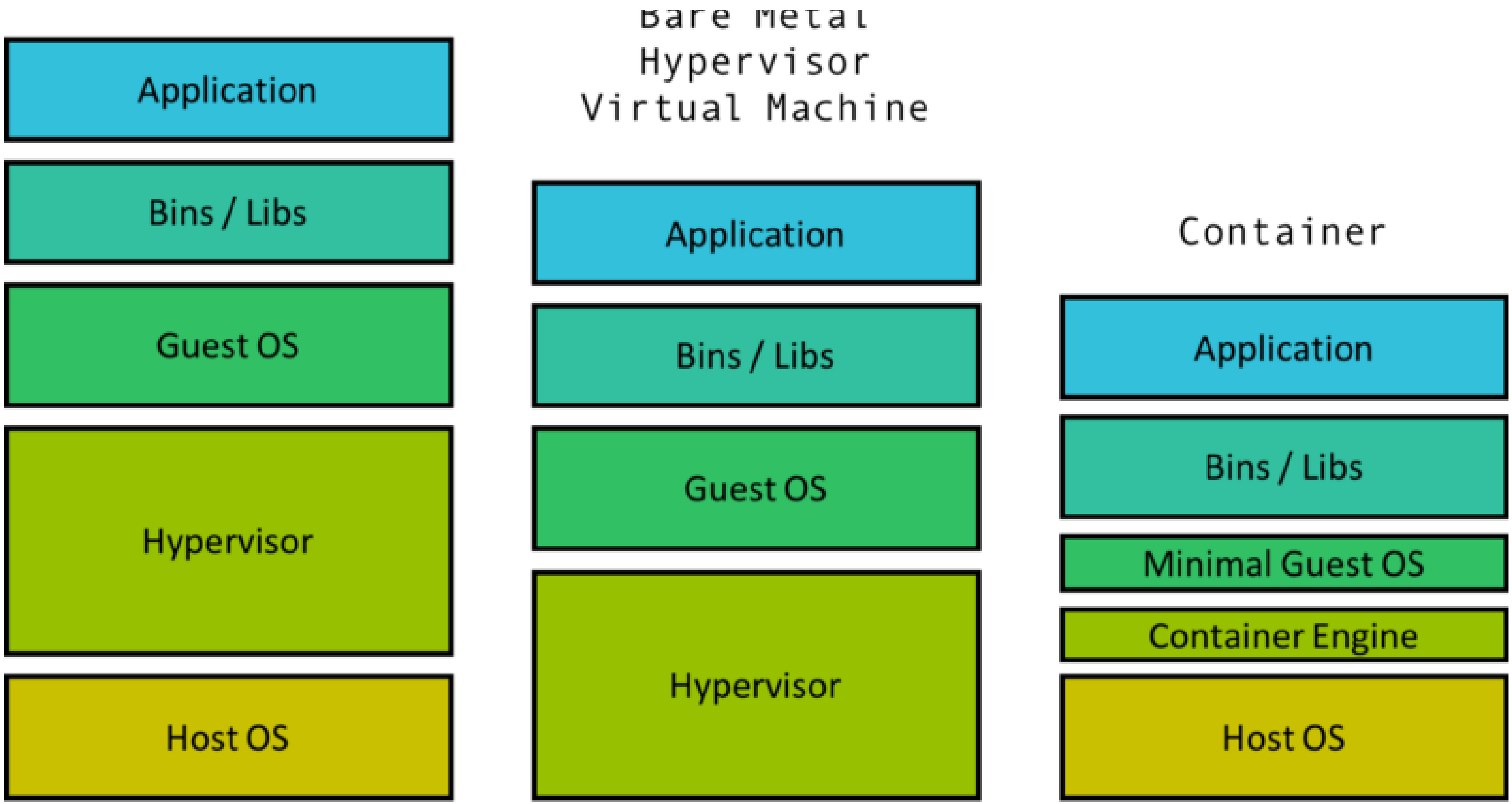
Forum Donate

Learn to code — free 3,000-hour curriculum

OCTOBER 26, 2018 / #DOCKER

Demystifying Containers 101: A Deep Dive Into Container Technology for Beginners


Bare Metal
Hypervisor
Virtual Machine
Container
Application
Bins / Libs
Guest OS
Hypervisor
Host OS
Application
Bins / Libs
Guest OS
Hypervisor
Application
Bins / Libs
Minimal Guest OS
Container Engine
Host OS

Introduction

Regardless of whether you are a student in school, a developer at some company, or a software enthusiast, chances are you heard of *containers*. You may have also heard that containers are lightweight virtual machines, but what does that really mean, how exactly do containers work, and why are they so important?

This story serves as a look into containers, their key great technical ideas, and the applications. I won't assume any prior knowledge in this field other than a basic understanding of computer science.

The Kernel and the OS

Your laptop, along with every other computer, is built on top of some pieces of hardware like the CPU, persistent storage (disk drive, SSD), memory, network card, etc.

To interact with this hardware, a piece of software in the operating system called the *kernel* serves as the bridge between the hardware and the rest of the system. The kernel is responsible for as the bridge between the hardware and the rest of the system. The kernel is responsible for scheduling *processes* (programs) to run, managing devices (reading and writing addresses on disk and memory), and more.

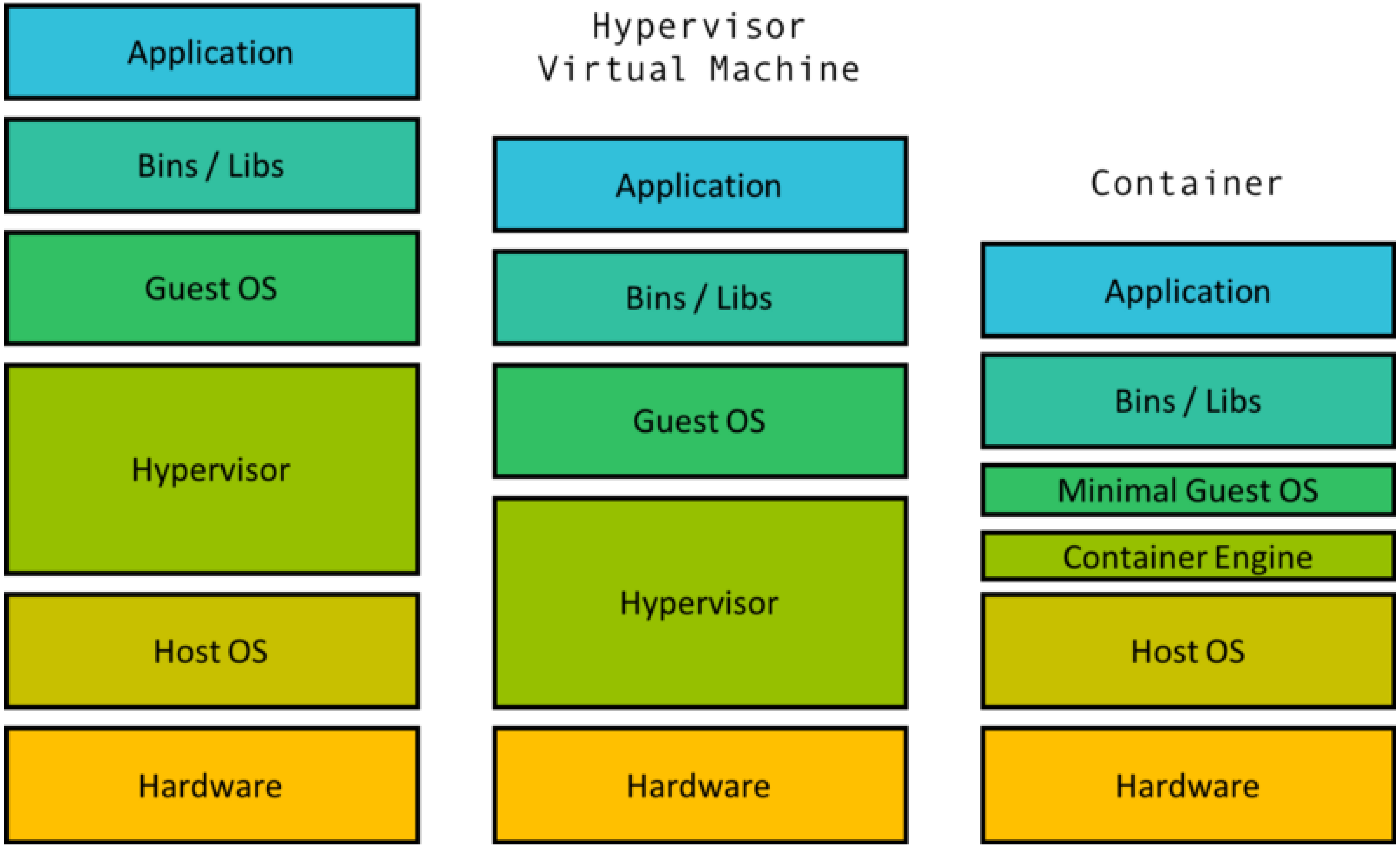
The Virtual Machine

So you have a computer that runs MacOS and an application that is built to run on Ubuntu. Hmmm… One common solution is to boot up a virtual machine on your MacOS computer that runs Ubuntu and then run your program there.

A *virtual machine* is comprised of some level of hardware and kernel virtualization on which runs a guest operating system. A piece of software called a *hypervisor* creates the virtualized hardware which may include the virtual disk, virtual network interface, virtual CPU, and more. Virtual machines also include a guest kernel that can talk to this virtual hardware.

The hypervisor can be hosted, which means it is some software that runs on the Host OS (MacOS) as in the example. It can also be bare metal, running directly on the machine hardware (replacing your OS). Either way, the hypervisor approach is considered heavy weight as it requires virtualizing multiple parts if not all of the hardware and kernel.

When there needs to be multiple isolated groups on the same machine, running a VM for each of these groups is way too heavy and wasteful of resources to be a good approach.



Overhead not to scale.

VMs require hardware virtualization for machine level isolation whereas containers operate on isolation within the same operation system. The overhead difference becomes really apparent as the number of isolated spaces increase. A regular laptop can run tens of containers but can struggle to run even one VM well.

In 2006, engineers at Google invented the Linux “control groups”, abbreviated as *cgroups*. This is a feature of the Linux kernel that isolates and controls the resource usage for user processes.

These processes can be put into *namespaces*, essentially collections of processes that share the same resource limitations. A computer can have multiple namespaces, each with the resource properties enforced by the kernel.

The resource allocation per namespace can be managed in order to limit the amount of the overall CPU, RAM, etc that a set of processes can use. For example, a background log aggregation application will probably need to have its resources limit in order to not accidentally overwhelm the actual server it’s logging.

While not an original feature, cgroups in Linux were eventually reworked to include a feature called *namespace isolation*. The idea of namespace isolation itself is not new, and Linux already had many kinds of namespace isolation. One common example is process isolation, which separates each individual process and prevents such things like shared memory.

Cgroup isolation is a higher level of isolation that makes sure processes within a cgroup namespace are independent of processes in other namespaces. A few important namespace isolation features are outlined below and pave the foundation for the isolation we expect from containers.

- Network Namespaces: Isolation of the network interface controller, iptables, routing tables, and other lower level networking tools.
- Mount Namespaces: Filesystems are mounted, so that the file system scope of a namespace is limited to only the directories mounted.
- User Namespaces: Limits users within a namespace to only that namespace and avoids user ID conflicts across namespaces.

To put it simply, each namespace would appear to be its own machine to the processes within it.

Linux Containers

Linux cgroups paved the way for a technology called *linux containers* (LXC). LXC was really the first major implementation of what we know today to be a container, taking advantage of cgroups and namespace isolation to create virtual environment with separate process and networking space.

In a sense, this allows for independent and isolated *user spaces*. The idea of *containers* follows directly from LXC. In fact, earlier versions of Docker were built directly on top of LXC.

Docker

Docker is the most widely used container technology and really what most people mean when they

by the Linux kernel and recently Windows as well.

Image source: Docker

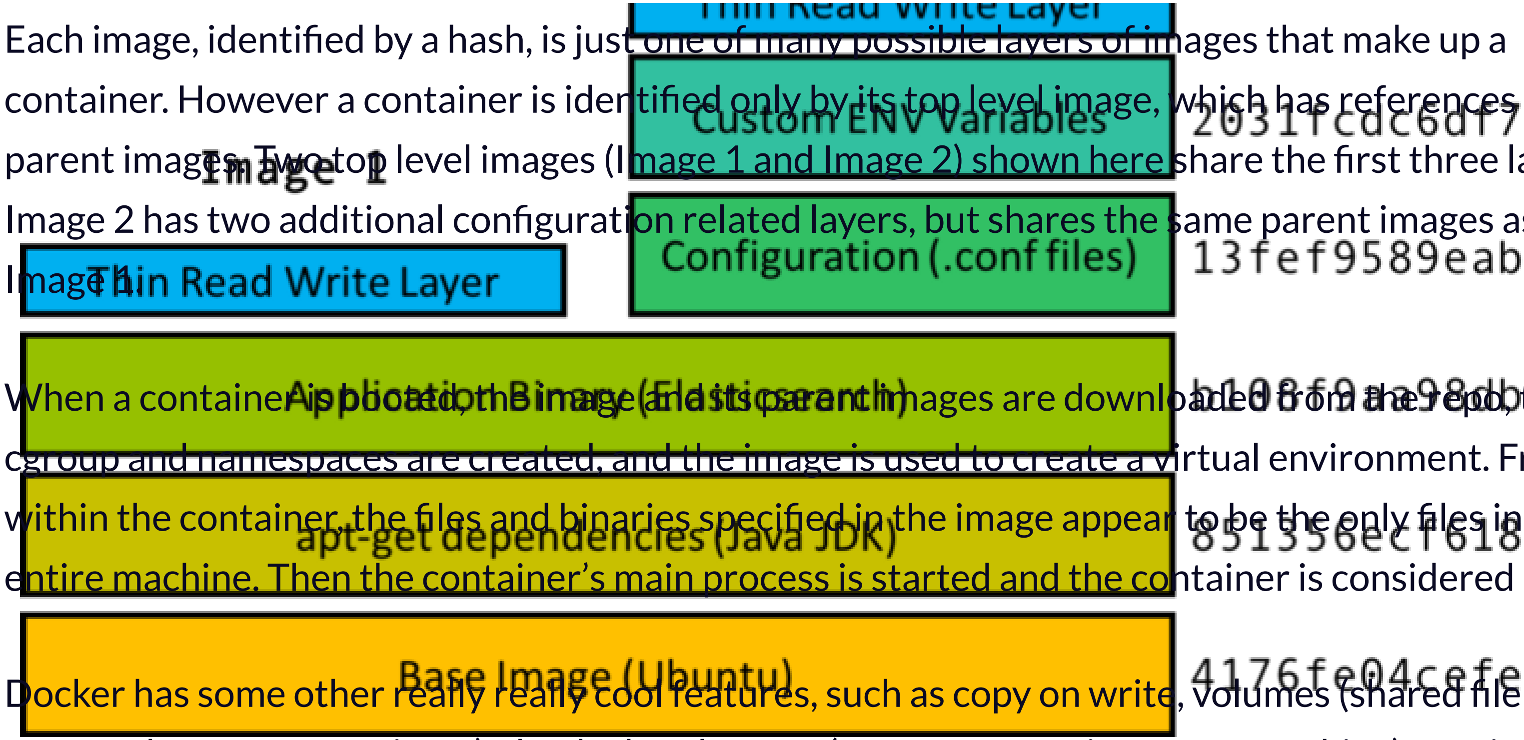
A Docker container is made up of layers of *images*, binaries packed together into a single package. The base image contains the operating system of the container, which can be different from the OS of the host.

The OS of the container is in the form an image. This is not the full operating system as on the host, and the difference is that the image is just the file system and binaries for the OS while the full OS

top of that may be the image that contains the application binary, and so on.

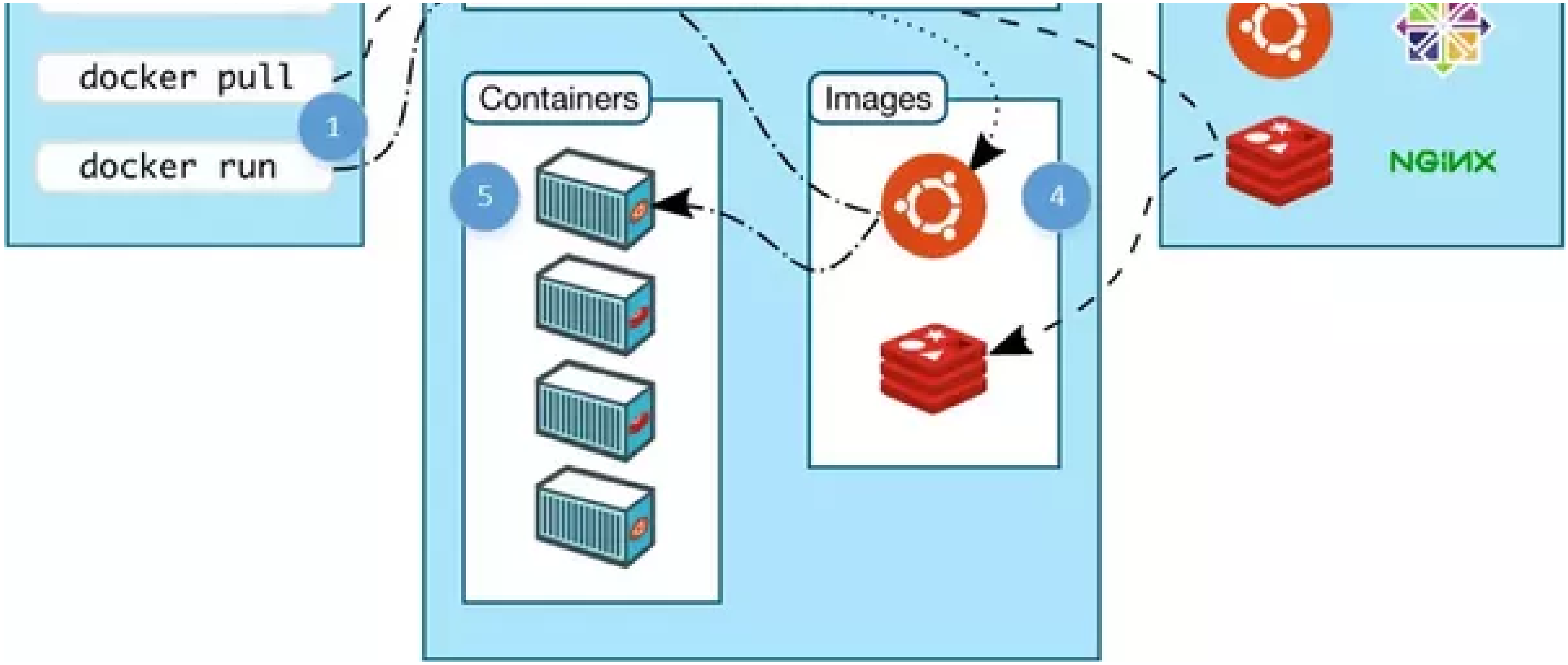
The cool part is if there are two containers with the image layers `a, b, c` and `a, b, d`, then you only need to store one copy of each image layer `a, b, c, d` both locally and in the repository. This is Docker's *union file system.*

Each image, identified by a hash, is just one of many possible layers of images that make up a container. However a container is identified only by its top level image, which has references to parent images. Two top level images (Image 1 and Image 2) shown here share the first three layers. Image 2 has two additional configuration related layers, but shares the same parent images as Image 1.



When a container is booted, the image and its parent images are downloaded from the repo, the cgroup and namespaces are created, and the image is used to create a virtual environment. From within the container, the files and binaries specified in the image appear to be the only files in the entire machine. Then the container's main process is started and the container is considered alive.

Docker has some other really really cool features, such as copy on write, volumes (shared file systems between containers), the docker daemon (manages containers on a machine), version controlled repositories (like Github for containers), and more. To learn more about them and see some practical examples of how to use Docker, this Medium article is extremely useful.



A command line client (1) tells a process on the machine called the docker daemon (2) what to do. The daemon pulls images from a registry/repository (3). These images are cached (4) on the local machine and can be booted up by the daemon to run containers (5). Image Source: Docker

Why Containers

Aside from process isolation, containers have many other beneficial properties.

The container serves as a self isolated unit that can run anywhere that supports it. And in each of these instances, the container itself will be exactly identical. It won't matter if the host OS is these instances, the container itself will be exactly identical. It won't matter if the host OS is CentOS, Ubuntu, MacOS, or even something non UNIX like Windows — from within the container the OS will be whatever OS the container specified. Thus you can be sure the container you built on your laptop will also run on the company's servers.

to scale an application, you simply need to scale the number of containers.

In this paradigm, each container is given a fixed resource configuration (CPU, RAM, # of threads, etc) and scaling the application requires scaling just the number of containers instead of the individual resource primitives. This provides a much easier abstraction for engineers when applications need to be scaled up or down.

Containers also serve as a great tool to implement *micro service architecture*, where each microservice is just a set of co-operating containers. For example the Redis micro service can be implemented with a single primary container and multiple replica containers.

This (micro)service orientated architecture has some very important properties that make it easy for engineering teams to create and deploy applications (see my earlier article for more details).

Orchestration

Ever since the time of linux containers, users have tried to deploy large scale applications over many virtual machines where each process runs in its own container. Doing this required being able to efficiently deploy tens to thousands of containers across potentially hundreds of virtual machines and manage their networking, file systems, resources, etc. Docker today makes this a

- actually take a specification and assign containers to machines (scheduling)
- actually boot the specified containers on the machines through Docker
- deal with upgrades/rollbacks/the constantly changing nature of the system
- respond to failures like container crashes
- and create cluster resources like service discovery, inter VM networking, cluster ingress/egress, etc.

This set of problems relates to the *orchestration* of a distributed system built on top of a set of (possibly transient or constantly changing) containers, and people have built some really miraculous systems to solve this problem.

In my next story I will talk in depth about the implementation of Kubernetes, the major open source orchestrator, along with two equally important but lesser known ones, Mesos and Borg.

This story is part of a series. I am an undergrad at UC Berkeley. My research is in distributed systems and I am advised by Scott Shenker.

systems and I am advised by Scott Shenker.

> Previous: How Microservices Saved the Internet

Trending Guides

10 to the Power of 0
Recursion

Forum
Donate

Learn to code — free 3,000-hour curriculum

What is Economics?
MBR VS GPT
Module Exports
Debounce
Python VS JavaScript
Helm Chart
Model View Controller
80-20 Rule
React Testing Library
OSI Model
ASCII Table Chart
HTML Link Code
Data Validation
SDLC
Inductive VS Deductive
JavaScript Keycode List
JavaScript Empty Array
JavaScript Reverse Array
Best Instagram Post Time
How to Screenshot on Mac
Garbage Collection in Java
How to Reverse Image Search
Auto-Numbering in Excel
Ternary Operator JavaScript

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty Code of Conduct Privacy Policy