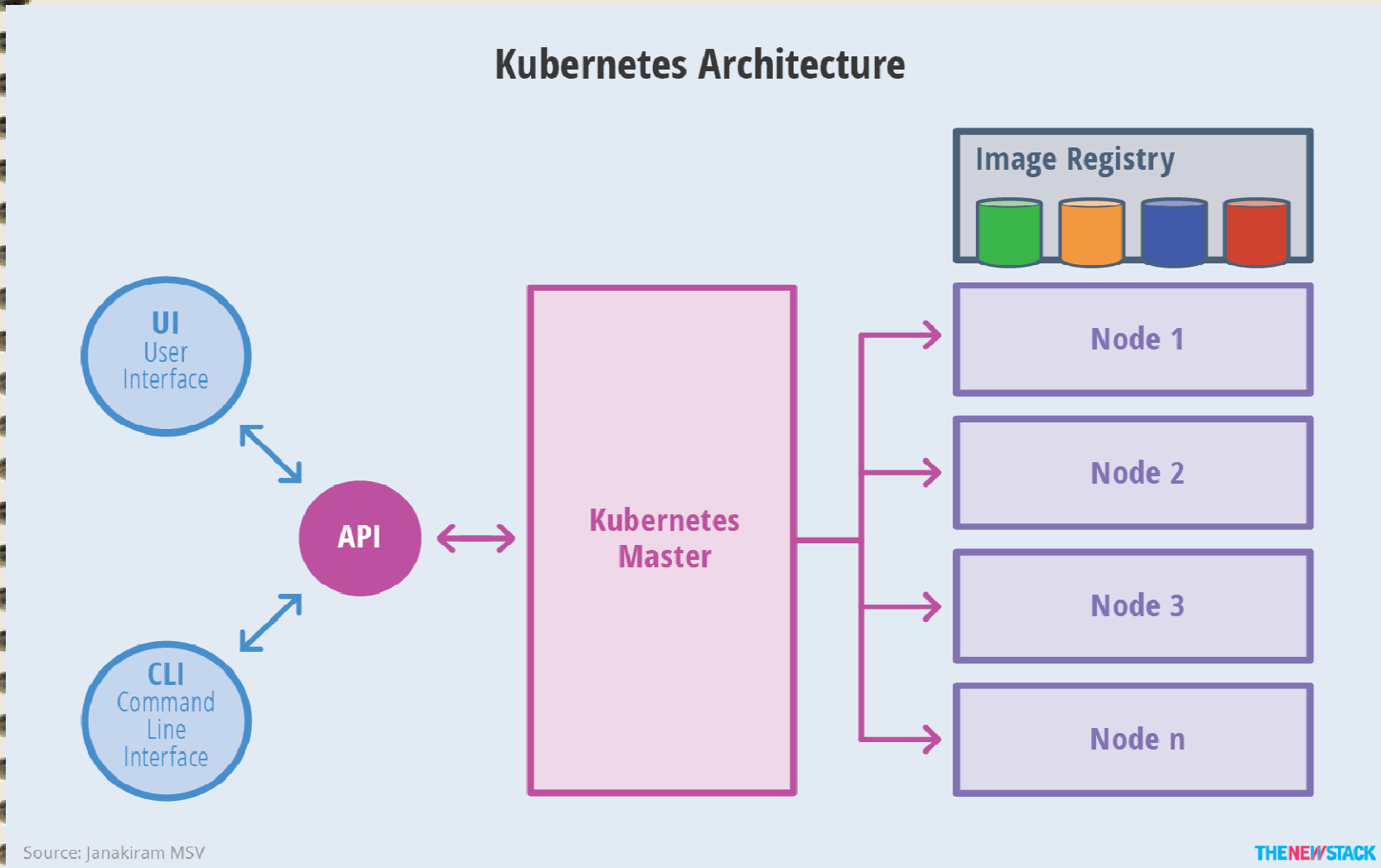
Apache Mesos

- Apache Mesos is an open source cluster manager that simplifies the complexity of running tasks on a shared pool of servers.
- Originally designed to support high-performance computing workloads, Mesos added support for Docker in the 0.20.0 release.
- A typical Mesos cluster consists of one or more servers running the mesos-master and a cluster of servers running the mesos-slave component.

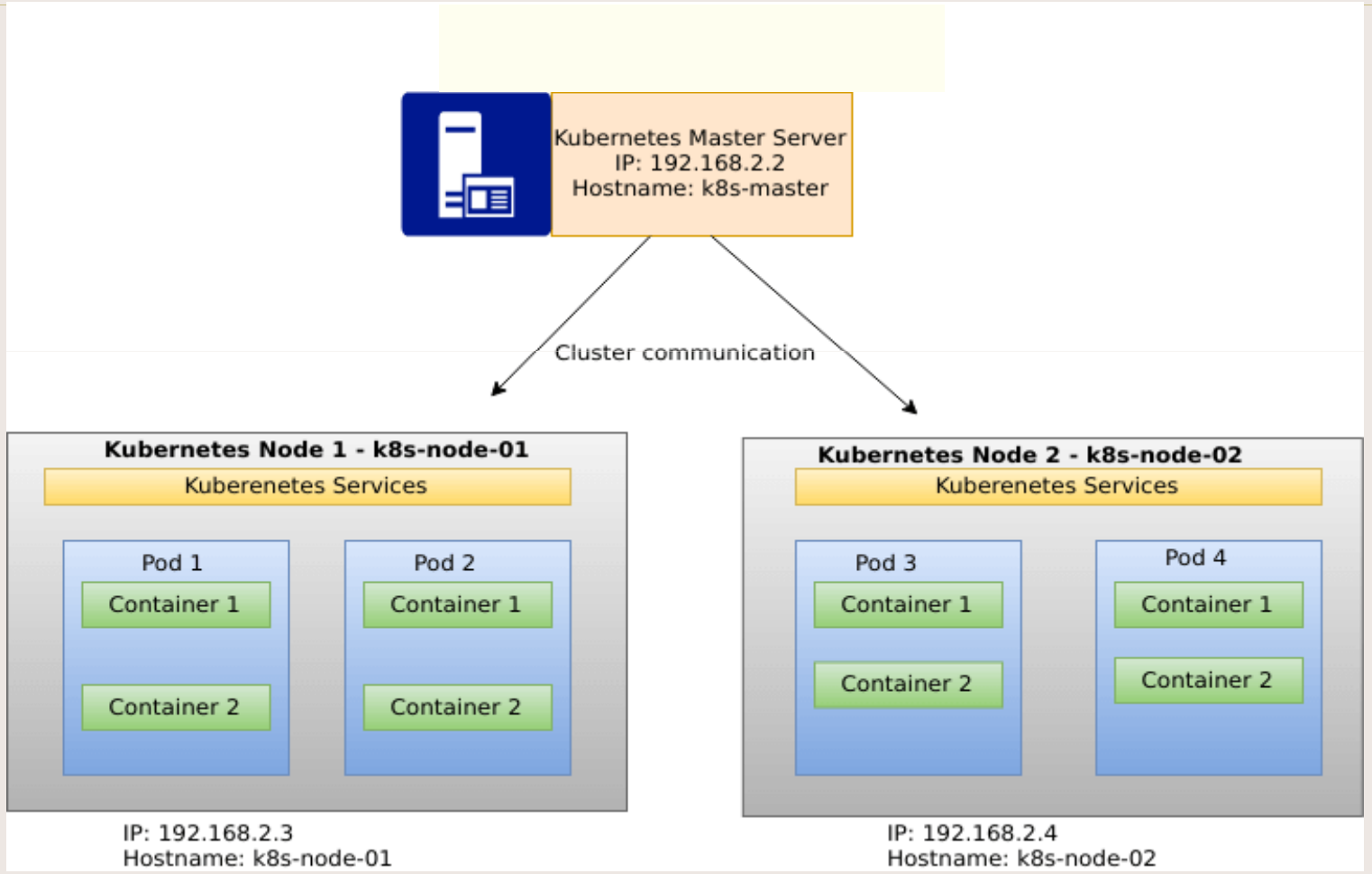
Kubernetes

- Container orchestration tool from Google that claims to deal with two billion containers every day.
- Kubernetes enjoys unique credibility.
- Kubernetes works with different container such as Docker and RKT containers.
- Kubernetes controls the containers through Virtual Machine Drivers.
- VM is needed to work with Kubernetes cluster.
-

Kubernetes architecture



Nodes in Cluster



Kubernetes Deployments

- Local Pvt network cluster
- Minikube : Single node cluster managed with virtual driver
- Docker in Docker cluster(experimentation)
- Kubernetes in vms shared from Cloud.
- Kubernetes shared as service from cloud.
  - AKS
  - GKS
  - AKS

Kubernetes Tools

- Virtual Machine Manager : Oracle Virtual Box
- Minikube : Local Cluster Manager
- Kubectl : Cluster Client
- Tools available for platforms like windows, Linux with flavors ,Mac and multiple other platforms.
- Works with Container Runtimes
  - Docker
  - RKT
  - CRI-O

Kubernetes in nutshell

- Kubernetes' architecture is based on a master server with multiple nodes which are pod managers.
- The command line tool, called kubecfg, connects to the API endpoint of the master to manage and orchestrate the pods through nodes.
- Below is the definition of each component that runs within the Kubernetes environment:

Kubernetes design

- Kubernetes is designed on the principles of scalability, availability, security and portability.
- It optimizes the cost of infrastructure by efficiently distributing the workload across available resources.

Kubernetes Node

- A node is a worker machine in Kubernetes, previously known as a minion.
- A node may be a VM or physical machine, depending on the cluster.
- Each node has the services necessary to run **pods** and is managed by the master components.
- The services on a node include Docker, kubelet and kube-proxy.

Kubernetes components

- **Master**: The server that runs the Kubernetes management processes, including the API service, replication controller and scheduler.
- **Node**: The host that runs the kubelet service and the Docker Engine. Nodes receive commands from the master.
- **Kubelet**: The node-level manager in Kubernetes; it runs on a minion.
- **Pod**: The collection of containers deployed on the same Node.

Kubernetes components

- **Replication controller**: Defines the number of pods or containers that need to be running.
- **Service**: A definition that allows the discovery of services/ports published by each container, along with the external proxy used for communications.
- **Kubecfg**: The command line interface that talks to the master to manage a Kubernetes deployment.

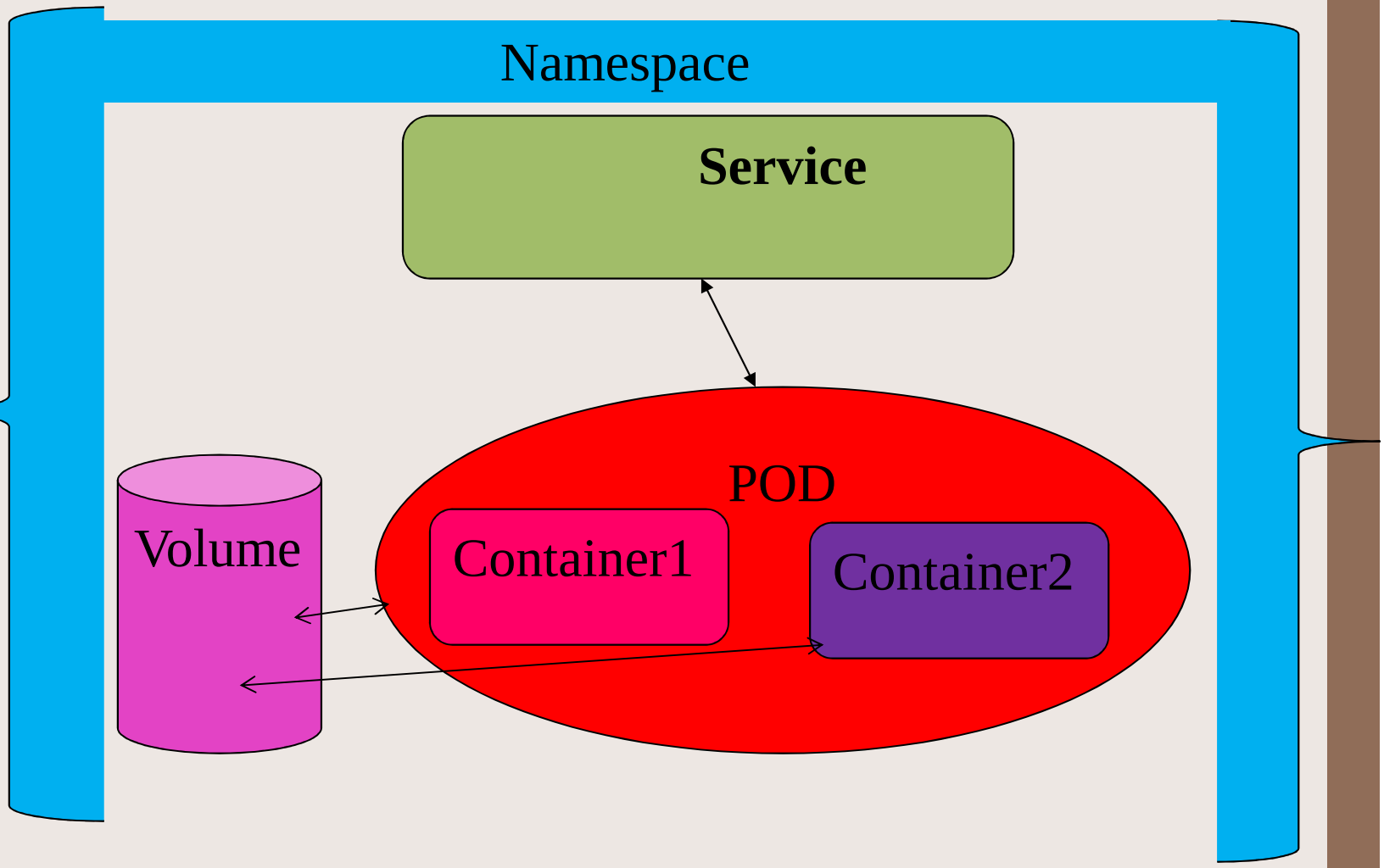
Pods in Kubernetes

- A pod is a collection of one or more containers. The pod serves as Kubernetes' core unit of management.
- Pods act as the logical boundary for containers sharing the same context and resources.
- The grouping mechanism of pods make up for the differences between containerization and virtualization by making it possible to run multiple dependent processes together.
- At runtime, pods can be scaled by creating replica sets, which ensure that the deployment always runs the desired number of pods.

Basic Objects

- **Pod** : Group of containers
- **Deployment** : Group of pods
- **Service** : Controls the request dispatching with pods
- **Volume** Shares data with containers
- **Namespace** : Package for all above objects

Object Structure



Kubernetes configuration

- The service definition, along with the rules and constraints, is described in a JSON file.
- For service discovery, Kubernetes provides a stable IP address and DNS name that corresponds to a dynamic set of pods.
- When a container running in a Kubernetes pod connects to this address, the connection is forwarded by a local agent (called the kube-proxy) running on the source machine to one of the corresponding backend containers.

Get Started with Cluster

- On Linux : Native Kubernetes tools with Docker,kubedam and kubectl
- On Windows10 with Hypervisor : Docker Desktop for Windows, preconfigured cluster and kubectl.
- On Windows 7 : minikube with VM and kubectl

Kubectl

- kubectl is a command line interface for running commands against Kubernetes clusters.
- Kubectl command structure
- Kubectl command, TYPE, NAME, and flags are:
- Commands are
  - Create
  - Get
  - Describe
  - Delete

Kubectl Type

- Specifies the resource type. Resource types are case-insensitive and you can specify the singular, plural, or abbreviated forms.
- For example, the following commands produce the same output:
- kubectl get pod pod1
- kubectl get pods pod1
- kubectl get po pod1

Kubectl resource Name

- NAME: Specifies the name of the resource.
- Names are **case-sensitiv**e.
- If the name is omitted, details for all resources are displayed,
- kubectl get pods.
- **kubectl get pod pod1 pod2**
- **Multiple resource types**
- **kubectl get pod/-pod1 replicationcontroller/rc1**
- To specify resources with one or more files: -f file1 -f file2 -f file<#>
- **kubectl get pod -f ./pod.yaml**

Kubectl Command Flags

- flags: Specifies optional flags.
- Use the -s or --server flags to specify the address and port of the Kubernetes API server.
- Flags that you specify from the command line override default values and any corresponding environment variables.
- To get the help, just run ’kubectl help’ from the terminal window.
- To print information about the status of a pod
- kubectl get pods <pod-name> --server-print=false

Kubectl commands

- To start the deployment
  - kubectl run app --image=pbadhe34/my-apps:app1 --port=8090
- To view the pods
  - kubectl get pods
- To view the deployments
  - kubectl get deployments
- To view the services
  - kubectl get services

Kubectl commands

- Expose the service for the deployment
- kubectl expose deployment app --type=NodePort
- Delete the service and deployments
- kubectl delete service, deployment app

Kubectl Configuration

- The definition of Kubernetes objects, such as pods, replica sets and services, are submitted to the master.
- Based on the defined requirements and availability of resources, the master schedules the pod on a specific node.
- The node pulls the images from the container image registry and coordinates with the local container runtime to launch the container.

POD in Host Network Mode

- The containers support the network communication in two modes
- Host Mode: The container address space utilizes the host address space.
- Multiple container can use the same host mode where their ip and port are assigned to host address and port.
- The same pod name cannot be reassigned.
- The pod name can be generated by the tag generateName.

Pod in Host Port mode

- The pod’s port and address is explicitly mapped to the host address and port no.
- Not more than one pod is able to get assigned the same port mapping.
- The pod name can be get generated.
- But still the port mapping once done from host side, cannot be reassigned again the same.

Exec Command

- Execute a command against a container in a pod.
- kubectl exec <pod-name> date
- Get output from running 'date' in container <container-name> of pod <pod-name>.
- kubectl exec <pod-name> -c <container-name> date
- Get an interactive TTY and run /bin/bash from pod <pod-name>.
- kubectl exec -ti <pod-name> /bin/bash : For singkle container pod. Which is default.

Logs command

- Print the logs for a container in a pod.
- kubectl logs <pod-name>: Single pod logs
- kubectl logs <pod-name> <container name>
- Start streaming the logs from pod <pod-name>
- kubectl logs -f <pod-name>

Linking the Containers in POD

- All the containers deployed with the same pod definition, get the same host name as localhost.
- So they are accessible to each other as localhost:<port No>
- The parent container has to change the way earlier it was linked to the child container.
- It has to look for linked container bas localhost insetead of any other name.

Sharing the Volume in POD

- As in docker, the containers here can have their own home directory path from the POD and multiple containers in the same pod can share the same home path as volume mount.
- The containers can exchange data through this shared local volume.
- But when the POD is restarted, every data initialized by the containers is lost.
- We can have Persistent volume from host file system as shared volume to manage the state across the restarts.

Persistent Volume

- The PersistentVolume subsystem abstracts details of how storage is provided from how it is consumed.
- Two new objects manage the persistent volume.
- PersistentVolume and PersistentVolumeClaim.
- The PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator.
- The PersistentVolume is a resource in the cluster just like a node is a cluster resource.

Persistent Volume Support

- Kubernetes supports **PersistentVolumes** of type hostPath. These PersistentVolumes are mapped to a directory inside the minikube VM.
- The Minikube VM boots into a tmpfs, so most directories will not be persisted across reboots (minikube stop).
- However, Minikube is configured to persist files stored under the following host directories.
  - /data
  - /var/lib/localkube
  - /var/lib/docker

PersistentVolume Config

```
apiVersion: v1
kind: PersistentVolume
metadata: name: pv0001
 spec:
    accessModes: - ReadWrite Once
    capacity:
      storage: 5Gi
   hostPath: path: /data/pv0001/
```

PV to POD

- The PVs are volume plug-ins like Volumes, but have a lifecycle independent of any individual pod that uses the PV.
- So they maintain the path from **host file system** and independently manage the storage.le.
- It is initialized by a definition in YML file and loaded with kubectl command.
- It is monitored just as any other resource in the cluster.

PersistentVolumeClaim

- A PersistentVolumeClaim (PVC) is a request for storage by a user.
- It is similar to a pod.
- Pods consume node resources and PVCs consume PV resources.
- Pods can request specific levels of resources (CPU and Memory).
- Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).

PVC to Container in POD

- While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems.
- Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented.

PV and PVC Lifecycle

- PVs are resources in the cluster.
- PVCs are requests for those resources and also act as claim checks to the resource.
- The interaction between PVs and PVCs follows this lifecycle:
- **Provisioning:** statically or dynamically.
- **Binding:** linkiing
- **Using:** The Pods use claims as volumes.
- Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it

Controller Objects

- Controllers build upon the basic objects, and provide additional functionality and convenience features.
- ReplicaSet : New version of Replication Controller
- Deployment: *Deployment* controller provides declarative updates for Pods and ReplicaSets.
- StatefulSet: StatefulSet is the workload API object used to manage stateful applications.

Controller Objects

- **DaemonSet:A** *DaemonSet* ensures that all or some Nodes run a copy of a Pod.
- Deleting a DaemonSet will clean up the Pods it created
- As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected.
- **Job**: A *job* creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the *job* tracks the successful completions.

Replica set with pods

- Replica sets deliver the required scale and availability by maintaining a pre-defined set of pods at all times.
- A single pod or a replica set can be exposed to the internal or external consumers via services.

Abstraction

- The etcd is an open source, distributed key-value database from CoreOS, which acts as the single source of information for all components of the Kubernetes cluster.
- The master queries etcd to retrieve various parameters of the state of the nodes, pods and containers.
- This architecture of Kubernetes makes it modular and scalable by creating an abstraction between the applications and the underlying infrastructure.

Workload Scalability

- Each container is designed to perform only one task(image).
- Pods can be composed of stateless containers or stateful containers.
- Stateless pods can easily be scaled on-demand or through dynamic auto-scaling.
- Kubernetes supports horizontal pod auto-scaling, which automatically scales the number of pods in a replication controller based on CPU utilization.

App Deployments

- Applications deployed in Kubernetes are packaged as MicroServices.
- These MicroServices are composed of multiple containers grouped as pods.
- Hosted Kubernetes running on Google Cloud also supports cluster auto-scaling. When pods are scaled across all available nodes, Kubernetes coordinates with the underlying infrastructure to add additional nodes to the cluster.

List sorting

- To output objects to a sorted list in the terminal window, add the --sort-by flag to a supported kubectl command.
- Sort your objects by specifying any numeric or string field with the --sort-by flag.
- To specify a field, use a jsonpath expression.
- kubectl [command] [TYPE] [NAME] --sort-by=<jsonpath_exp>
- To print a list of pods sorted by name, run:
- kubectl get pods --sort-by=.metadata.name

High Availability

- The application workloads demand availability at both the infrastructure and application levels.
- In clusters at scale, everything is prone to failure, which makes high availability for production workloads strictly necessary.
- While most container orchestration engines and PaaS offerings deliver application availability, Kubernetes is designed to tackle the availability of both infrastructure and applications.

Make it available

- Kubernetes ensures high availability by means of replica sets, replication controllers and pod sets.
- Operators can declare the minimum number of pods that need to run at any given point of time.
- If a container or pod crashes due to an error, the declarative policy can bring back the deployment to the desired configuration.
- Stateful workloads can be configured for high availability through pet sets.

Security

- Security in Kubernetes is configured at multiple levels.
- The API endpoints are secured through transport layer security (TLS), which ensures the user is authenticated using the most secure mechanism available.
- Kubernetes clusters have two categories of users — service accounts managed directly by Kubernetes, and normal users assumed to be managed by an independent service.

Portability

- Kubernetes is designed to offer freedom of choice when choosing operating systems, container runtimes, processor architectures, cloud platforms and PaaS.
- A Kubernetes cluster can be configured on mainstream Linux distributions, including CentOS, CoreOS, Debian, Fedora, Red Hat Linux and Ubuntu.

Containers

- It can be deployed to run on local development machines; cloud platforms such as AWS, Azure and Google Cloud; virtualization environments based on KVM, vSphere and libvirt; and bare metal.
- Users can launch containers that run on Docker or rkt runtimes, and new container runtimes can be accommodated in the future.

Health Checks

- Kubernetes supports user-implemented application health checks.
- These checks are performed by the kubelet running on each minion to ensure that the application is operating correctly. Currently,

Supported Health Checks

- Kubernetes supports three types of health checks:
- **HTTP health check**: The kubelet will call a web endpoint. If the response code is between 200 and 399, it is considered a success.
- **Container exec**: The kubelet will execute a command within the container. If it returns “OK,” it is considered a success.
- **TCP socket**: The kubelet will attempt to open a socket to the container and establish a connection. If the connection is made, it is considered healthy.

Dashboard

- The dashboard is the web application, part of cluster to monitor and control the minikube cluster.
- To launch it : minikube dashboard
- URL : http://192.168.99.100:30000
- View the cluster objects like pods,services,volumes,replicas etc.
- Control the life cycle of cluster components
- Monitor the events for cluster object
- Deploy new components

POD Life cycle

- Pods are the deployment units in Kubernetes clsuer which represent the groups of containers.
- Kubernetes *Pods* are itself mortal. They are born and when they die, they are not resurrected.
- Each Pod gets its own IP address when started , those IP addresses cannot be relied upon to be stable over time.

Services to track PODs

- When some set of Pods as back-ends provides functionality to other Pods as frontends inside the Kubernetes cluster, how do those frontends find out and keep track of which back-ends are in that set?
- What if some pods in the backend gets down/crashed/started new instances ?
- How to manage the pods life automatically.
- How to decide which pod to send request from front end pods.?
- The service objects support these pod management.

Services

- A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them.
- The set of Pods targeted by a Service is usually determined by a Label Selector .
- In an application cluster, the frontend pods do not care which backend they use.
- While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that or keep track of the list of back-ends themselves.
- The Service abstraction enables this decoupling.

Service port mapping

- Kubernetes Services support TCP and UDP for protocols. The default is TCP.
- The service can map an incoming port to any targetPort of the POD containers.
- By default the targetPort will be set to the same value as the port field in container port.
  - The targetPort of service can be mapped to the POD container port as port mapping .
  - The service targetPort value can be dynamically assigned which will act as front end port for pOD containers.

Service Port Name

- The targetPort can be a string, referring to the name of a port in the backend Pods. The actual port number assigned to that name can be different in each backend Pod.
- This offers a lot of flexibility for deploying and evolving the Services.
- For example, the port number that pods expose in the next version of the backend software can be changed , without breaking clients.

Discovery of POD

- Services enable the discovery of pods by associating a set of pods to a specific criterion.
- Pods are associated to services through key-value pairs called labels and selectors.
- Any new pod with labels that match the selector will automatically be discovered by the service. This architecture provides a flexible, loosely-coupled mechanism for service discovery.

Service Types

- The applications in pods such as frontends expose through a Service onto an external world outside of the cluster boundary.
- **ClusterIP**: Exposes the service on a cluster-internal IP. This makes the service only reachable from within the cluster.
- This is the default Service Type.

NodePort Service

- **NodePort**: Exposes the service on each Node's IP at a static port (the NodePort).
- A ClusterIP service, to which the NodePort service will route, is automatically created.
- To contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
  - Kubernetes master allocates a port from a range specified by --service-node-port-range flag (default: 30000-32767), and each Node will proxy that port (the same port number on every Node) into the Service.
  - That port get reported in the Service's .spec.ports[*].nodePort field.

LoadBalancer service

- **LoadBalancer**: Exposes the service externally using a cloud/cluster provider's load balancer.
- Internally the NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.
- The service is accessible by an external ip address assigned by the cluster provider.
- Traffic from the external load balancer is directed at the backend Pods.

ExternalName Service

- **ExternalName**: Maps the service to the contents of the externalName field (e.g. user.in.server.com), by returning a CNAME record with its value.
- No proxying of any kind is set up. This requires a kube-dns deployment in the cluster.
- Type NodePort

Scaling the Service

- You can specify the initial instances of containers in the pod as replicas in the deployment definition.
- The ReplicationController or ReplicaSet controls the replication management.
- The Service itself is another controller in Kubernetes API.
- The instances of containers in the pods can be scaled up and down by using kubectl scale command.
-

Service commands

- Read the services
- sudo kubectl get services
- sudo kubectl get service py-app
- sudo kubectl describe service py-app
- Scale the deployments
- kubectl scale --current -replicas=2 --replicas=3 deployment/mysql
- Scale the replication controller 'userData' to 3.
- kubectl scale --replicas=3 rc/userData

Life of POD in service

When any of the pods goes down or gets crashed, the service controller automatically creates new instances of it and tracks their life.

Update the service

- Update the services with
  - New version of image for pod containers
  - New configuration values
  - New POD definitions
  - Newer deployments
- No need to stop the existing service and restart with new configuration/new images etc.
- The rolling update supports in place updates.

Rolling Update

- **Rolling updates** allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones.
- The new Pods get scheduled on Nodes with available resources.
- Scaling the application to run multiple instances is also the requirement for performing updates without affecting application availability.
- By default, the maximum number of Pods that can be unavailable during the update and the maximum number of new Pods that can be created, is one.

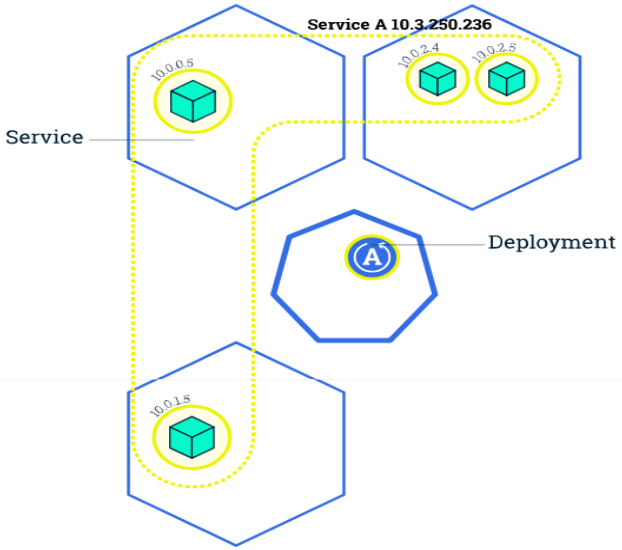
Versioned Updates

- In Kubernetes, updates are versioned and any Deployment update can be reverted to previous (stable) version.
- Similar to application Scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update.
- An available Pod is an instance that is available to the users of the application.

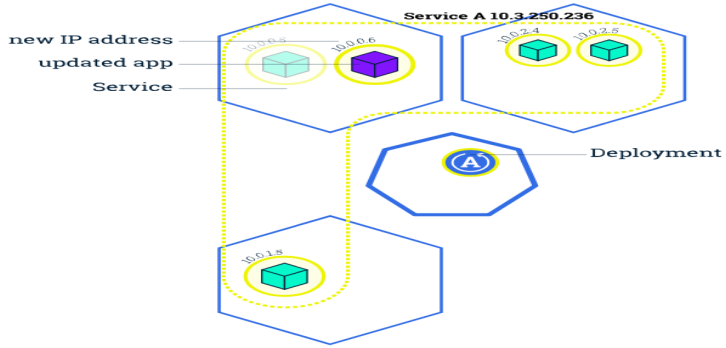
Rolling Update Actions

- Rolling updates allow the following actions:
- Promote an application from one environment to another (via container image updates)
- Rollback to previous versions
- Continuous Integration and Continuous Delivery of applications with zero downtime.
- Perform a rollback

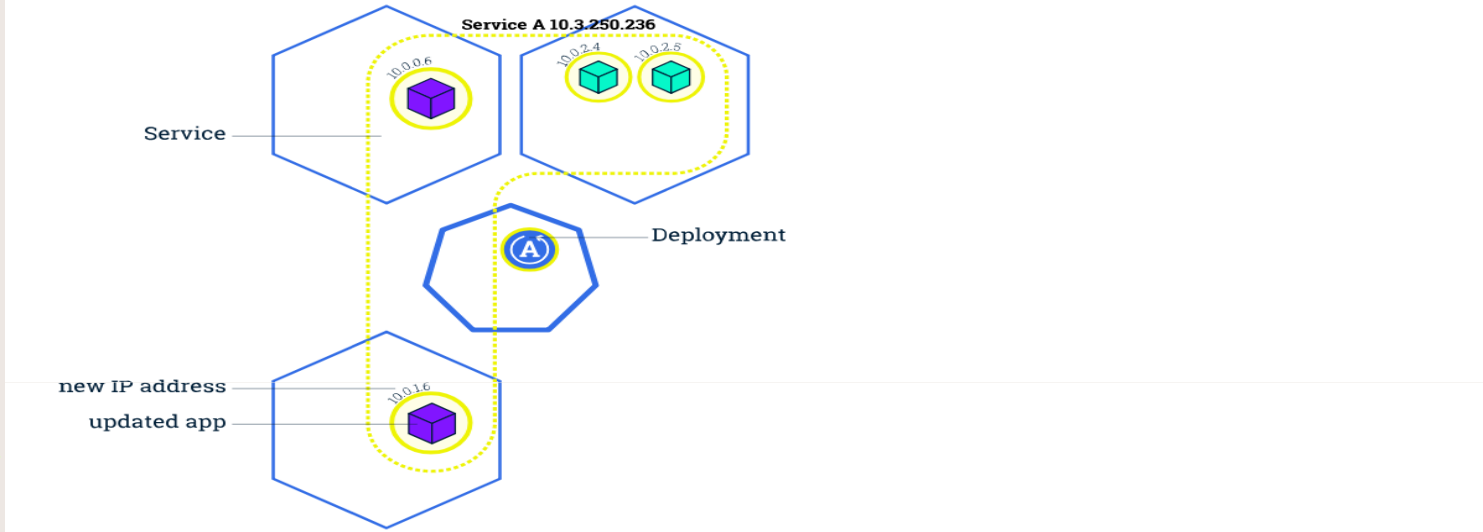
Rolling update service



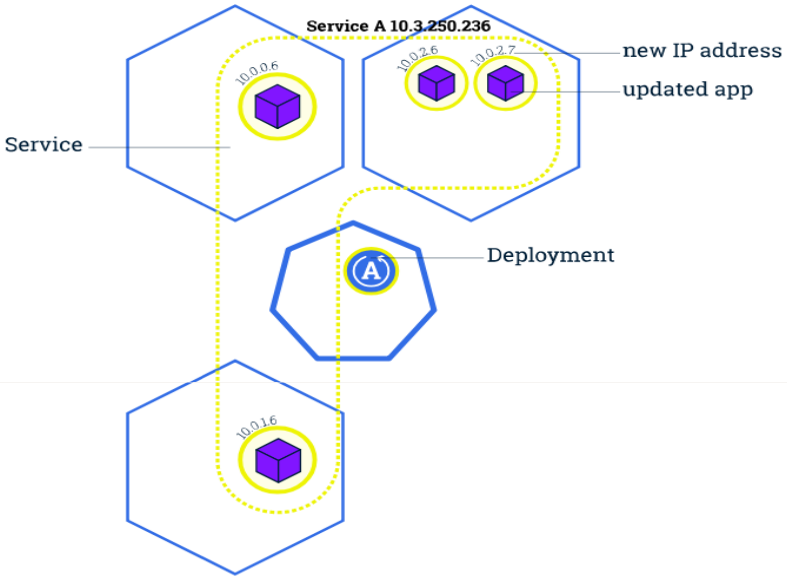
Update Stage1



Update Stage2



Update Stage3



RollOut commands

- The kubectl rollout command manages a deployment.
- Update the deployment by changing the image
- kubectl set image deployment/default-http-backend
- kubectl rollout status deployment/nginx-deployment
- kubectl rollout history deployment/nginx-deployment
- kubectl rollout pause RESOURCE
- kubectl rollout resume RESOURCE
- To roll back
- kubectl rollout undo deployment/nginx-deployment
-

Cluster Monitoring

- Monitoring an application's current state is one of the most effective ways to anticipate problems and discover bottlenecks in a production environment.
- Network Administrators handling availability, performance, and deployments are making teams to create, or use, orchestrators to handle all the services and servers.
- A container orchestration tool such as Kubernetes handles containers in several computers, and removes the complexity of handling distributed processing.

Dashboard and Service

- To access the Kubernetes Dashboard, run the command in a shell after starting minikube to get the address:
- minikube dashboard
- **Services**
- To access a service exposed via a node port, run this command in a shell after starting minikube to get the address
- minikube service [-n NAMESPACE] [--url] NAME

Application Health

- Monitors logs from
  - PODs
  - Containers
  - Services
- Monitor resources
  - Volume
  - Deployments
  - ReplicasSets

Resource Usage

- kubectl top command usage to to see the resource consumption for nodes or pods.
- Display Resource Usage for CPU,Memory,Storage for nodes and pods.
- Kubectl top <pod name/id>
- Kubectl top <Node name/id>
- This command requires **Heapster** to be correctly configured and working on the server.

Kubectl top options

- **--cluster**=The name of the kubeconfig cluster to use
- **--context**=The name of the kubeconfig context to use

REST API Access

- Run kubectl in proxy mode (recommended). it uses the stored api-server location and verifies the identity of the API server using a self-signed cert.
- Provide the location and credentials directly to the http client. This works with client code that is confused by proxies.
- http://localhost:8080/api/

Cluster Access

- To access a cluster, you need to know the location of the cluster and have credentials to access it
- Check the location and credentials
- $ kubectl config view
- Using kubectl proxy
- Runs kubectl in a mode where it acts as a reverse proxy.
- It handles locating the API server and authenticating.
- kubectl proxy --port=8080 &

Proxy config

- **Configuring the Proxy**
- The easiest way to access Kubernetes API is to configure a proxy.
- The kubectl comes with an option to configure it.
- Open a new terminal window and run the following command to create the proxy.
- kubectl proxy --port=8000
- Through this proxy session, any request sent to **localhost:8000** will be forwarded to the Kubernetes API server.

Kubectl REST API

- http://localhost:8000/api
- http://localhost:8000/api/v1/nodes | jq '.items[] .metadata.labels‘
- http://localhost:8000/api/v1/namespaces/default/services
- http://localhost:8000/v1/proxy/namespaces/default/services

Create New POD via POST

- Create nginx-pod.json file.
- Create POD by post
- curl -s http://localhost:8000/api/v1/namespaces/default/pods \ -XPOST -H 'Content-Type: application/json' \ -d@nginx-pod.json \ | jq '.status‘
- Create service by POST
- curl -s http://localhost:8000/api/v1/namespaces/default/services \ -XPOST -H 'Content-Type: application/json' \ -d@nginx-service.json \ | jq '.spec.clusterIP'

Verify the pods

- curl http://localhost:8000/v1/proxy/namespaces/default/services/nginx-service/

Kubernetes in Cloud

- Setup Kubernetes cluster on cloud machines
- Kubernetes cluster pre-configurred from cloud and available as service.
- Azure Kubernetes service(AKS)
- AWS Kubernetes service
- GCP Kubernetes service(GKS)

Network Requirements

- Network for Distributed Computing:
  - The network should be reliable
  - The network Latency should be zero
  - Bandwidth should be infinite
  - The network should be secure
  - The network topology should not Change
  - There is one Administrator
  - The network transport cost should be minimum
  - The network should be homogenous

Network dependency

- As the number of micro-services increase, you have to deal with the interactions between them, monitor the overall system health, be fault tolerant, have logging and telemetry in place, handle multiple points of failure, and more.
- Each of the micro-services needs to have these common functionalities in place so that the service to service communication is smooth and reliable.

Service Mesh for microservices

- The service mesh is defined as an infrastructure layer which handles the inter-service communication in a micro-service architecture.
- Service mesh reduces the complexity associated with a MicroService application and supports provides lot of the functionalities.
  - Load balancing
  - Service discovery
  - Health checks
  - Authentication

ServiceMesh to Micro-services

- Traffic management and routing
- Circuit breaking and failover policy
- Security
- Metrics and telemetry
- Fault injection

ServiceMesh at high level

- **The service mesh** layers on top of the **Kubernetes** infrastructure and is responsible for making communications between **services** over the network safe and reliable
- The Service mesh allows to separate the business logic of the application from observability, and network and security policies.

Lstio service mesh

- **Istio** is an open source independent **service mesh** that provides the fundamentals you need to successfully run a distributed microservice architecture.
- The Istio reduces complexity of managing microservice deployments by providing a uniform way to secure, connect, and monitor microservices.
- It controls how microservices share data with one another and includes APIs that let **Istio** integrate into any logging platform, telemetry, or policy system for monitoring.

StackDriver on GCP

Monitor, troubleshoot, and improve application performance on the Google Cloud environment.

- Collect metrics, logs, and traces across Google Cloud and your applications
- Use built-in out-of-the-box dashboards and views to monitor the platform and applications
- Query and analyze these signals
- Set up appropriate performance and availability indicators
- Set up alerts and notification rules with your existing systems.

Thank You