FROM LXC TO DOCKER: Containers Get Portable

Hongchuan Li, Xuewei Zhang, Xiang Li

Virtualization on Linux

- Full Virtualization
  - Complete simulation of the actual hardware to allow guest OS to run unmodified
- Para-virtualization
  - A hardware environment is not simulated; however, the guest programs are executed in their own isolated domains, as if they are running on a separate system. Guest programs need to be specifically modified to run in this environment.
- OS – Level Virtualization
  - The same OS kernel is used to implement the "guest" environments. Applications running in a given "guest" environment view it as a stand-alone system.

Linux Containers

- LXC is often considered as something in the middle between a chroot on steroids and a full fledged virtual machine. The goal of LXC is to create an environment as close as possible as a standard Linux installation but without the need for a separate kernel.

Linux Containers

man lxc

```
...
    * General setup
      * Control Group support
        -> Namespace cgroup subsystem
        -> Freezer cgroup subsystem
        -> Cpuset support
        -> Simple CPU accounting cgroup subsystem
        -> Resource counters
          -> Memory resource controllers for Control Gro
      * Group CPU scheduler
        -> Basis for grouping tasks (Control Groups)
      * Namespaces support
        -> UTS namespace
        -> IPC namespace
        -> User namespace
        -> Pid namespace
        -> Network namespace
    * Device Drivers
      * Character devices
        -> Support multiple instances of devpts
      * Network device support
        -> MAC-VLAN support
        -> Virtual ethernet pair device
    * Networking
      * Networking options
        -> 802.1d Ethernet Bridging
    * Security options
      -> File POSIX Capabilities
...
```

cgroups

- Cgroups are organized hierarchically, like processes, and child cgroups inherit some of the attributes of their parents. However, there are differences between the two models.
- Cgroups are similar to processes in that:
- they are hierarchical, and
- child cgroups inherit certain attributes from their parent cgroup.
- The fundamental difference is that many different hierarchies of cgroups can exist simultaneously on a system. If the Linux process model is a single tree of processes, then the cgroup model is one or more separate, unconnected trees of tasks (i.e. processes).
- Multiple separate hierarchies of cgroups are necessary because each hierarchy is attached to one or more subsystems. A subsystem represents a single resource, such as CPU time or memory. Red Hat Enterprise Linux 6 provides ten cgroup subsystems, listed below by name and function.

Cgroups: members

- **Config_cgroups** adds support for grouping sets of processes together, for use with process control subsystems such as Cpusets, CFS, memory controls or device isolation.
- **Config_cgroups** provides a simple namespace cgroup subsystem to provide hierarchical naming of sets of namespaces, for instance virtual servers and checkpoint/restart jobs.
- **Config_cgroup_freezer** provides a way to freeze and unfreeze all tasks in a cgroup.

Cgroups: members

- **Config_cpusets** will let you create and manage CPUSETs which allow dynamically partitioning a system into sets of CPUs and Memory Nodes and assigning tasks to run only within those sets. This is primarily useful on large SMP or NUMA systems.
- **Config_cgroup_cpuacct** provides a simple Resource Controller for monitoring the total CPU consumed by the tasks in a cgroup.
- **Config_resource_counters** enables controller independent resource accounting infrastructure that works with cgroups.

Cgroups: members

- **Config_memcg** provides a memory resource controller that manages both anonymous memory and page cache. (See Documentation/cgroups/memory.txt) This option increases fixed memory overhead associated with each page of memory in the system. By this, 8(16)bytes/PAGE_SIZE on 32(64)bit system will be occupied by memory usage tracking struct at boot. Total amount of this is printed out at boot.
- **Config_cgroup_sched** lets CPU scheduler recognize task groups and control CPU bandwidth allocation to such task groups. It uses cgroups to group tasks.

Cgroups: members

- **Config_devpts_multiple_instances** enables support for multiple instances of devpts filesystem.
- **Config_macvlan** allows one to create virtual interfaces that map packets to or from specific MAC addresses to a particular interface.
- **Config_veth** virtual Ethernet pair device (local Ethernet tunnel)
- **Config_bridge** 802.1d Ethernet bridging
- **Config_security_file_capabilities** enables filesystem capabilities, allowing you to give binaries a subset of root's powers without using setuid 0.

Cgroups namespace supports

- **Config_namespaces** provides the way to make tasks work with different objects using the same id. For example same IPC id may refer to different objects or same user id or pid may refer to different tasks when used in different namespaces.
- config_uts_ns
- Config_ipc_ns
- Config_user_ns
- Config_pid_ns
- Config_net_ns

Use of LXC

- Steam
- LXC Web Panel
- LXC provider for Vagrant
- Docker

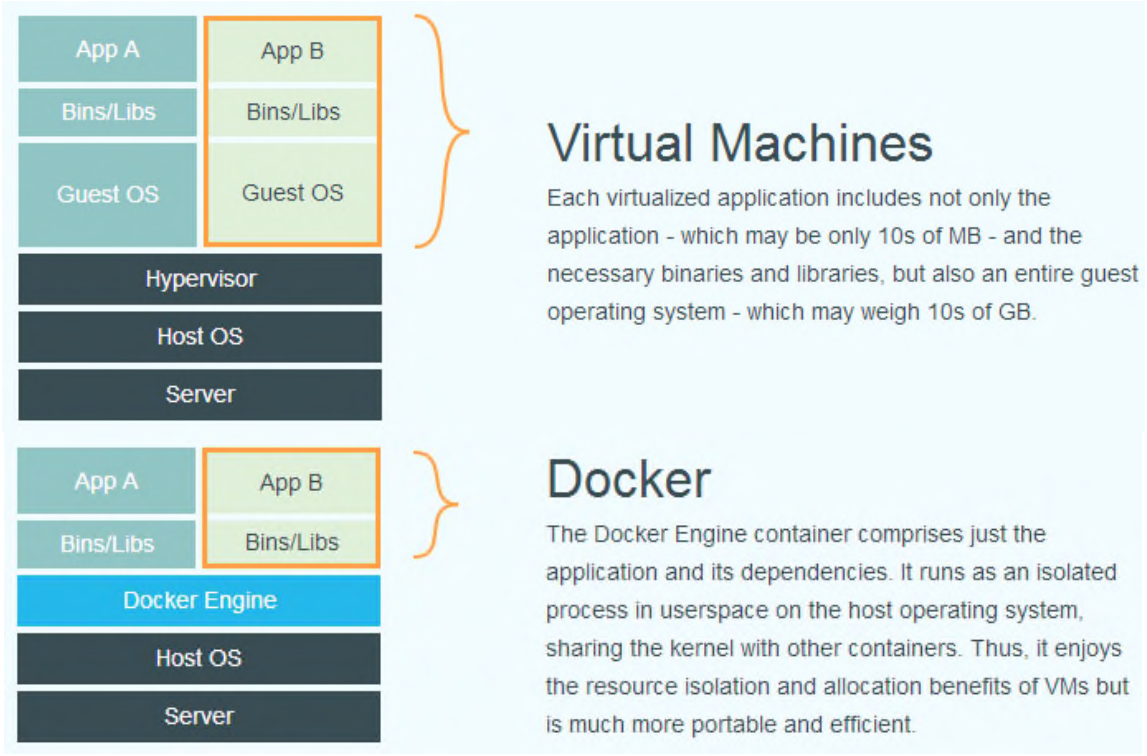
Docker

- Docker containers are both hardware-agnostic and platform-agnostic. This means they can run anywhere, from your laptop to the largest cloud compute instance and everything in between - and they don't require you to use a particular language, framework or packaging system. That makes them great building blocks for deploying and scaling web apps, databases, and backend services without depending on a particular stack or provider.

Container vs VM

	Virtual Machine (KVM, VMware)	Container (LXC, OpenVZ)
Hardware	Simulated	Uses it (almost) directly
Supported OS'es	Almost any	Only Linux
Space	User space	Kernel space
Separation	Full	Control Groups (cgroups)
Startup time	Seconds to minutes	Miliseconds
Scalability	A few	Sky is the limit (thousands)
Custom kernel	Yes	No
Enterprise features (live migration, etc)	Yes	No
Ease of creation	Moderate	Easy
Time consumption of creation	High	Low
Size	HUGE	Small

Docker vs VM


App A
App B
Bins/Libs
Bins/Libs
Guest OS
Guest OS
Hypervisor
Host OS
Server
Virtual Machines
Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.
App A
App B
Bins/Libs
Bins/Libs
Docker Engine
Host OS
Server
Docker
The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

What Docker Can Do

- Automating the packaging and deployment of applications
- Creation of lightweight, private PAAS environments
- Automated testing and continuous integration/deployment
- Deploying and scaling web apps, databases and backend services

Docker Basics

- Image - Used for create Docker containers
- Container
- Registry – Used for downloading images

UnionFS

- UnionFS is a filesystem service for Linux, FreeBSD and NetBSD which implements a union mount for other file systems. It allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem.
- Docker uses AUFS (AnotherUnionFS)
- AUFS support readonly/readwrite/writeout-able authority for every member repository

Dockerfile

- Docker can build images automatically by reading the instructions from a dockerfile.
- It has four parts:
  - Image
  - Maintainer
  - Command for image update
  - Command on container startup

Dockerfile

- Example of a typical dockerfile

```
# This dockerfile uses the ubuntu image
# VERSION 2 - EDITION 1
# Author: docker_user
# Command format: Instruction [arguments / command] ..


FROM ubuntu

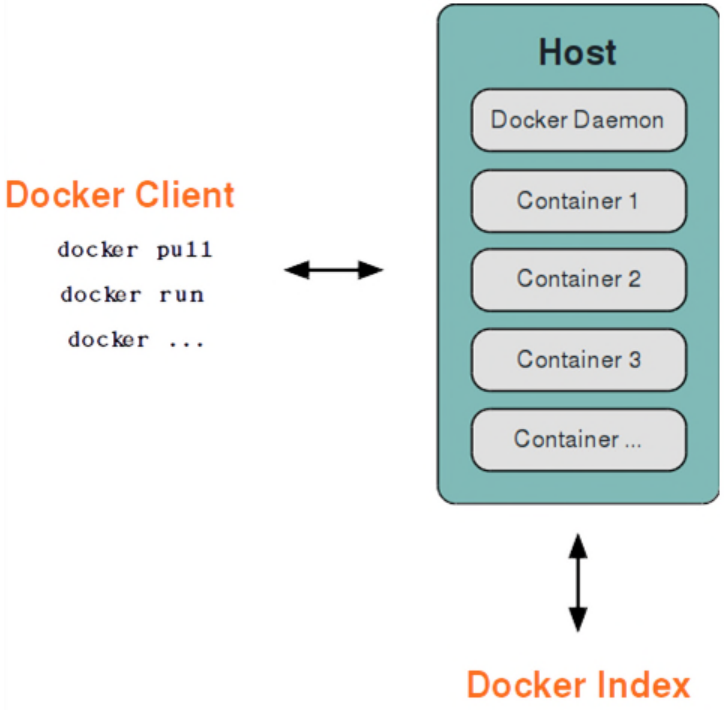
MAINTAINER docker_user docker_user@email.com

RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >> /etc/apt/sources.list
RUN apt-get update && apt-get install -y nginx
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf

CMD /usr/sbin/nginx
```

Docker Architecture

- Client and Server
- Docker daemon takes requests from client and processes (build, run, and distribute containers).
- Client and server can run on the same machine or communicate using socket or RESTful API



Docker Architecture

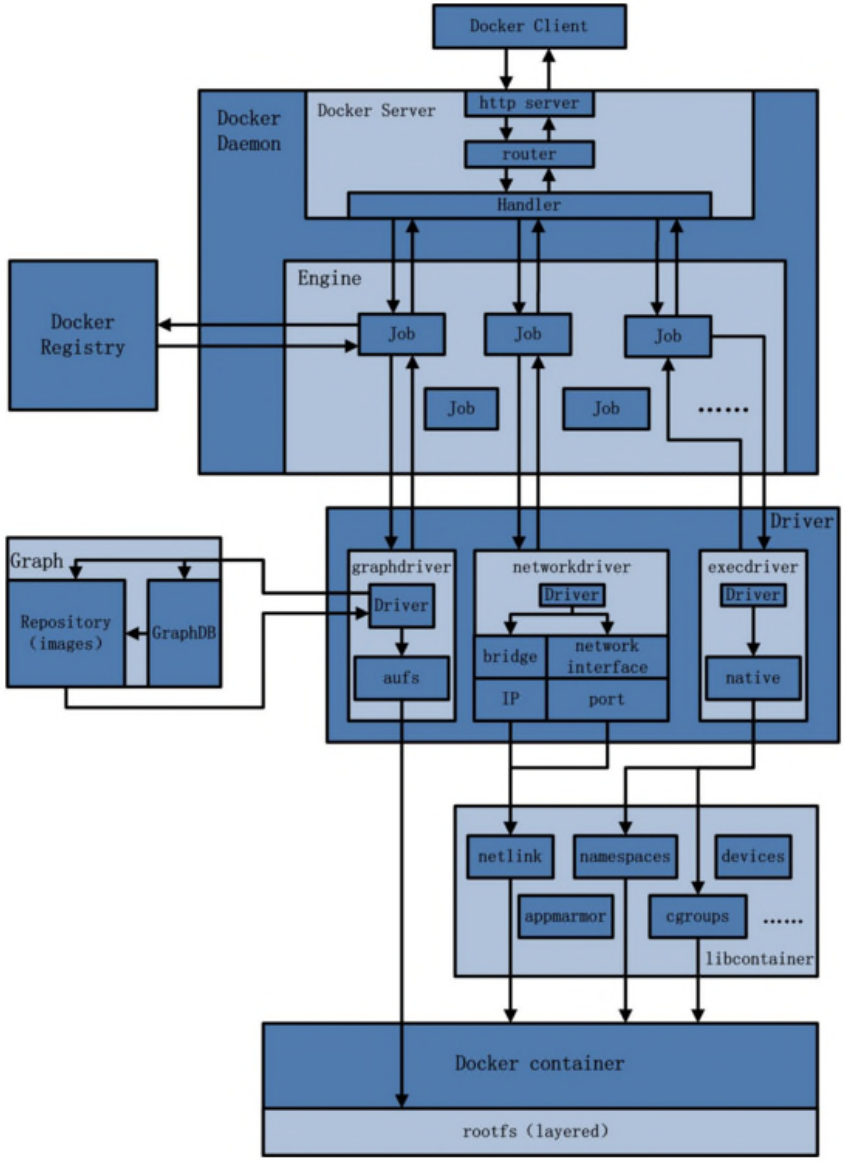
Docker Daemon, as the major part of the Docker Architecture, will work as a server so that it can admit the requests from clients.

The Engine will actually process the jobs.

As job running, when we need Images, we will download Images from Registry and the graphdriver will save the images as Graph

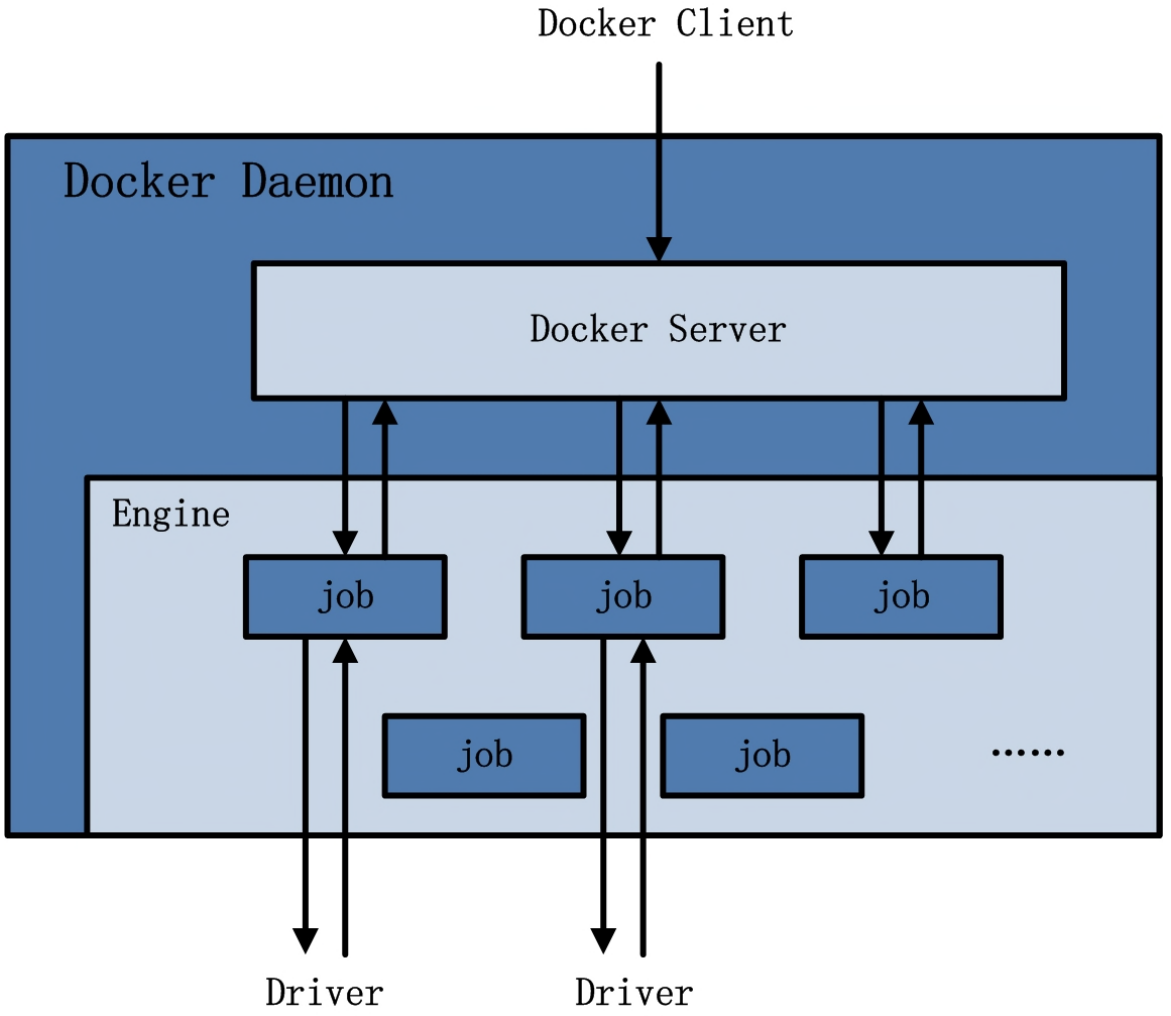
When we need to create network environment, we will use networkdriver to configure;

We use execdriver to control the resource allocation or execute a specific user command.



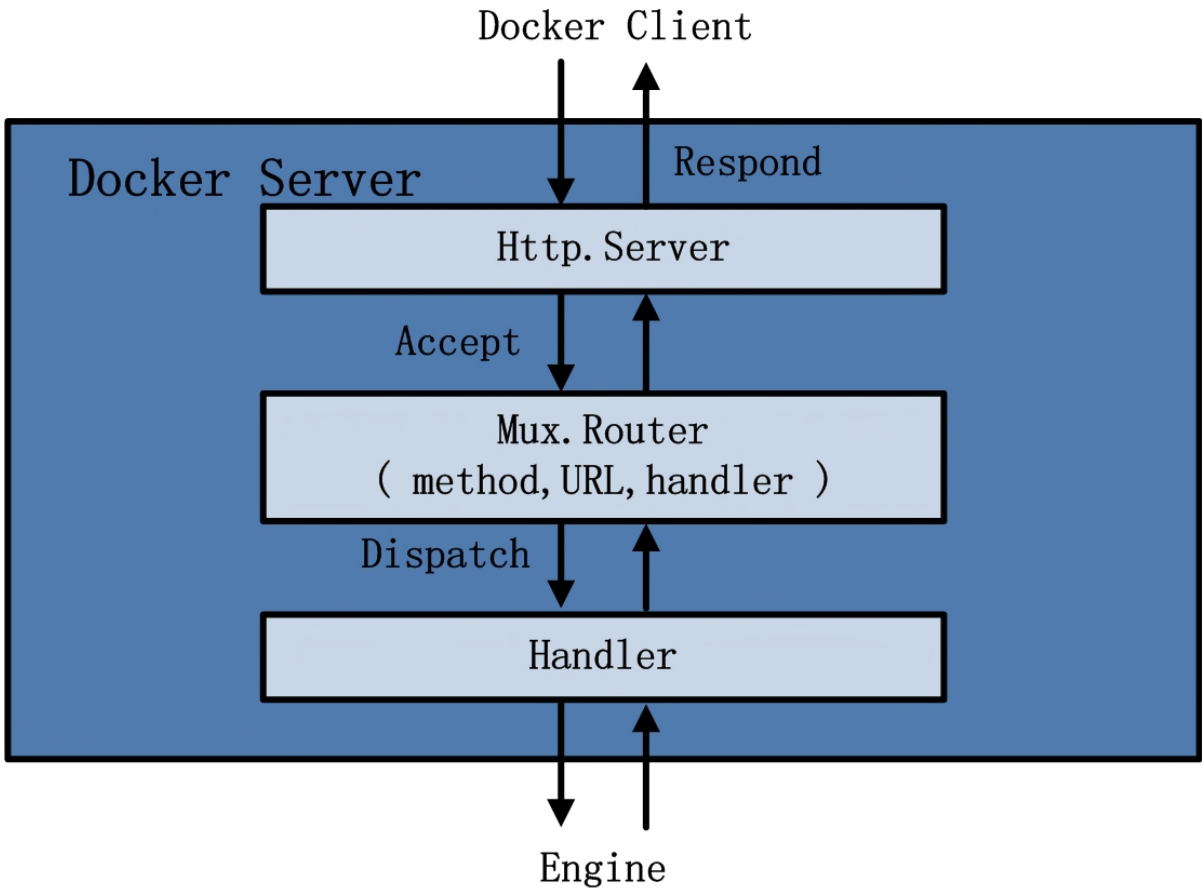
Docker Daemon

- A system daemon process, accept and begin process the request from clients
- When the request arrives, it asks router to schedule jobs, getting the appropriate handler to process the request
- Server, Engine, Job



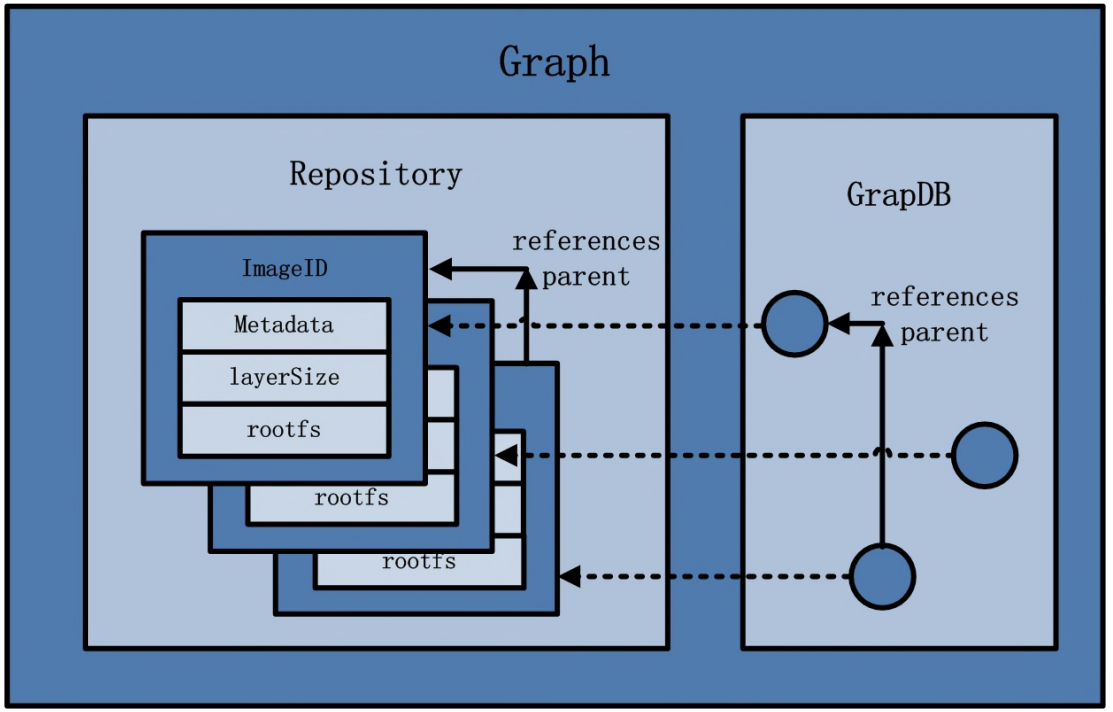
Docker Server

- Receive and schedule the requests from clients
- create mux.Router for routing
- Clients use HTTP to communicate with server
- Handler will process the request



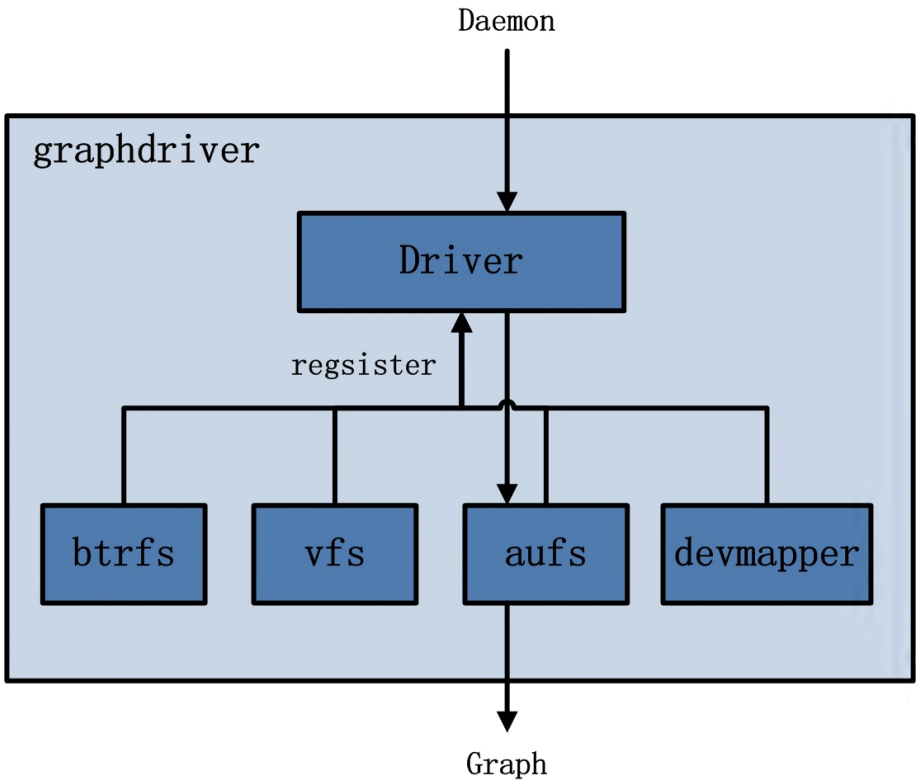
Graph

- Keeps the images and relationship among images
- GraphDB is based on SQLite



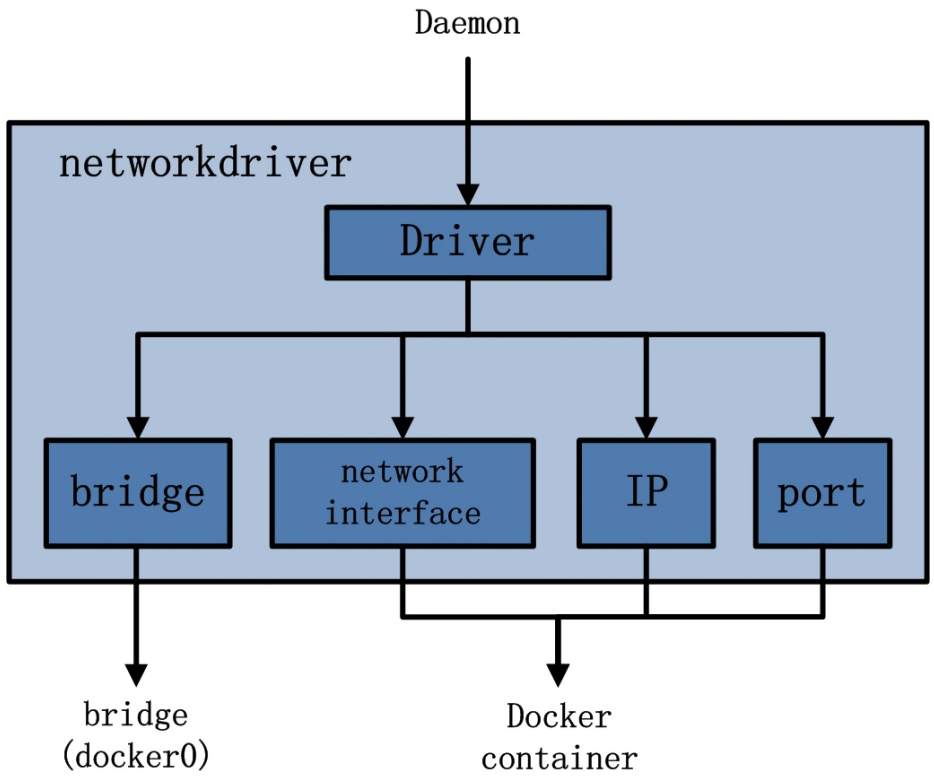
Graphdriver

- The driver for graph, used for managing the images



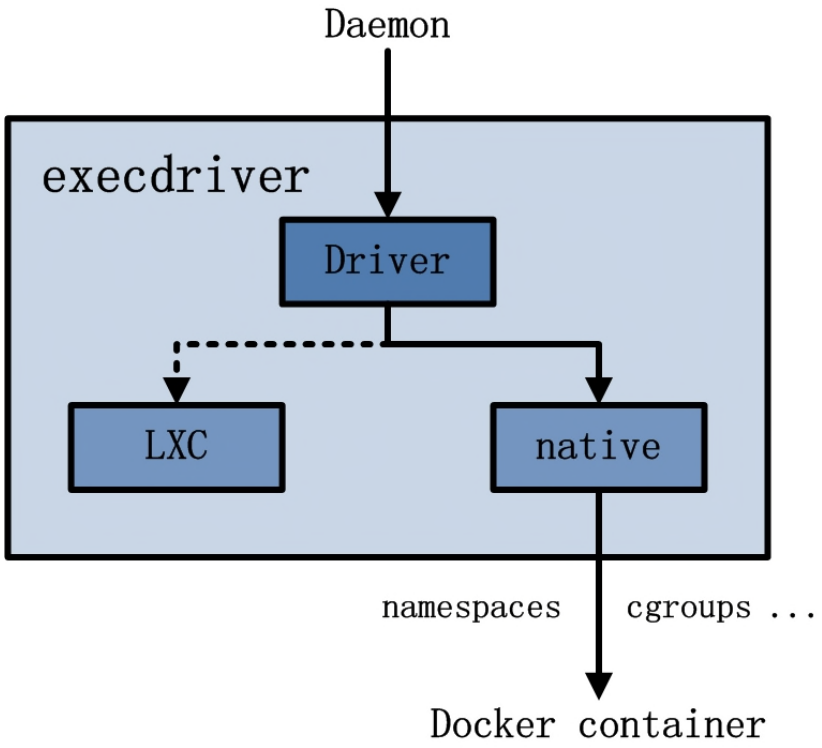
Networkdriver

- Configure the network, such as creating bridges and virtual network card, assigning IP for containers, etc.



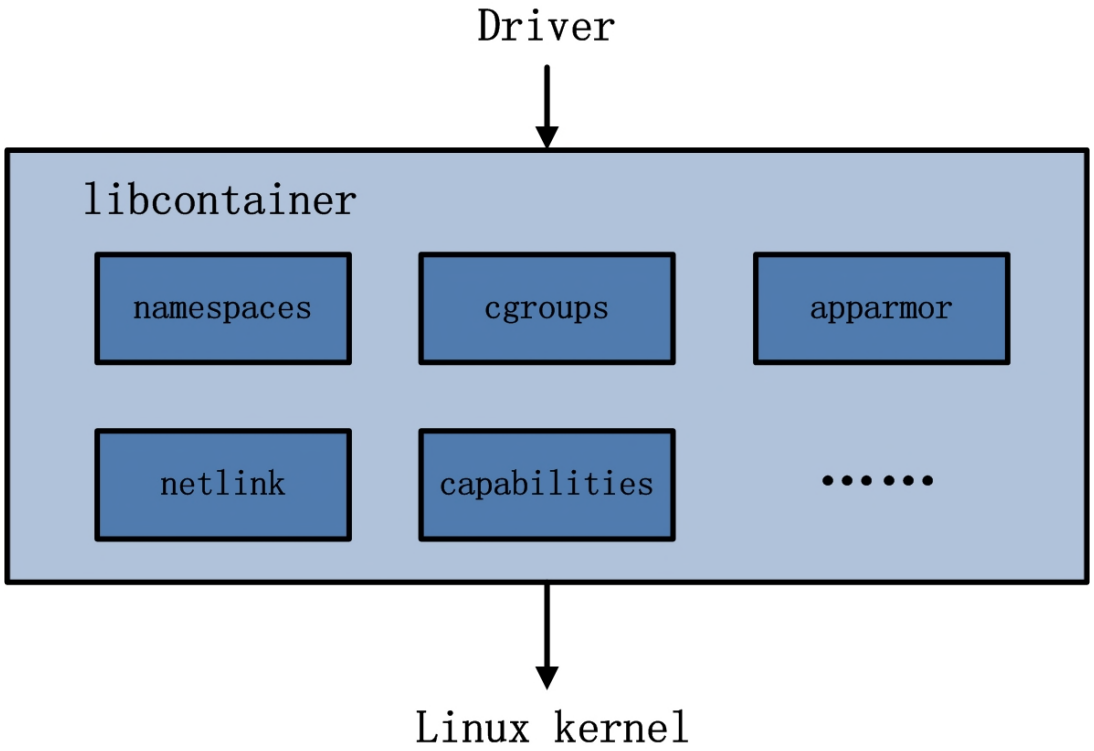
Execdriver

- Creates namespace
- Monitors usage of resources
- Executes processes

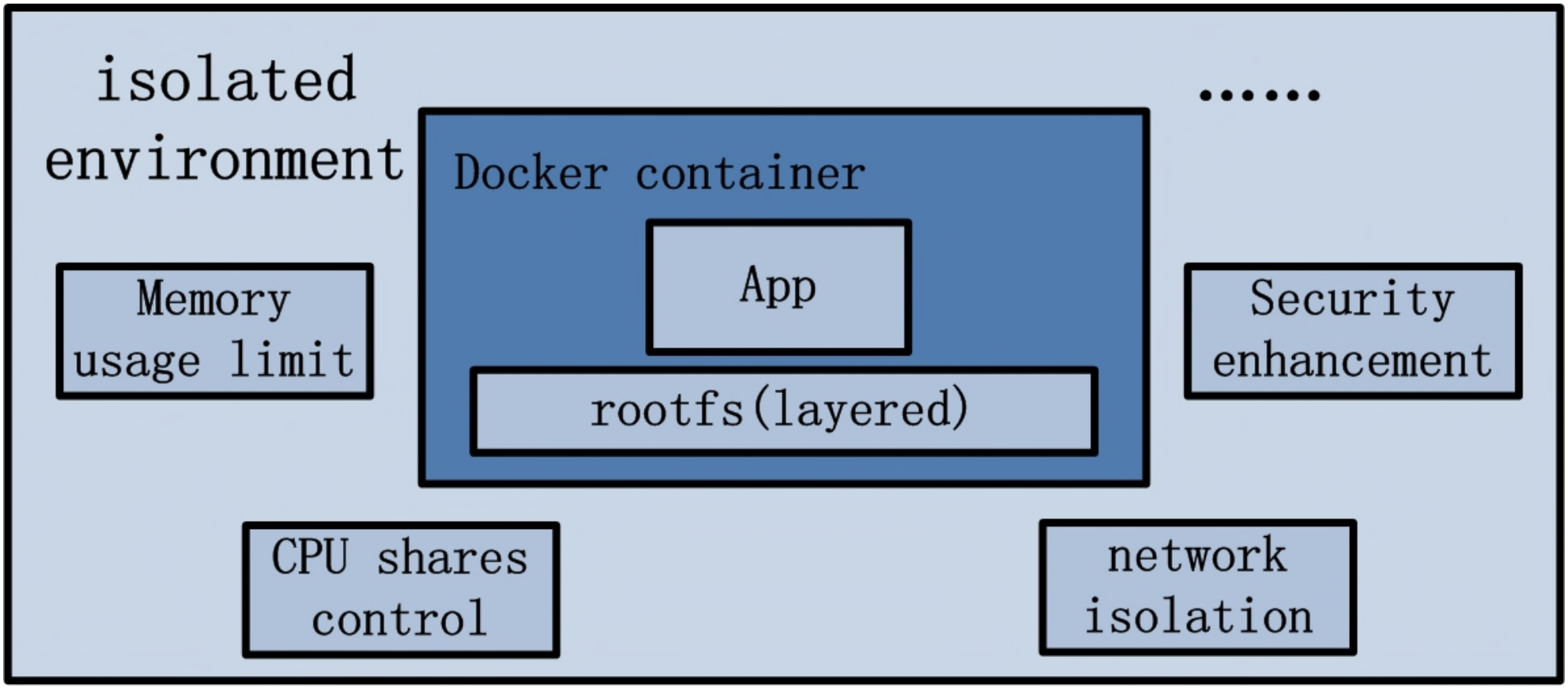


Libdriver

- Provides APIs for high-end needs of manage containers
- Therefore users do not need to refer LXC
- The APIs are great for other applications to use Docker



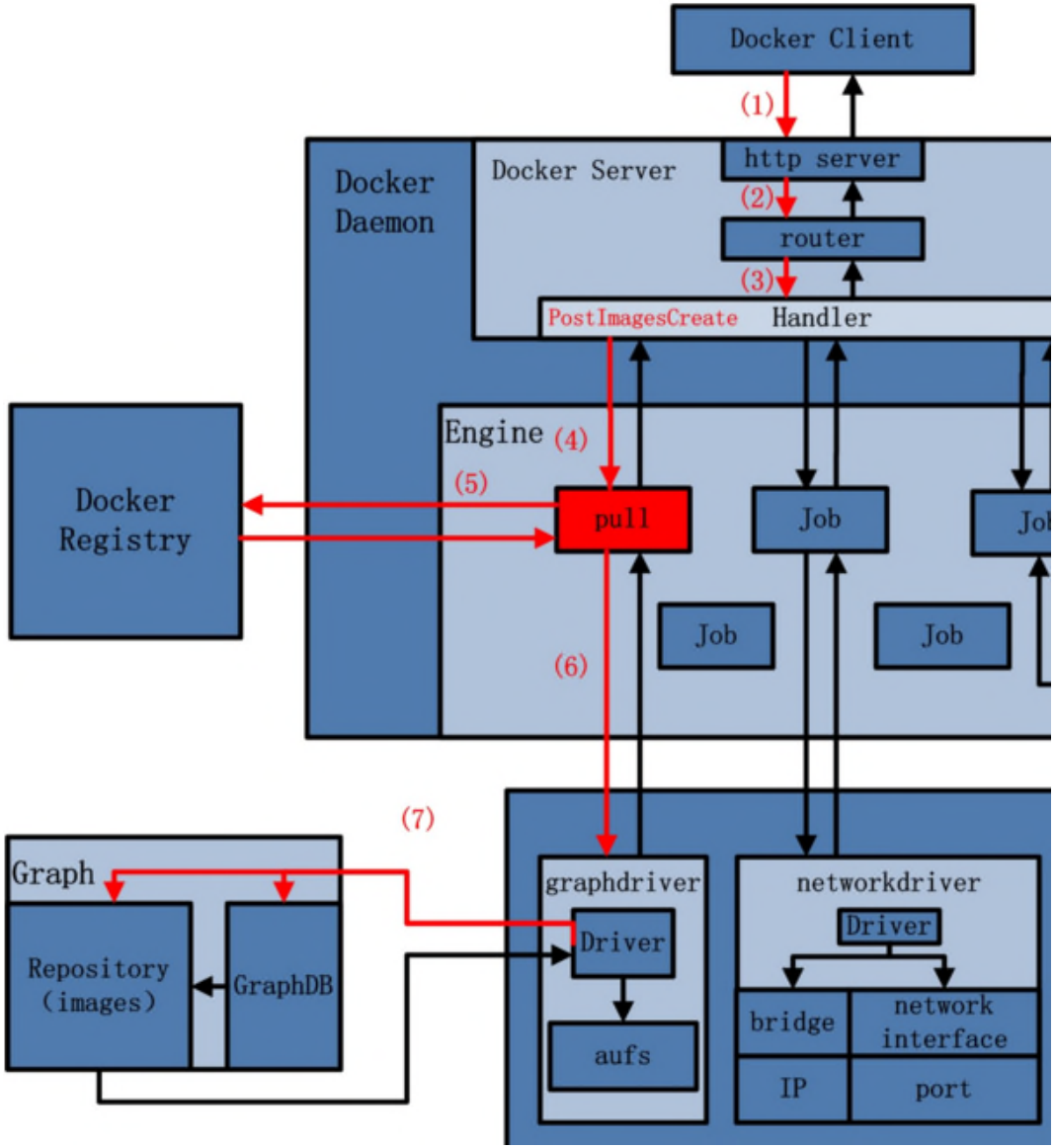
Docker Container



Example: Docker pull

- Download the specified image from registry and save it into local graph, for future use.

1. Client sends HTTP request to server (post)
2. Server accept and sends to mux.router
3. Mux.router sends to the needed handler (in this case, PostImageCreate)
4. PostImageCreate creates and executes a pull job
5. The pull job downloads images
6. The pull job sends the image to graphdriver
7. Graphdriver create a graph and record image in GraphDB.


Docker Client
(1)
Docker Daemon
Docker Server
http server
(2)
router
(3)
PostImagesCreate Handler
Engine
(4)
(5)
Docker Registry
pull
Job
Job
Job
(6)
(7)
Graph
Repository (images)
GraphDB
graphdriver
Driver
aufs
networkdriver
Driver
bridge
network interface
IP
port

Example: Docker run

- Execute a command in a new Docker container
- What is going on:
- 1. Create a rootfs
- 2. Create the environment and execute

Example: Docker run

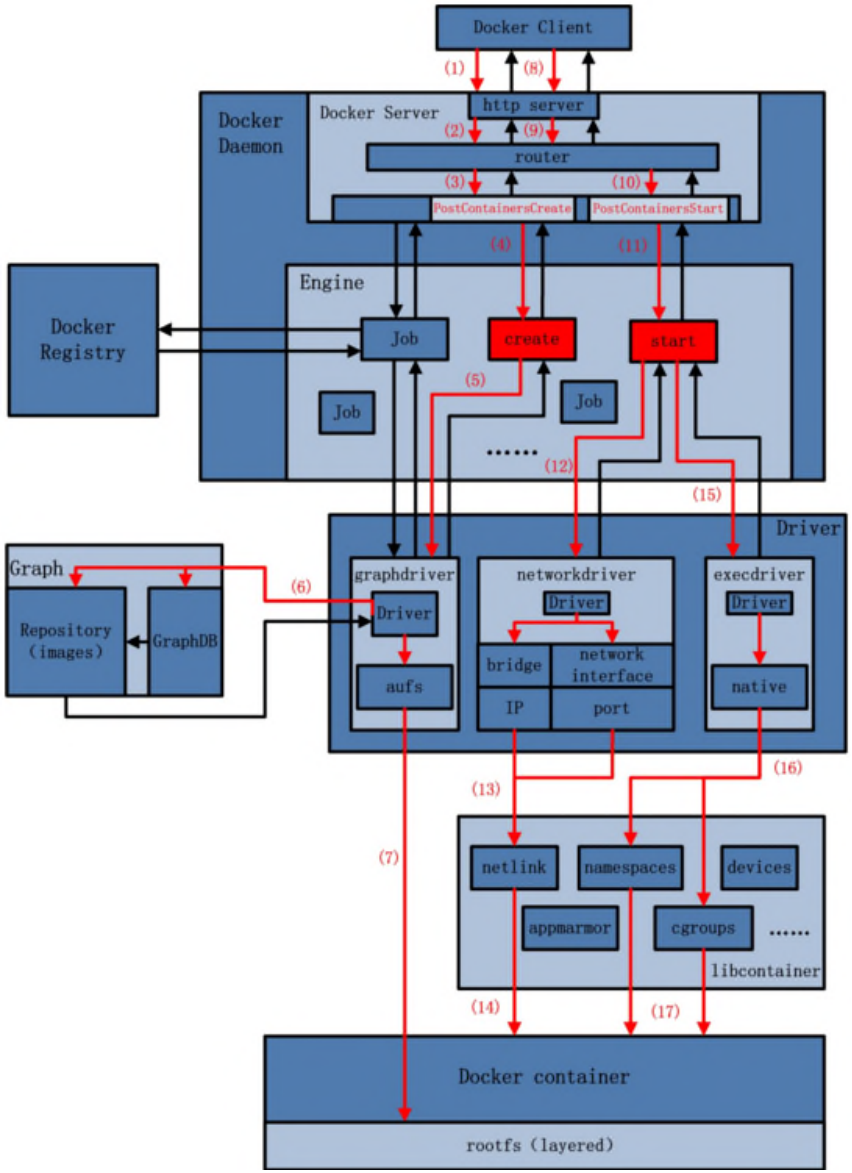
- 1. client receives run command, and sends a http post to server as “/containers/create?”+”xxx”
- 2. server accepts the request and sends to mux.router to get handler
- 3. mux.router specifies PostContainersCreate
- 4. PostContainersCreate creates “create” job and runs it
- 5. the “create” job does Container.create to communicate with graphdriver
- 6. graphdriver gets images needed by rootfs from Graph
- 7. graphdriver loads and setups all images of rootfs for the container

Example: Docker run

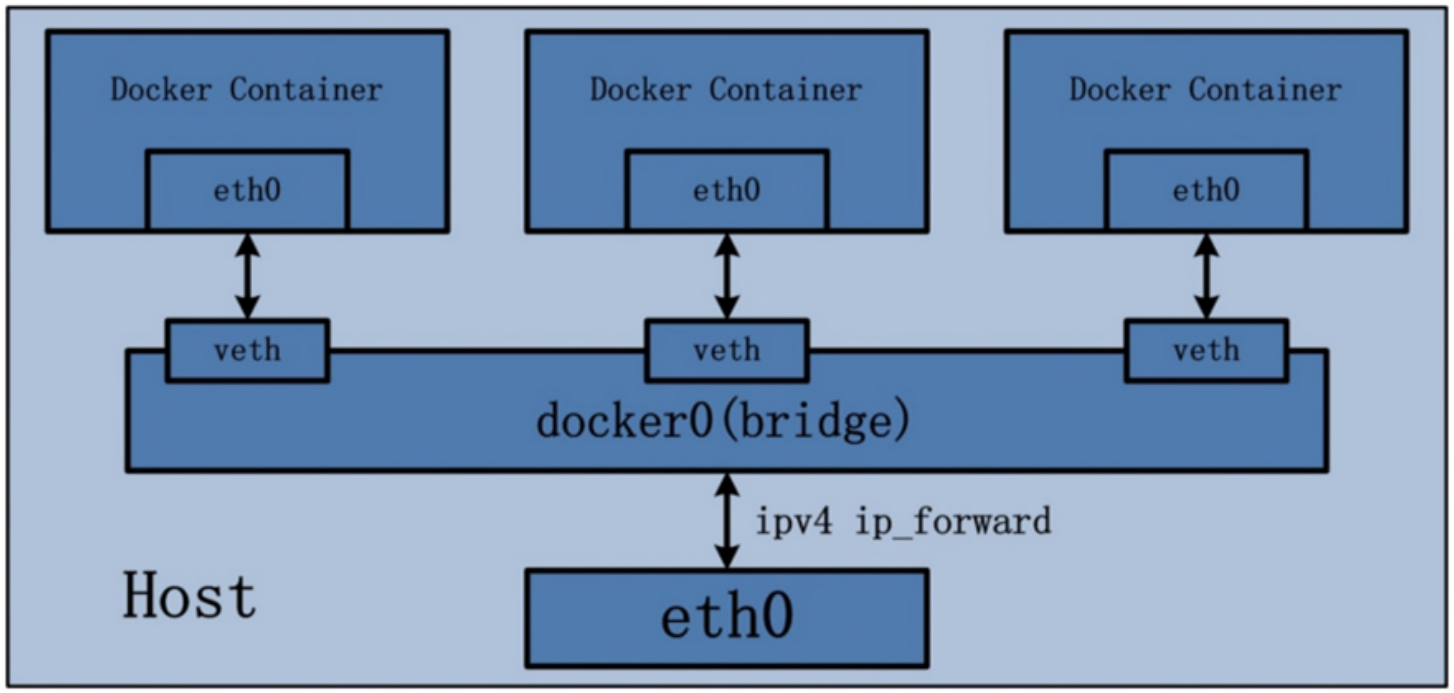
- 8. client receives the status from server, and send post to server as “/containers/”+container_ID+”/start”
- 9. server accepts the request and ask mux.router to get handler
- 10. mux.router gets the PostContainersStarter handler
- 11. PostContainerStarter creates “start” job and runs it
- 12. “start” job communicates with networkdriver
- 13. networkdriver assigns IP, port, firewall configuration with netlink package in libcontainer
- 14. netlink finishes the creation of network environment

Example: Docker run

- 15. “start” job begins to call execdriver
- 16. execdriver initializes the container’s run time environment such as namespace, using libcontainer
- 17. libcontainer finishes the initialization and execute the command from user


Docker Client
(1)
(8)
Docker Daemon
Docker Server
http server
(2)
(9)
router
(3)
(10)
PostContainersCreate
PostContainersStart
(4)
(11)
Engine
Docker Registry
Job
create
start
(5)
Job
Job
.....
(12)
(15)
Driver
Graph
(6)
graphdriver
networkdriver
execdriver
Repository (images)
GraphDB
Driver
Driver
Driver
aufs
bridge
network interface
IP
port
native
(16)
(13)
(7)
netlink
namespaces
devices
appmarmor
cgroups
.....
libcontainer
(14)
(17)
Docker container
rootfs (layered)

Docker Daemon Network



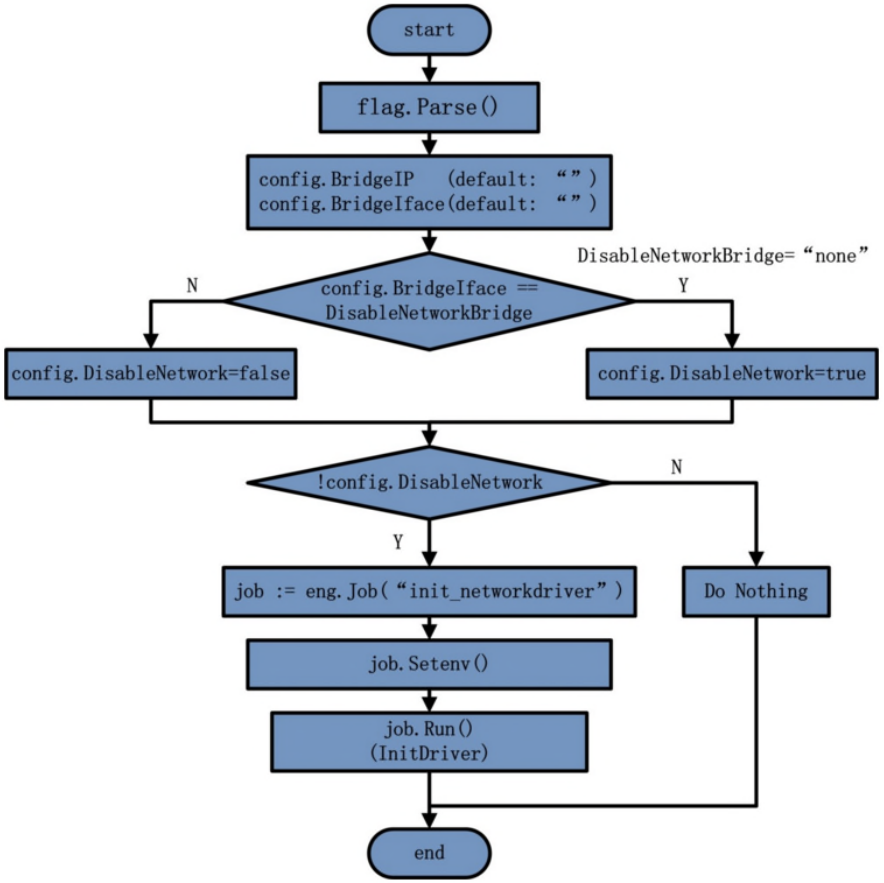
Docker Daemon Network

- Most people use bridging in Docker
- Configuration:
- flag.BoolVar(&**config.EnableIptables**, []string{"#iptables", "-iptables"}, true, "Enable Docker's addition of iptables rules")
- flag.BoolVar(&**config.EnableIpForward**, []string{"#ip-forward", "-ip-forward"}, true, "Enable net.ipv4.ip_forward")
- flag.StringVar(&**config.BridgeIP**, []string{"#bip", "-bip"}, "", "Use this CIDR notation address for the network bridge's IP, not compatible with -b")
- flag.StringVar(&**config.BridgeIface**, []string{"b", "-bridge"}, "", "Attach containers to a pre-existing network bridge\nuse 'none' to disable container networking")
- flag.BoolVar(&**config.InterContainerCommunication**, []string{"#icc", "-icc"}, true, "Enable inter-container communication")

Docker Daemon Network Flags

- EnableIptables: ensure that Docker has authority to add entries into ipdatbles
- EnableIpForward: ensure that net.ipv4.ip_forward can be usef for transmitting data among devices
- BridgeIP: configure network address when Daemon is booting
- BridgeIface: set bridge for Docker network; if is “none” then we cannot create bridge (i.e., no network)
- InterContainerCommunication: ensure the communication among containers
- DefaultBindingIP: specify the default IP address when binding ports of container

Docker Daemon Network Initialization

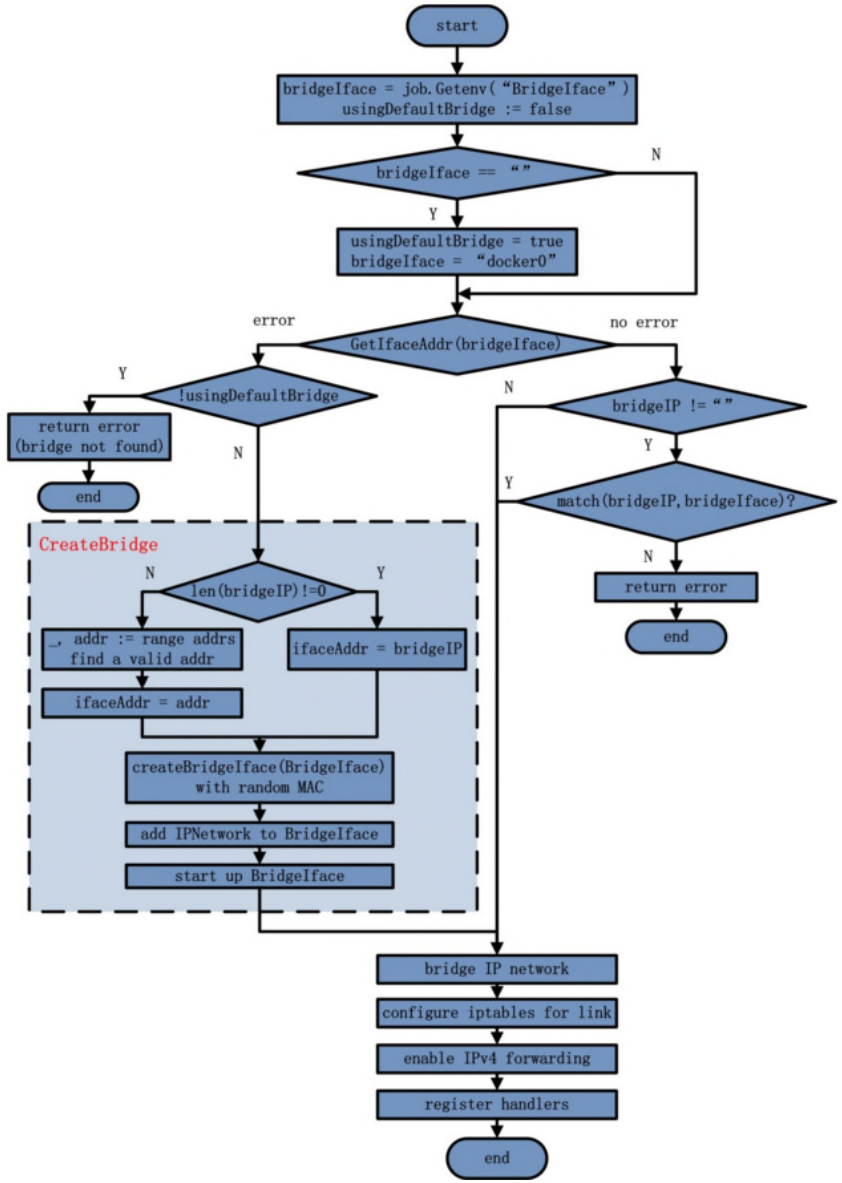


Docker Daemon Network Initialization

- Create Network:
- 1. Create the job “init_networkdriver”
- 2. Configure environment flags: EnableIptables、InterContainerCommunication、EnableIpForward、BridgeIface、BridgeIP, and DefaultBindingIP
- 3. Execute the job

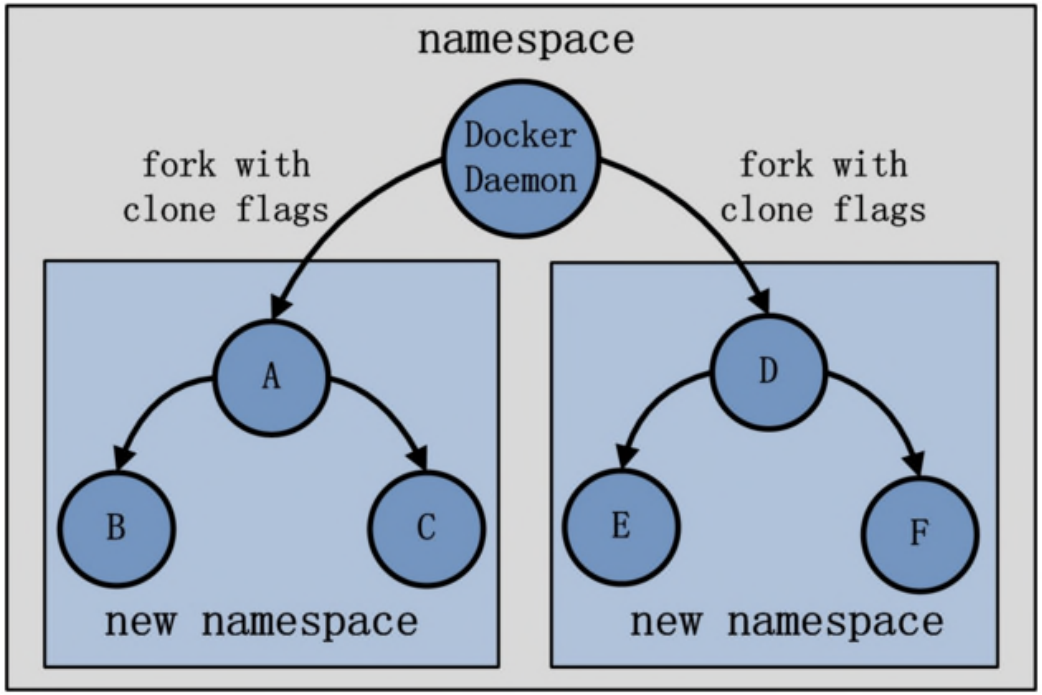
Create Bridge

- 1. Get environment variables (flags) from job init_networkdriver
- 2. Set the bridgeIface (bool usingDefaultBridge, default false)
- 3. Locate the bridgeIface (getIfaceAddr returns IP address or error)
- 4 (a) – No error
  - If no error, then the bridge exists
  - Verify the bridgeIP is the same as user enters
  - If no IP is specified by user, use the bridgeIP
- 4 (b) – Error
  - If Error, then we create new bridge
  - Two situations:
    - User specifies the bridgeIface, but it does not exist – return error
    - User does not specify, and docker0 is not created – create docker0 and return docker0's IP address
  - createBridge()
    - Get an IP address for docker0
    - Create docker0, and allocate a random MAC
    - Set the IP address for docker0
    - Start docker0 bridge



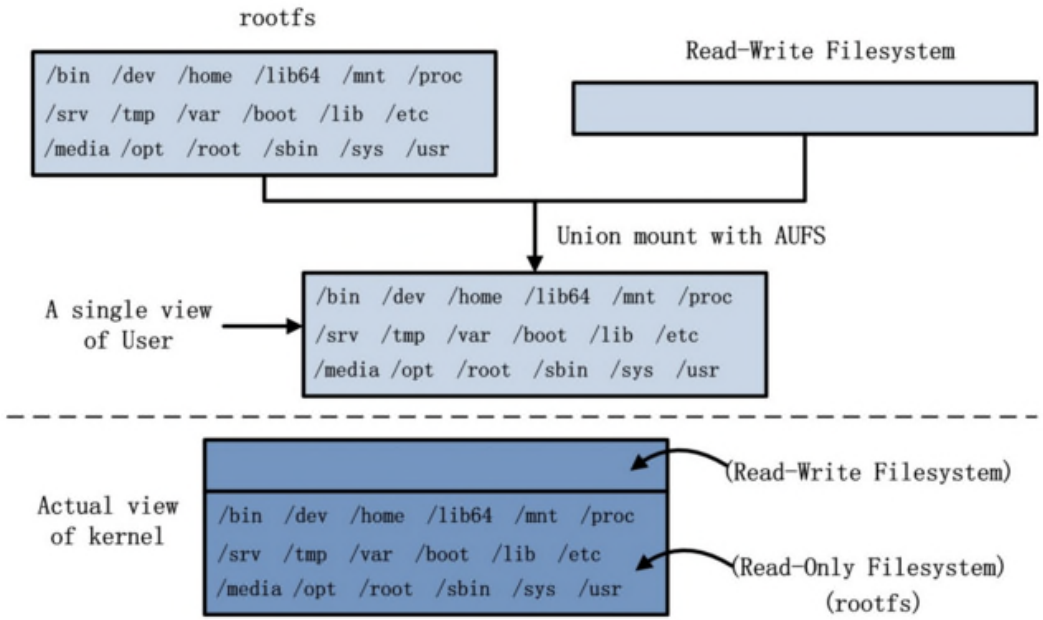
Docker Container Namespace

- Use cgroup
- When a container process is created, Docker Daemon can get the pid of that process and set the pid in the cgroup system for resource allocation
- We still need to configure manually the resources Docker Container needs for network



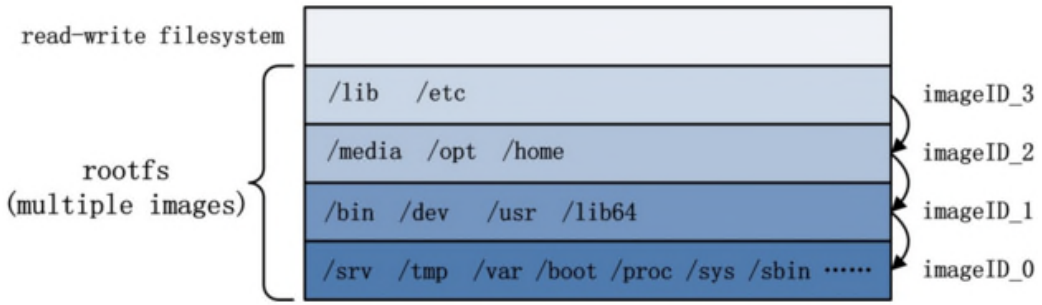
rootfs

- The root directory of a Docker Container
- Contains that container's system configuration, tool, and container files
- When Docker Container is mounting rootfs, it is set to be read-only
- The file system is not directly set to be read-write; instead, using AUFS, Docker mounts another read-write file system on to rootfs
- Example: an Ubuntu image



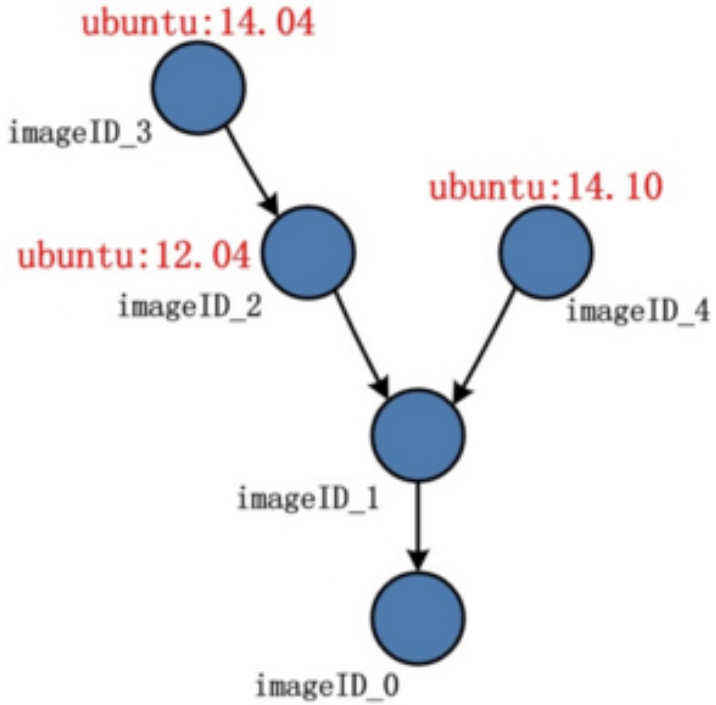
Docker Image

- A part of rootfs
- One rootfs can contain many images
- Example: 4-layer image



Docker Image

- Parent Image and Base Image
- Example:
- 5 images in left tree, 6 images in right tree
- imageID_0 is the parent of imageID_1, which is the parent of 2 and 4, etc.
- When downloading Ubuntu:14.04 and Ubuntu: 14.04, Docker will only download one copy of imageID_2, imageID_1 and imageID_0



Usage & Conclusion

- Build, Ship, Run for distributed systems
- IaaS
- Open Source, API-level

Citation & References

- LXC: https://fourdollars.github.io/lxc-intro/
- Docker Architecture: http://www.infoq.com/cn/articles/docker-source-code-analysis-part1
- Hongliang Sun's page about Docker: http://www.infoq.com/cn/author/%E5%AD%99%E5%AE%8F%E4%BA%AE