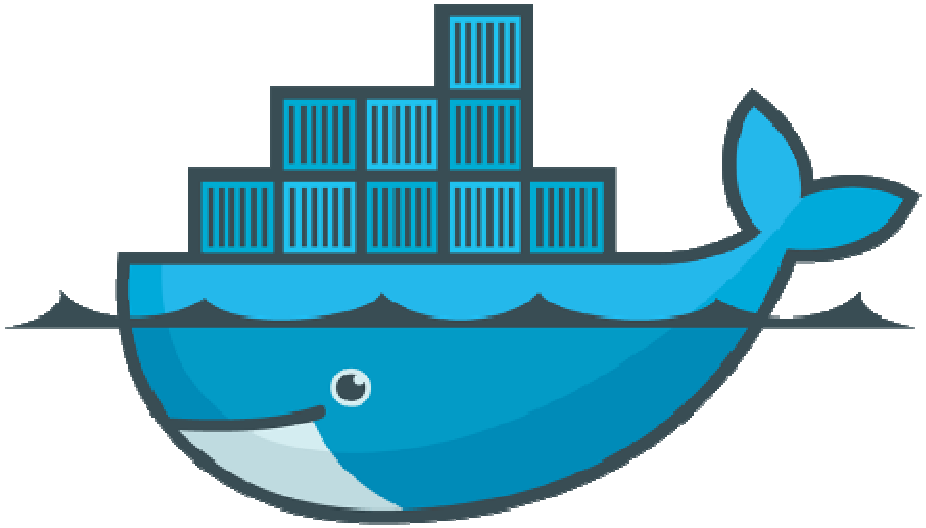
Docker Introduction



Prakash Badhe
prakash.badhe@vishwasoft.in

Docker setup

- Hardware virtualization needed.
- Windows 10 64 bit with **Docker for windows** as native application.(**NO VM needed**)
- Linux with Docker as native application.(NO VM needed)
- Windows 7 64 bit with Docker ToolBox(Creates **Linux VM** on windows and installs Docker on the Linux VM).
- Mac os with Docker ToolBox(Creates **Linux VM** on Mac and installs Docker on the Linux VM).

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

Setup : Windows 7 64 bit

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

Setup : Ubuntu Linux

- Linux Ubuntu 16.0 onwards
- Hardware virtualization enabled in BIOS
- Docker for Linux Ubuntu
- 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
- Ports 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
- OS security compromised and affects the deployment
- Load balancing customized differently for different application servers.
- Scaling is not so easy without adding an exetrnal node.

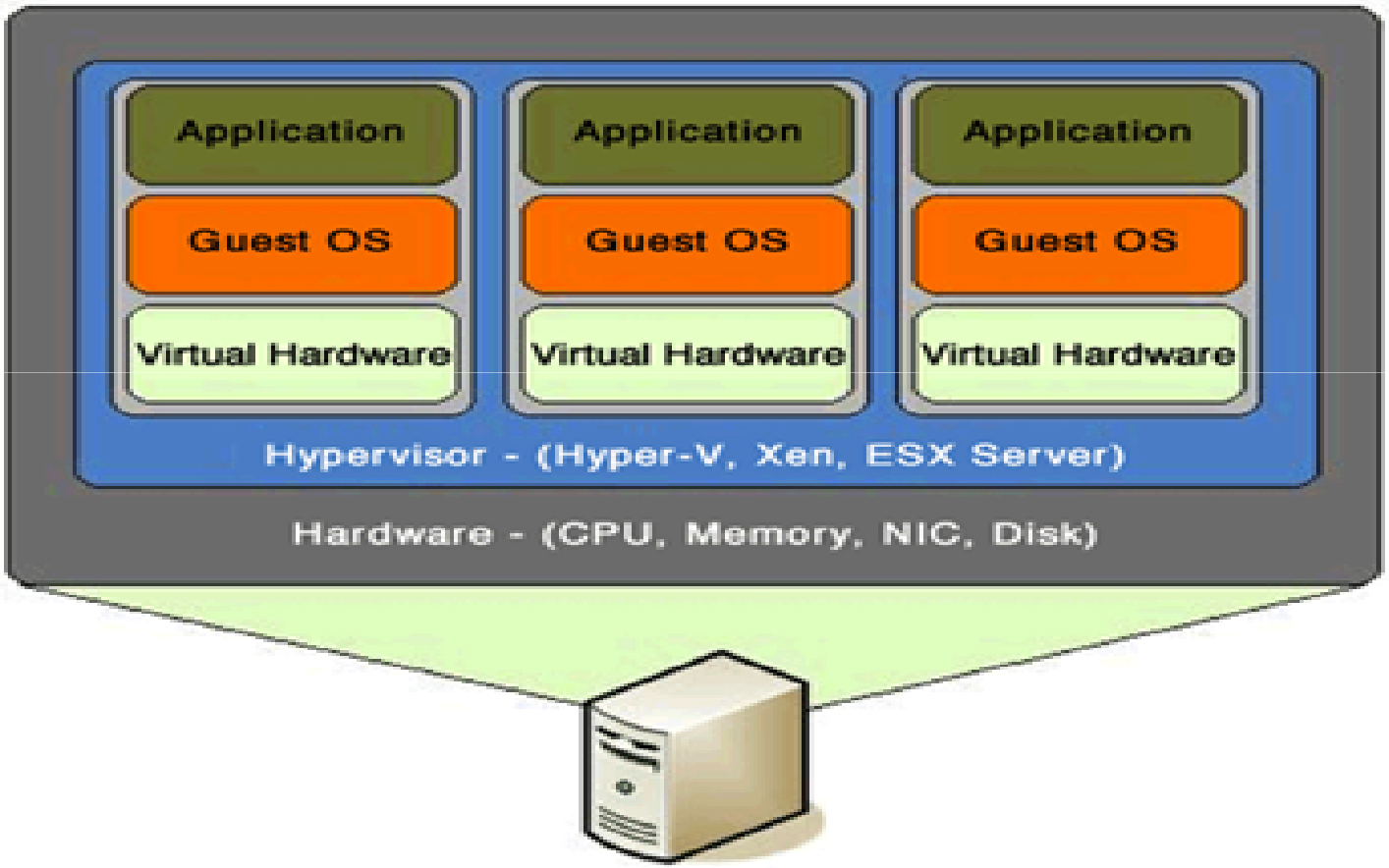
Deployment Solutions

- 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

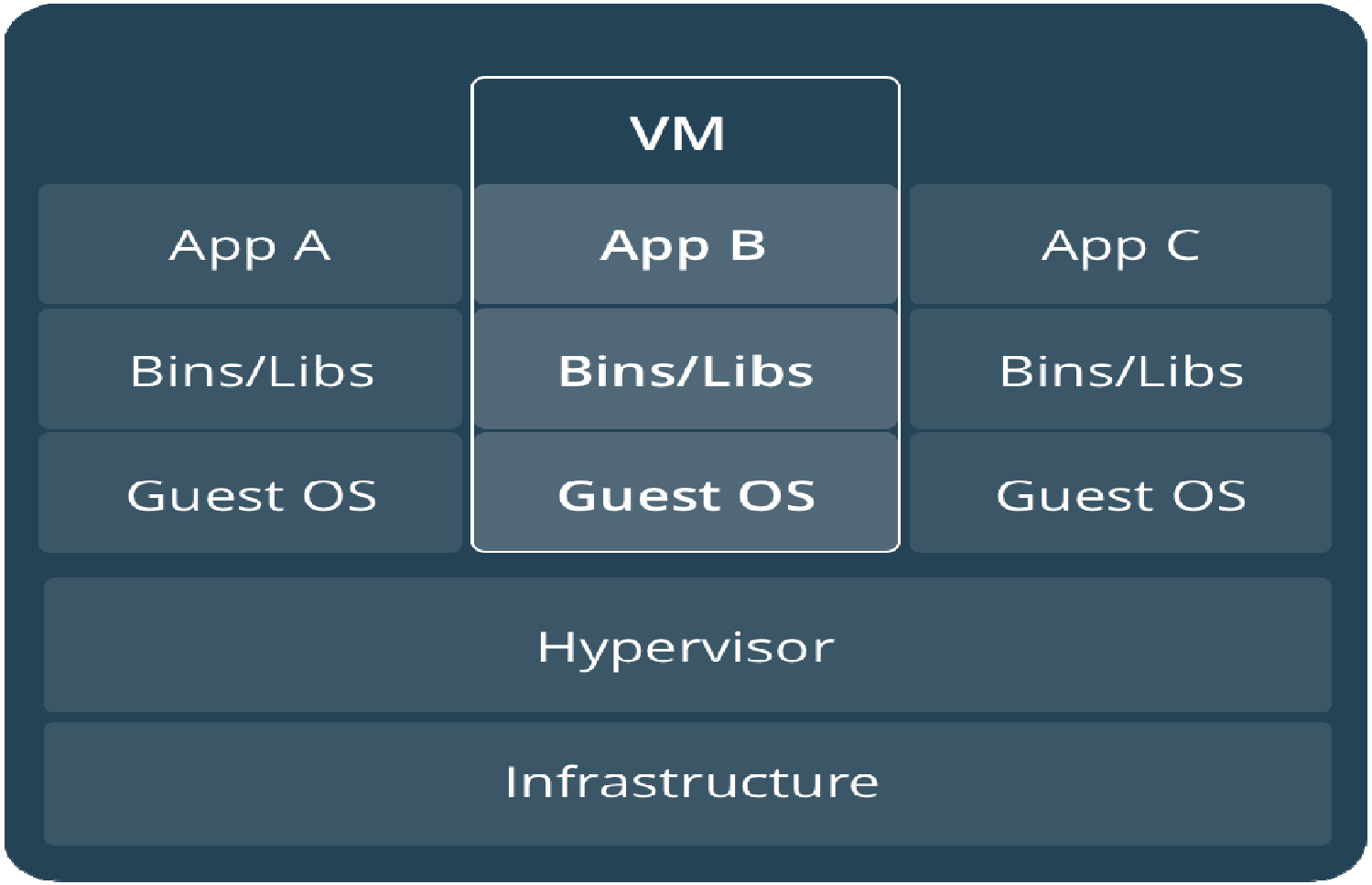
Virtual Machine

- 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 need more resources and memory.
- **The VMs are isolated from other VMs on the same base OS.**
  - **Inter-communication and sharing across the VMs 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



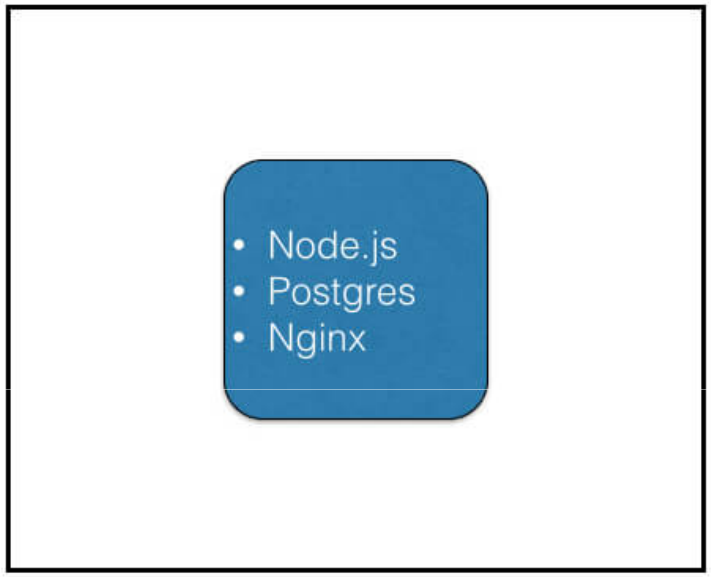
VM limitations

- 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

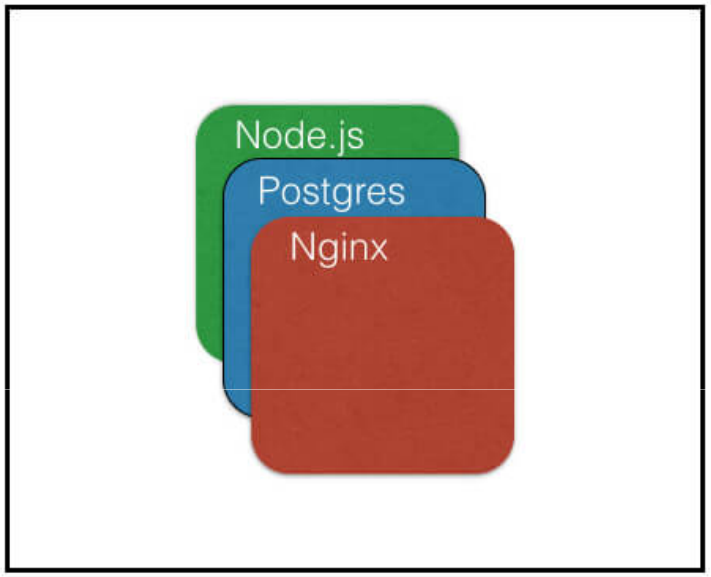
This container can contain any different types of items.

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

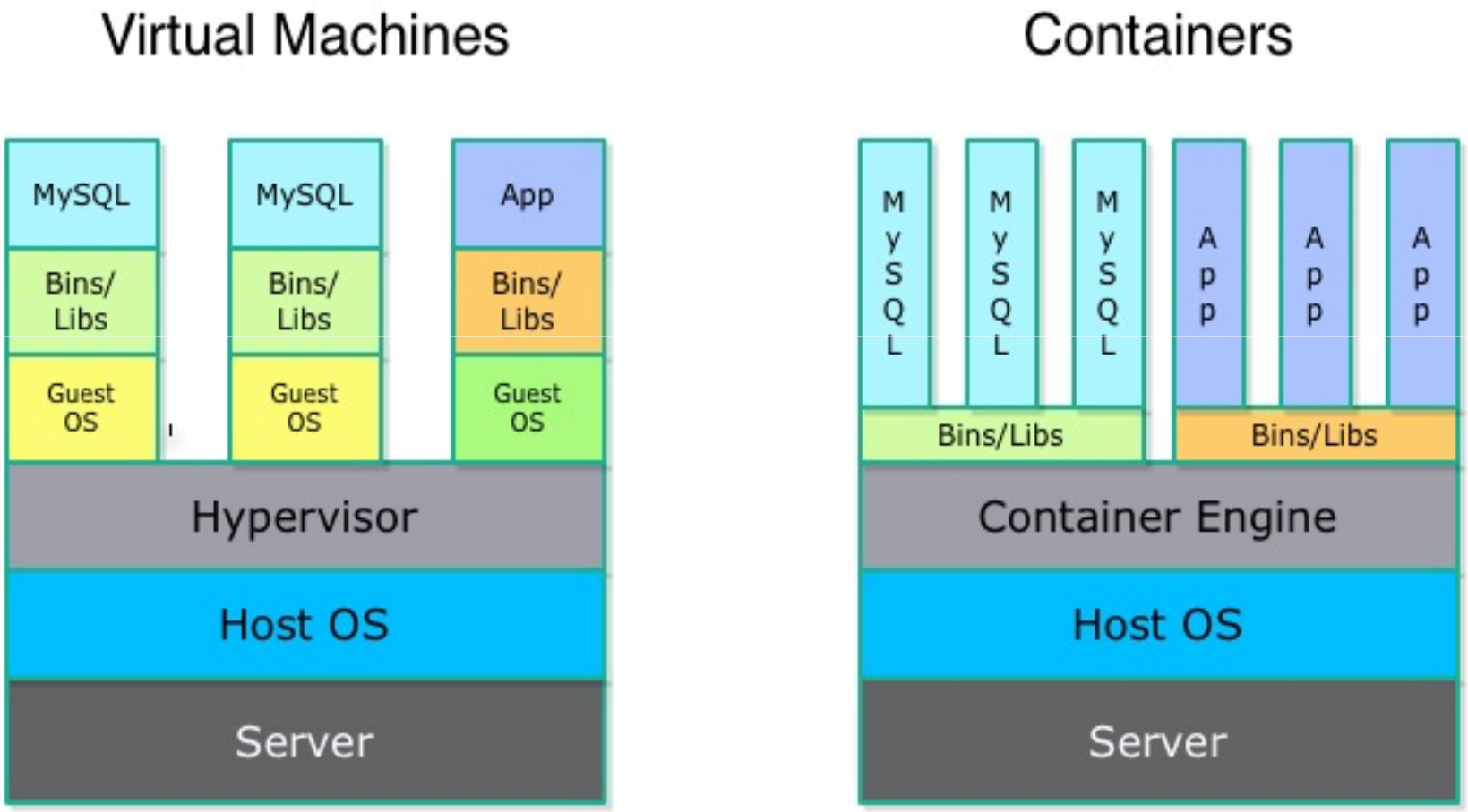
Application Container

- 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



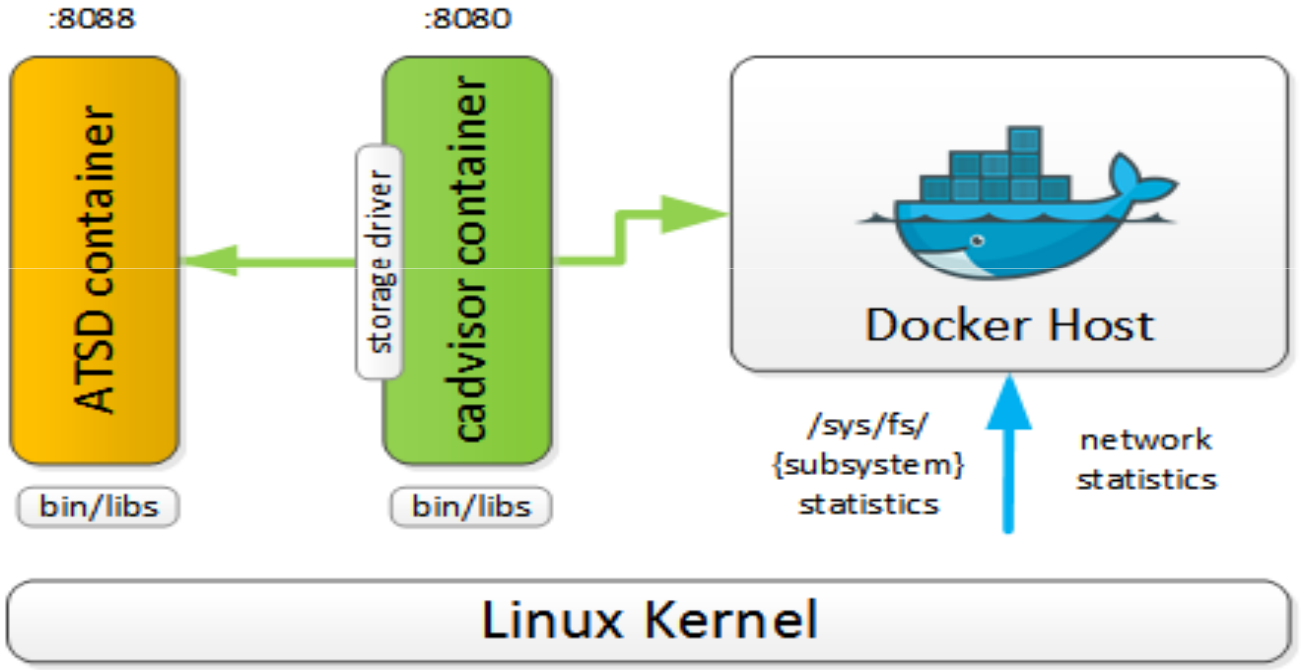
Docker Platform

- Docker is a platform for developers and sysadmins to **develop, deploy, and run**applications with containers.
- The use of Linux containers to deploy applications is called *containerization*.
- Containers are not new, but their use for easily deploying applications is New.

Docker Container Engine

- 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.

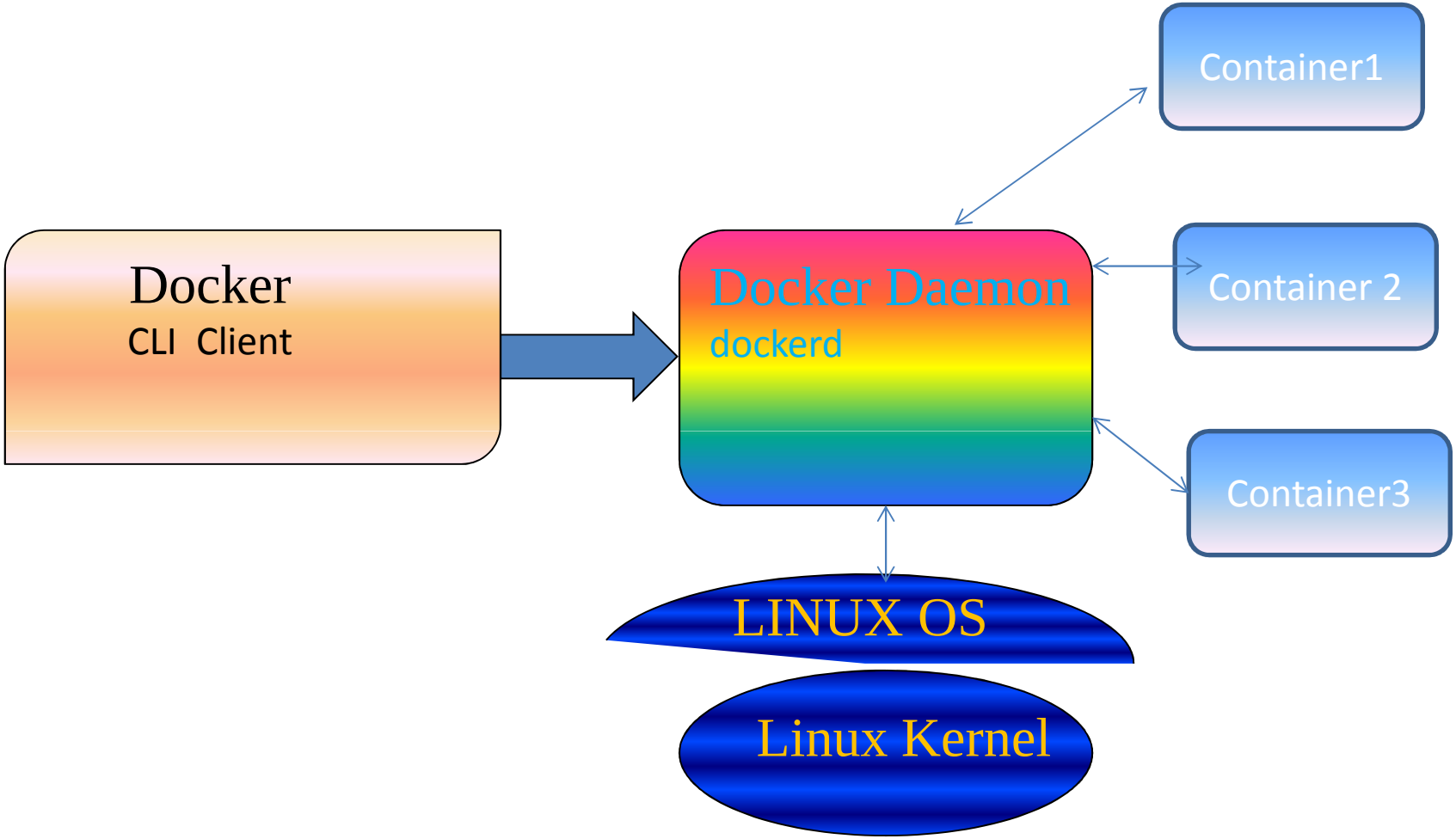
Docker Containers



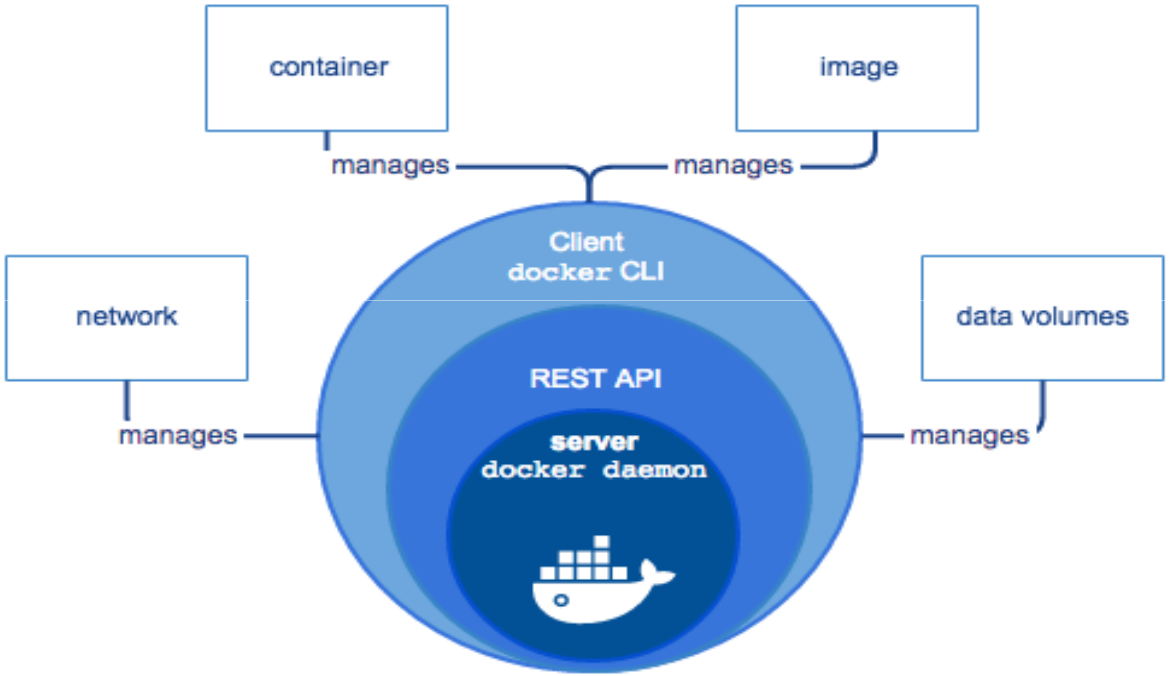
Docker tools

- 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.

Docker Process



Docker Daemon engine



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.

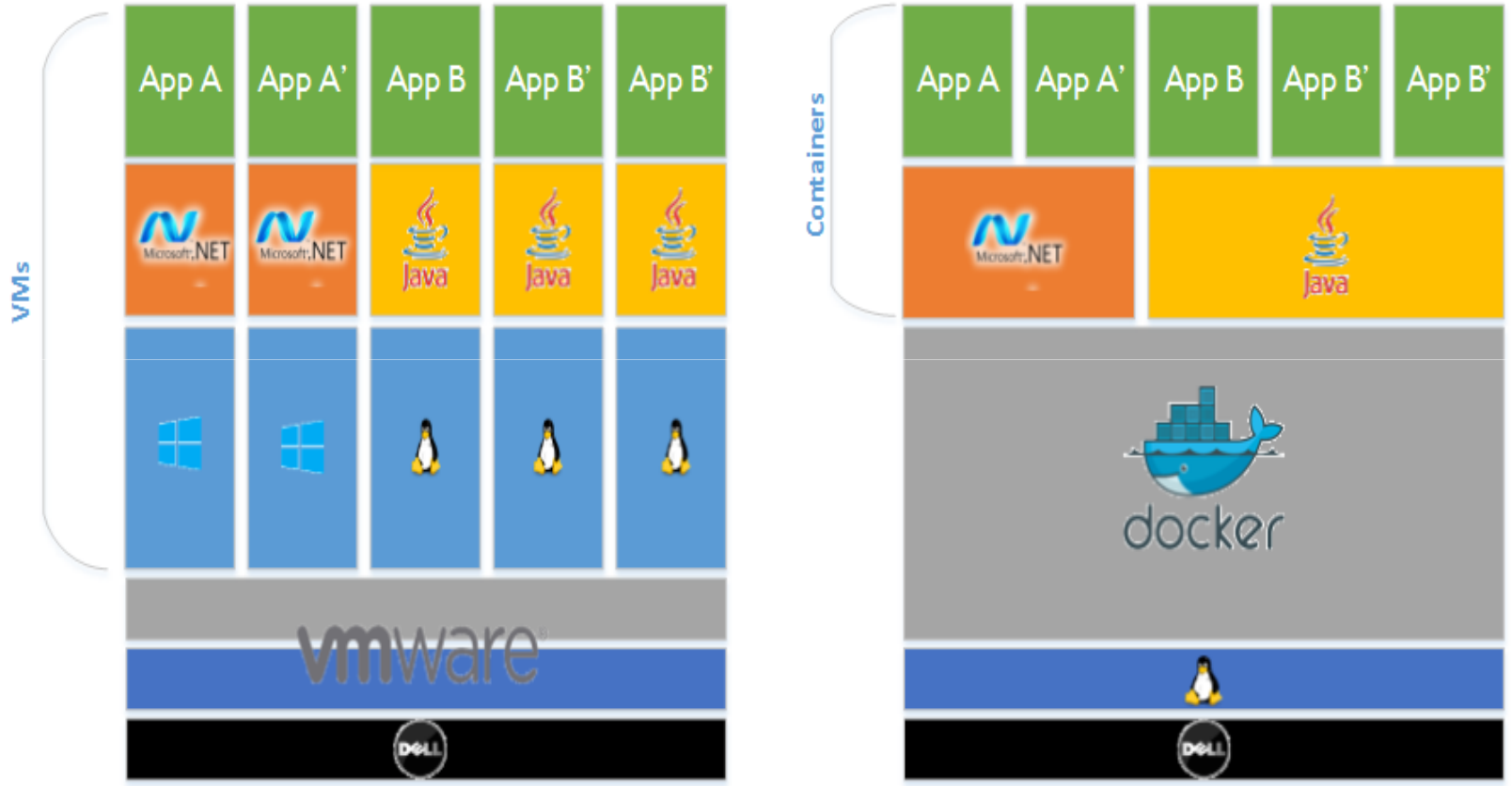
Docker Container Management

- 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 Support

- Container instance management
- Cluster with Docker containers
- Load balancing across containers
- Monitoring the container operations

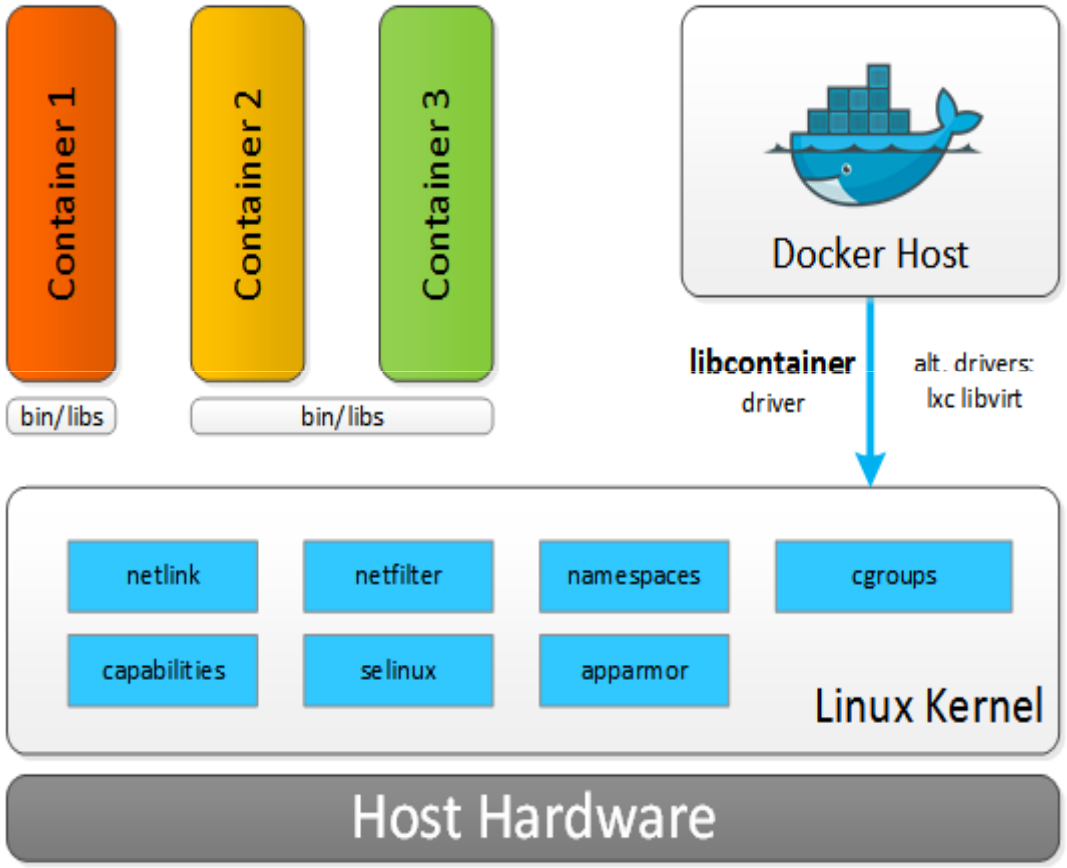
Applications on Docker vs. VM



Containers and virtual machines

- A **container** runs *natively* on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.
- By contrast, a **virtual machine** (VM) runs a full-blown “guest” operating system with *virtual* access to host resources through a hypervisor.
- The VMs provide an environment with **more** resources than most applications need.

Docker Container applications



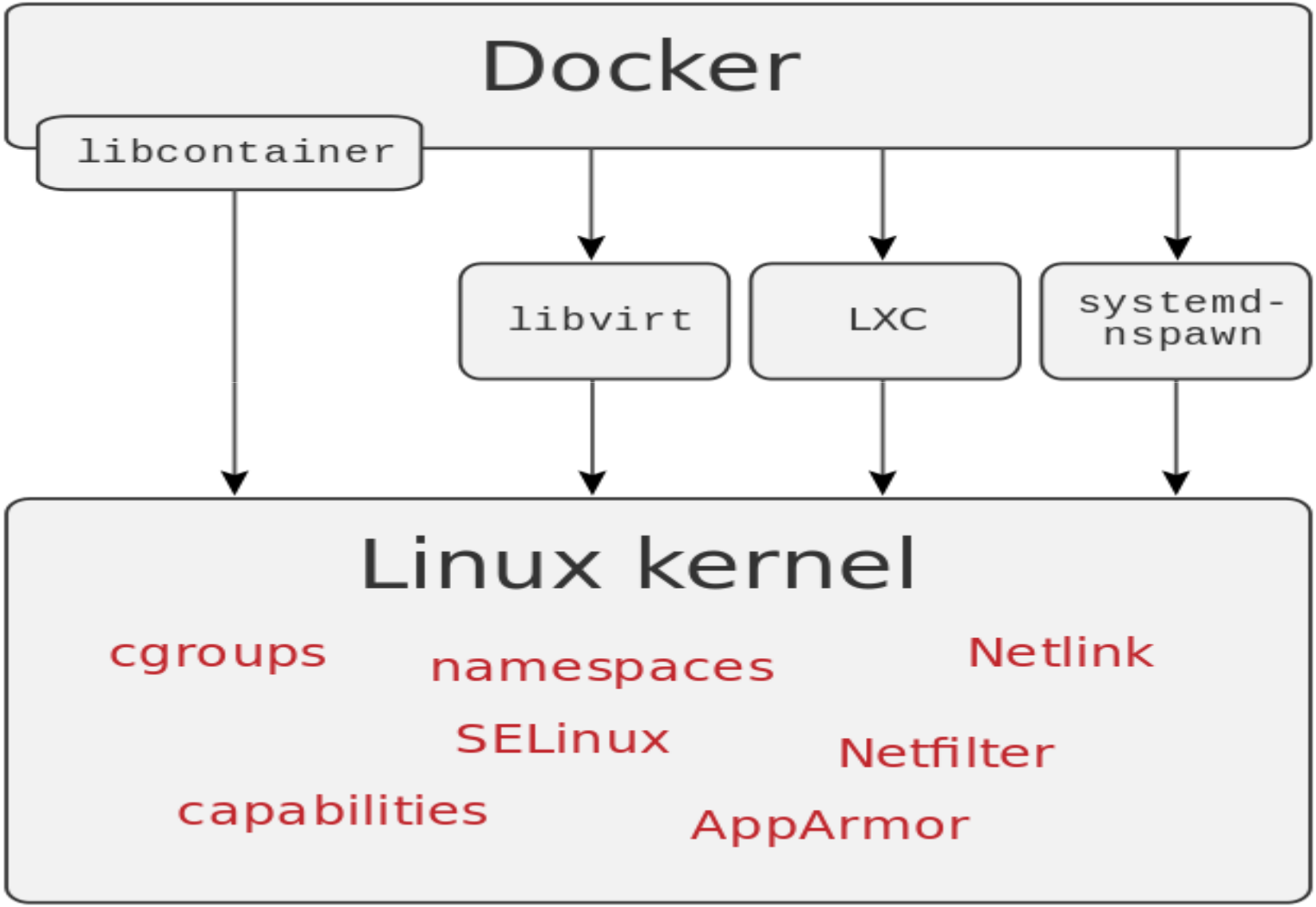
Universal deployments

- 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

Docker Container Support

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

Docker Internals



Docker Container

- A container is launched by running an image.
- An **image** is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.
- A **container** is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process).

Start with Docker

- List Docker CLI commands
  - docker –help
  - docker container --help
  - docker –version
  - docker info

Pull image and start container

- Pull the docker image from docker hub
  - docker pull hello-world
  - Check the images pulled on local machine
  - docker image ls
  - Create the docker container from image pulled
  - docker create hello-world
  - Check the container created
  - docker container ls –a
  - Run the docker container
  - docker start container-id/name
  - Check the container running
  - Docker ps or docker ps –a
  - Check the log of container
  - docker logs container-id/name
  - docker stop container-id/name
  - To remove the container
  - docker rm container-name/id
  - docker run hello-world (Fully combined command)

Run with proxy

- Set the proxy with Docker for windows
- Set the proxy in config file on Linux in /etc/default/docker or /etc/sysconfig/docker
- Pass the proxy settings from command line
- docker build --build-arg

http_proxy=http://192.168.33.10:3128

https_proxy=http://192.168.33.10:3128

Docker Images and Containers

- List Docker images
  - docker image ls
- List Docker containers (running, all, all in quiet mode)
  - docker container ls
  - docker container ls --all
  - docker container ls -aq

Docker 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.

Docker Hub Registry

- The ‘https://hub.docker.com/ ‘ is the registry/repository that hosts thousands of dockers application 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 this registry.
- Can we have a local registry in our network?
  - Reduce the connectivity issues
  - Make our images private to us only.
  - Save time on building the images.

Create Your own Image

- Create an empty directory on your local machine.
- Change into the new directory.
- Create a file called Dockerfile
- Add the code into that file, and save it.

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.

Custom Dockerfile


```
# Use a parent image
FROM python:2.7-slim
# Set the working directory to /app
WORKDIR /app
#Copy the current directory contents into the container at /app
COPY . /app
# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt
# Make port 80 available to the world outside this container
EXPOSE 80
 # Define environment variable
ENV NAME World
# Run app.py when the container launches
CMD ["python", "app.py"]
```

Proxy settings in Dockerfile

- Proxy servers can block connections to your web app once it's up and running.
- If you are behind a proxy server, add the following lines to your Dockerfile, using the ENV command to specify the host and port for your proxy servers.
- # Set proxy server, replace host:port with values for your servers
- ENV http_proxy host:port
- ENV https_proxy host:port

Build the image

- From the directory
  - docker build --tag=py-app .
- Check the image built
  - docker image ls

Run the container from local Image

- Run the application container
  - docker run py-app
- Check the container properties with ip
  - docker describe <container-name or id>
- Check the container app on host in browser
  - http://container-ip:80
- To make the container accessible outside VM
- Map your host machine's port 4000 to the container's published port 80
  - docker run -p 4000:80 py-app
  - http://localhost:4000 or http://host-ip:4000
  - To stop the container
  - docker container stop <Container NAME or ID>

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.

Applications 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.

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.

Format the image display


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

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>

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 configurable.
- The nginx container process the access and error logs to Stdout and stderror separately.

Docker Container 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.

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.

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.

Docker Bridge Network

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 defined in the same docker-compose YML file are accessible by their names.

Linking 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

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.

Docker 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.

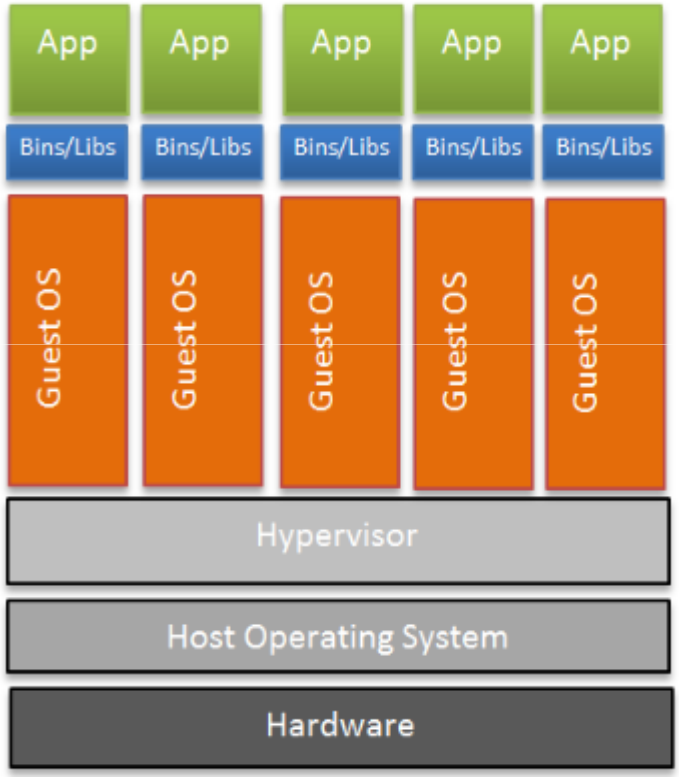
Service Object

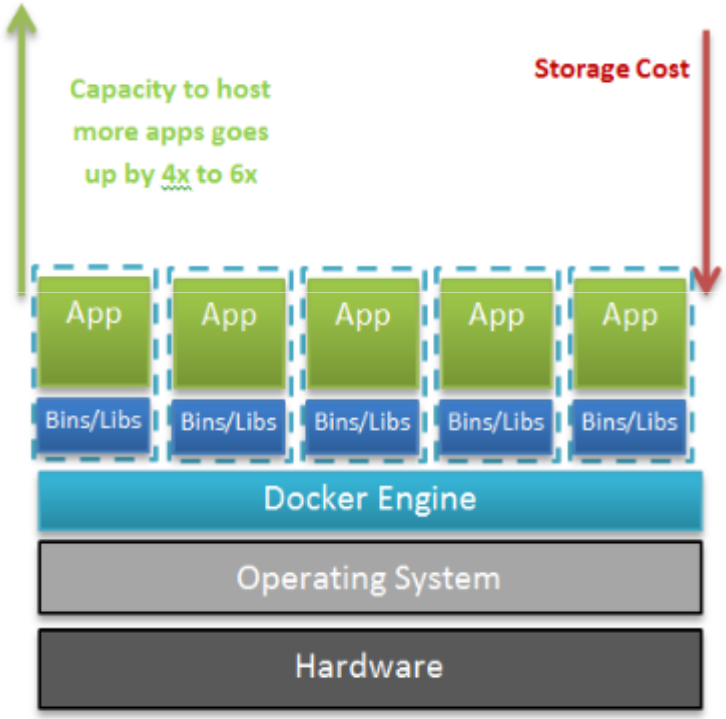
- The service is the image for a microservice within the context of some larger application.
- Examples of services might include an HTTP server, a database, or any other type of executable program that you wish to run in a distributed environment.
- Create a service, specify which container images to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

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.

Docker applications scaling





Scaling the applications

- 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.

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 Local 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’

More Docker Tools

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

Docker 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.

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.

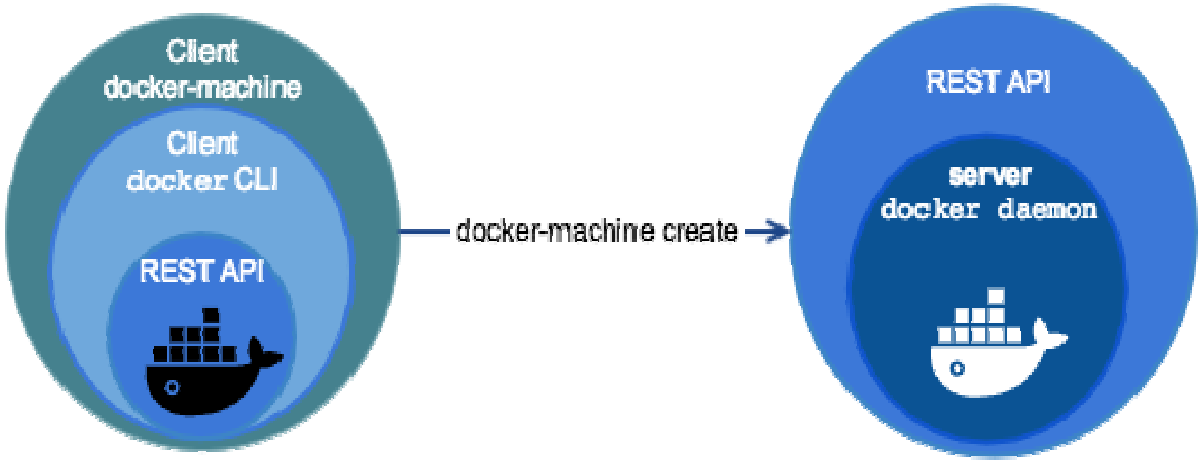
Docker-machine Tool

- 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“

Machine Communication

- 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.

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.

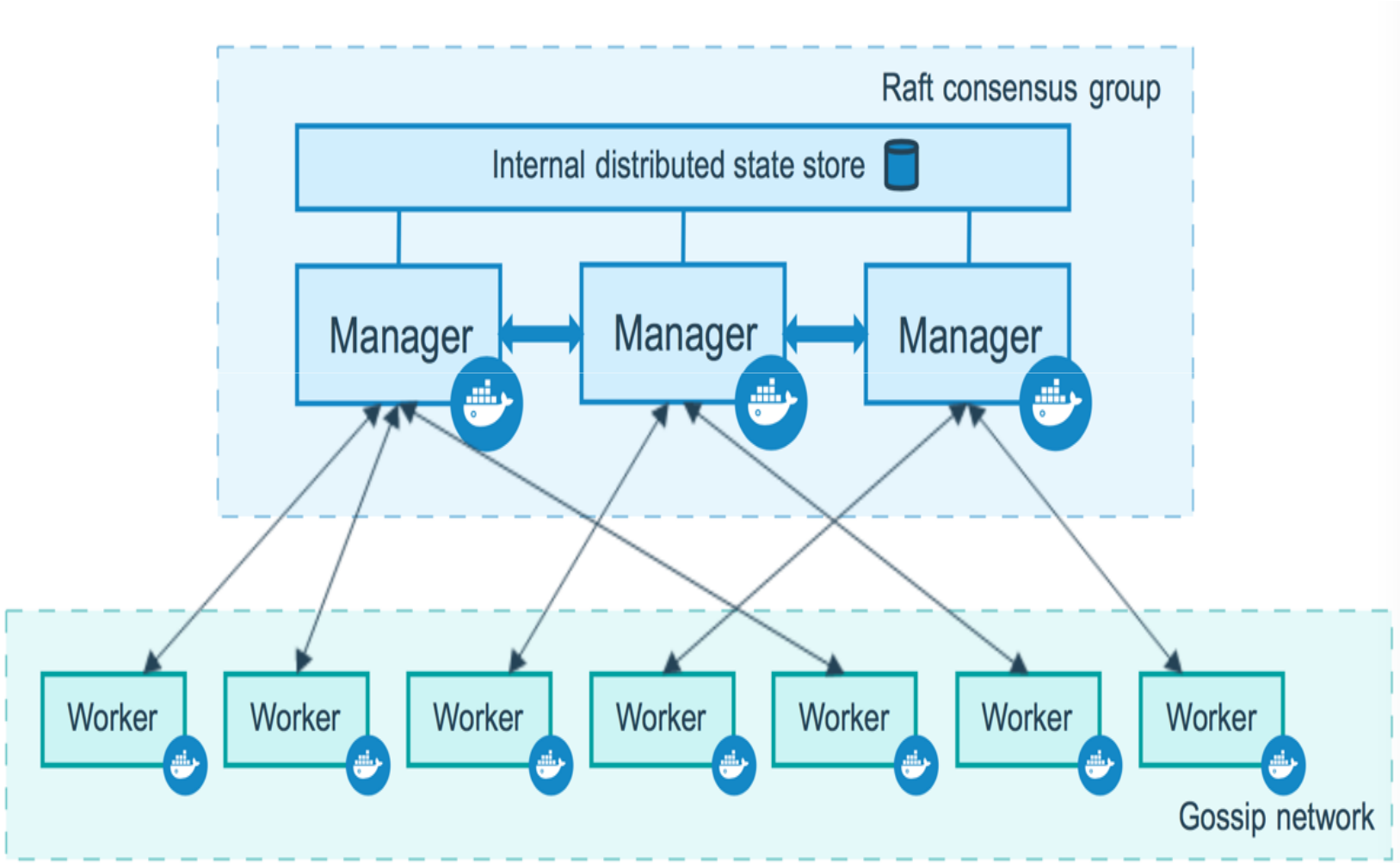
Docker Swarm Cluster

- 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 container.

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..

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.

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.

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.

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.

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.

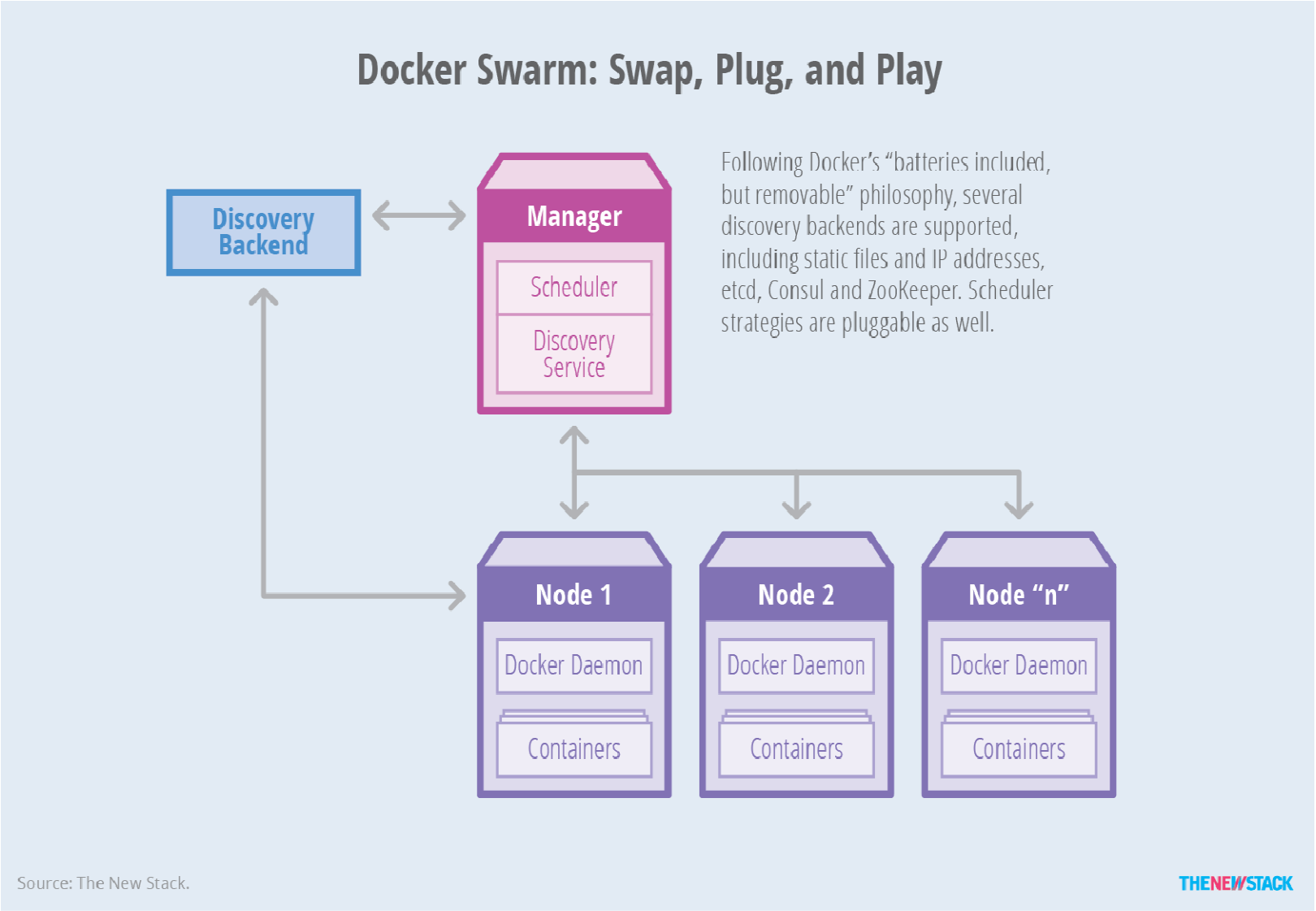
Health Monitoring

- **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



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.