VIRTUAL MACHINES

LECTURE 24 * 14-736 (DISTRIBUTED SYSTEMS) * SPRING 2019

WHAT IS VIRTUALIZATION?

"In computing, virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources."

-- https://en.wikipedia.org/wiki/Virtualization

WHAT, THEN, IS A VIRTUAL MACHINE?

- A virtual , rather than actual machine
  - Okay, so what does that mean?
  - Something with the "virtues" or good parts of a machines
  - Without the realities of being that machine!

- A software program, known as a *guest*, that runs on one computer, known as a *host*, that can run software as if it (the program) were an actual computer, often of a different type.

WHY USE VIRTUAL MACHINES?

- Share resources among many uses
  - One physical machine can host many guests
- Decouple the physical environment from the presented environment
  - Run Atari 2600 games on Macs or Windows PCs
  - Run Android and phone software on an Mac or Windows PC
  - Deliver several different Linux environments (different OS versions, libraries, etc) from one Linux host.
- Provide for protection
  - Different VMs for different domains, applications, users, etc.
- Provide for elasticity
  - Launch and Recall VMs as needed to meet demand.
  - Provide a unit of accounting, e.g. AWS.
- Provide a mechanism for migration, checkpointing, etc
  - Recovery, maintenance

WHY NOT USE VIRTUAL MACHINES?

- Complexity
  - There is overhead in maintaining multiple environments
  - Most of us just run native on our phones, laptops, etc.
- Efficiency
  - VMs introduce overhead, which can reduce increase costs and latency
  - Of course, sharing efficiently can lower costs and provide better efficiency for the same cost, too.
- Performance Isolation
  - Hard to manage well with sharing, but there are some tools.
- Protection
  - VMs mostly provide a better model for this and improve it
  - But, any sharing presents risks that real-world physical separation does not.

WHY CRITICAL IN CLOUD ENVIRONMENTS?

- Sharing of resources
  - Improve utilization
- Fungibility
  - Decouple guest hardware and software configuration from configuration
- Isolation
  - Pretty well defined protection model
- Elasticity
  - Easy to create and destroy
- Robustness
  - Model for checkpointing, recovering, migrating
- Metering
  - Can define to provide various qualities of service, e.g. processor speeds, memory models, networking capacities, etc.
  - All, for example, by time-sharing or space-sharing capabilities of host.

VIRTUALIZED APPLICATIONS

- In may ways, the ideal model
- Each app runs in its own VM
  - It has its own environment, which can be unique from the rest.
  - May include a few related apps
  - Has everything it needs packaged
- But, can mean a lot of overhead if many apps sharing the same host
  - We'll talk about containers soon

TYPES OF VIRTUAL MACHINES

- Full virtualization
  - Virtual machines runs entirely as a program in guest OS without any special support from guest OS
  - Nice in that it requires no specialized support
  - Not nice in that it has to grind through virtualizing expensive operations that can be done faster in host.
- Paravirtualization
  - Host OS modified to provide an API to enable VM to request the host to perform operations on behalf of the guest.
  - Makes operations that are inefficient to virtualize efficient
  - Requires a modified guest
  - May complicate protection and/or isolation models, etc.

HARDWARE SUPPORT

- Can eliminate many of the pain points
  - Traps
  - Hardware access, e.g. for I/O (Interrupts, DMA, etc)
  - Supervisor vs user mode
    - Consider what happens if a guest can run in supervisor mode
    - Consider what happen if a guest OS cannot run in supervisor mode
  - Etc
- Very powerful when combined with paravirtualization
- Ties implementation to specific, evolving hardware support.

RELATED TECHNOLOGIES

- Simulators
  - Simulates internal mechanisms as well as emulating behaviors, even when dramatically inefficient and unnecessary.
  - Mostly used for research, debugging with full transparency, etc
  - Far too slow for production use
- Containers ("OS-level virtualization")
  - Share not only hardware, but also OS components (Limits presented OS)
  - Improves efficiency
  - Complicates protection and isolation model
  - Examples: Docker, BSD Jails, etc.
  - More soon

VIRTUAL MACHINE MONITOR (VMM) A.K.A. HYPERVISOR

- Manages virtual machines
  - Creates
  - Destroys
  - Suspends
  - Resumes
  - Migrates
  - Etc.
- Typically manages all VMs on one host
- Name derived from *supervisor*, an old-school synonym for a running OS (kernel) in its role managing processes and resources
  - The hypervisor is, in some ways, a supervisor for the guest supervisors. Hyper sounding bigger than super, and all

TYPES OF HYPERVISORS

- Native Hypervisor, a.k.a. Bare-metal Hypervisor
  - Runs on guest hardware instead of conventional operating system
  - Old school IBM stuff and Microsoft Hyper-V (Based on trimmed down Windows) are classic examples
- Hosted Hypervisor
  - Runs as user software within a conventional operating system environment
  - VMware is classic example
- Reality isn't always so clear
  - What do you call a hosted hypervisor running on an OS that is hosting nothing but that hypervisor and the VMs it manages?
  - Does it matter if paravirtualization blends the line between OS, hypervisor, and VM?
  - KVM is classic example
- Obviously, the tighter the integration, the more efficient things likely will be.

CONTAINERS
A.K.A. OS VIRTUALIZATION

- Maintain one host OS
  - Guest OS is the same type
  - Share it for efficiency
- Isolate guests within host OS
  - For protection purposes
  - For environment purpose (Libraries, file system, users, etc)
  - Maybe for resource management purposes
- More efficient model
  - More sharing = Less overhead

CONTAINERS A.K.A. OS VIRTUALIZATION, *CONT*

- Built using existing OS mechanism
  - In many ways co-developed with those mechanisms
- But, weaker in some ways
  - Need to present same guest OS
  - Performance/Security model harder to understand
  - Limits to ability to control performance impact
- Model is "share first, isolate second"
  - Careful! Careful!

CONTAINER BUILDING BLOCKS: CHROOT

- chroot = change root
- Uses any directory within the file system's tree as the root for a process and its descendants.
  - It can't get out of this box in the file system
- Old school use: chroot a Web server.
  - Breaking the Web server doesn't expose host file system
- But, only isolates the file system

CONTAINER BUILDING BLOCKS: LINUX NAMESPACES

- Creates a partitioned view of certain linux kernel resources such that only certain processes can see certain resources
- Types of namespaces:
  - Mount (mnt)
  - Process ID (pid)
  - Network
  - Interprocess Communication (ipc)
  - Unix Time Share, a.k.a. uname (uts)
  - User ID (user)
  - Control group (cgroup)
- Basic model is that once resources are isolated into a namespace, only the original processes and their descendants can't get out of that view.
  - Now we can partition the view of the file system – and kernel resources

CONTAINER BUILDING BLOCKS: MOUNT (MNT) NAMESPACE

- Mount points are the points where one file system is grafted onto another file system.
- For example
  - /afs is the mount point where the AFS distributed file system is graphed into the visible file system.
  - /proc is the mount point where the kernel's virtual file system is mounted into the local file system
- The mnt namespace allows mounts to be viewed within certain namespaces, but not others
  - Subtrees can also be shared among namespaces.
    - This allows changes to mounts within them to be seen across the namespace.
- So, now we can not only limit what portion of a file system a process can see, but we can build it up by layering mounts on top of it – and deleting them.
  - And define them hierarchically (take X, add Y to form Z; take Z and add A and subtract B, etc)

CONTAINER BUILDING BLOCKS: PROCESS ID (PID) NAMESPACE

- Basically like a "chroot" for the pid tree
  - A new namespace is created and a process is forked into it using a more parameterizable version of fork() called clone()
  - This process now has pid=1 in this namespace.
  - Only its descendants are visible within the name space
- Careful! Careful!
  - Containers are a broken first, fixed from there model
  - Most tools get their process information from /proc, which is in the file system
    - So, unless this, too is "fixed", *top, ps*, etc, will all still show global processes (oops)

CONTAINER BUILDING BLOCKS: NETWORK (NET) NAMESPACE

- Network namespaces virtualize the network stack. On creation a network namespace contains only a loopback interface.
- Each network interface (physical or virtual) is present in exactly 1 namespace and can be moved between namespaces.
  - Virtual network interfaces can share physical ones, thereby allowing a physical network interface to be shared. (Kesden note)
- Each namespace will have a private set of IP addresses, its own routing table, socket listing, connection tracking table, firewall, and other network-related resources.
- Destroying a network namespace destroys any virtual interfaces within it and moves any physical interfaces within it back to the initial network namespace.

-- https://en.wikipedia.org/wiki/Linux_namespaces#Mount_(mnt)

CONTAINER BUILDING BLOCKS: INTER-PROCESS COMMUNICATION (IPC) NAMESPACE

- Inter-process communication makes use of shared memory to allow processes to communicate with each other
  - Pages of shared memory get mapped into multiple processes' virtual memory
  - Libraries make uses of this shared memory to provide structured communication
    - Shared memory, mailboxes, etc.
  - Synchronization primitives may also make use of it
    - Semaphores, mutexes, etc.
- This has an obvious impact upon isolation.
- IPC Namespaces partition IPC so it can't be used across name spaces

CONTAINER BUILDING BLOCKS: UNIX TIME SHARE, A.K.A. UNAME (UTS) NAMESPACE

- UTS is an old school name for what is returned by "uname -a"
- Basically, this namespace allows processes to have different hostnames and domain names.

CONTAINER BUILDING BLOCKS: USERID (UID) NAMESPACE

- You've guessed this one
- It lets different name spaces have different userids, including their mappings
  - Including uid(#) → userid(string) mappings
  - So, both uids and userids can e reused across name spaces.
  - root and uid=0 can both be reused within each user space
- The root user within a user space has the ability to do root things with any protected resources owned by that namespace or any of its descendants.
  - Recall, for example, that network interfaces are owned by some namespace and can be moved among them – but not shared among them.

CONTAINER BUILDING BLOCKS: CONTROL GROUP (CGROUP) NAMESPACE

- Control groups (cgroups) are basically a mechanism for the hierarchical grouping of processes for resource management
- Resource management is applied to a group – including its descendants
- Many different resources can be managed:
  - CPU, Memory, disk I/O. Network I/O, etc
- Monitoring and accounting can be done on a per-group basis.
- Some process management can be done on a per cgroup basis
  - Freezing, Resuming, Killing, etc.

UNION FILE SYSTEM (UNIONFS)

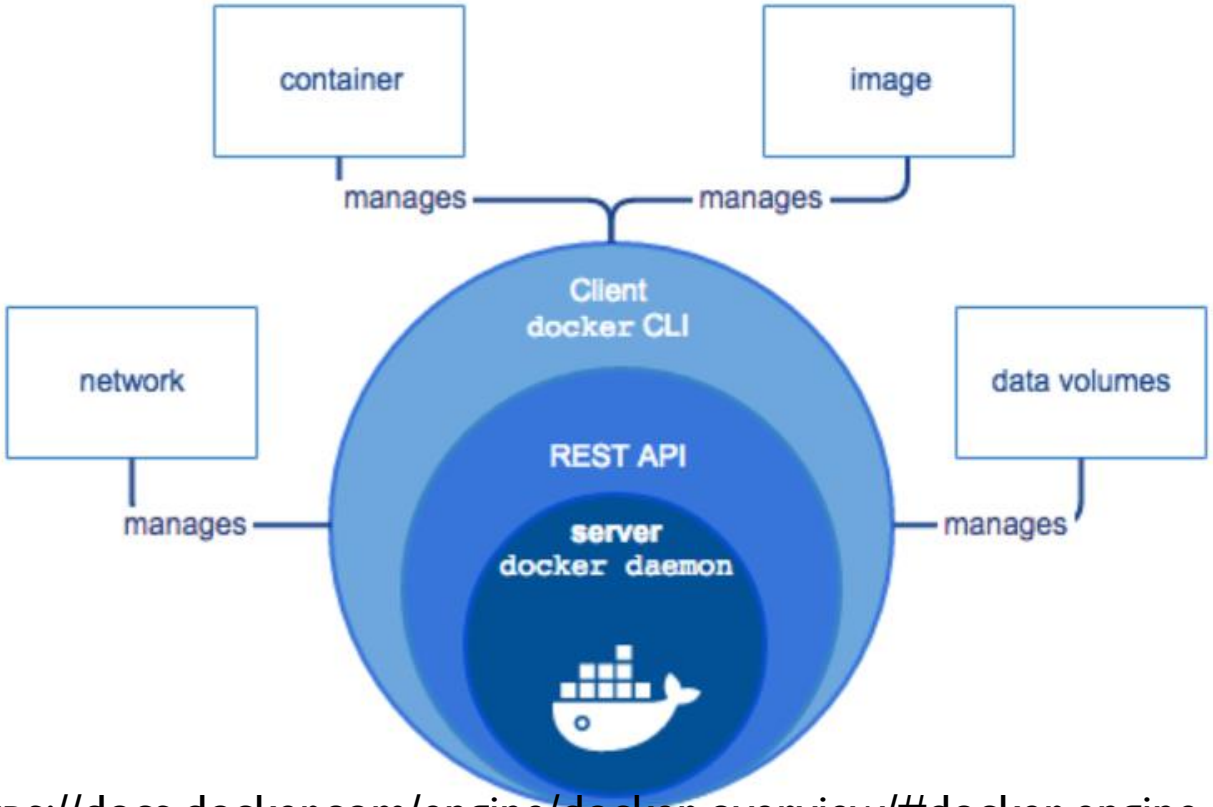
- Ability to mount one file system layered over another
- See lower level file systems through upper level file systems to the extent non-conflicting.
- But, upper level file systems hide conflicting things in lower layers
- Makes it easy to define one file system by specializing another
- In some sense it enables inheritance for a file system .

CONTAINER BUILDING BLOCKS: CONTAINER LIBRARIES

- Libraries may package functionality into an interface to support containerizing environments:
  - Examples include libcontainer, LXC, libvirt, &c e
- These are currently not generally portable
  - But, one can *imagine* a standardized interface working across operating systems with rich enough features
  - Wouldn't that be nice ☺

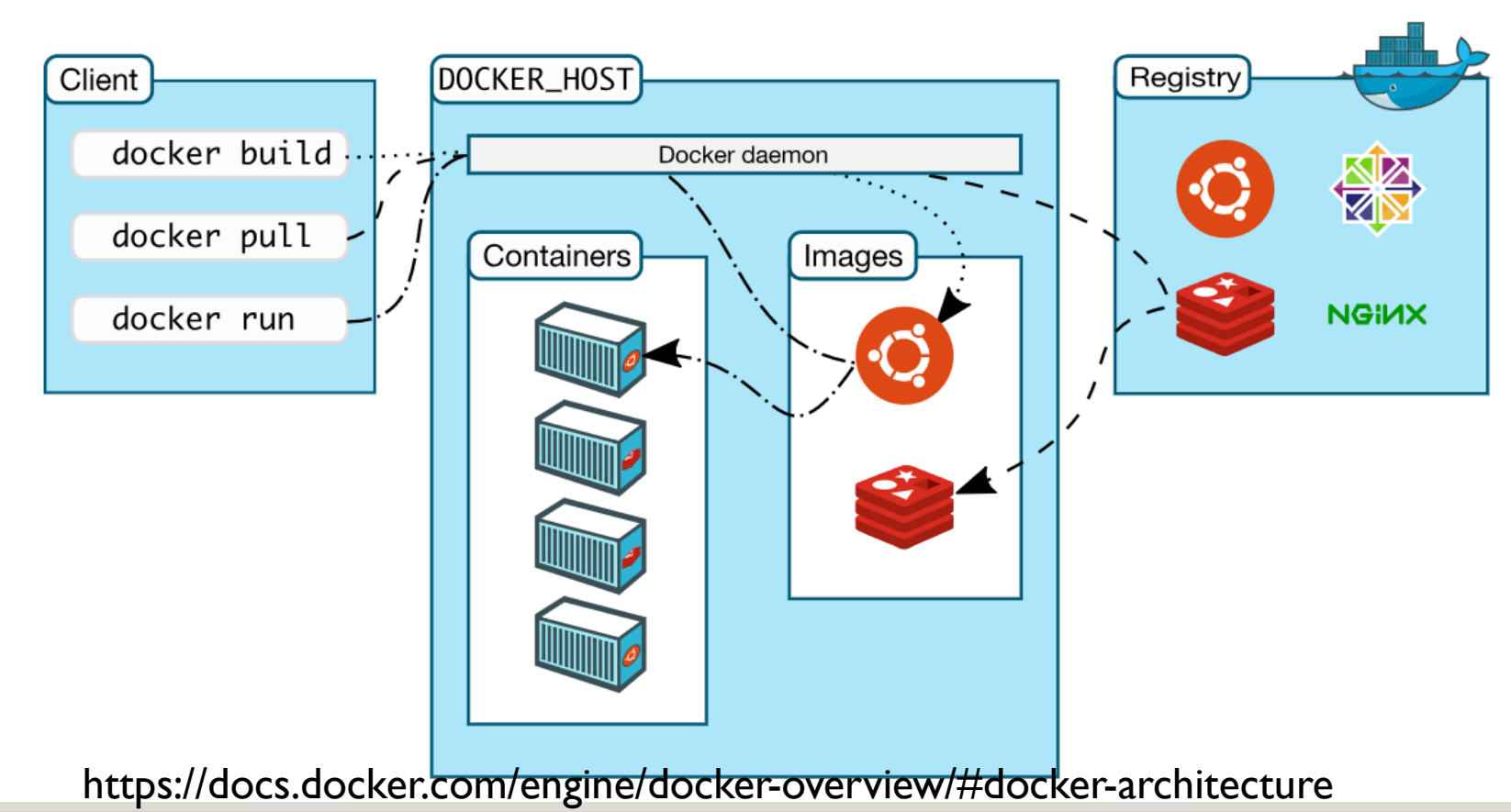
DOCKER

- Probably the best known Linux container solution including:
  - Network, data volumes, images, and containers
- Onion:
  - Daemon manages the containers
  - API provides for programmatic control of the daemon
  - Command-line interface (CLI) is built upon the API and provides command line tools



https://docs.docker.com/engine/docker-overview/#docker-engine

DOCKER ARCHITECTURE/ECOSYSTEM



https://docs.docker.com/engine/docker-overview/#docker-architecture

CONTAINERS + VMS

- In some ways these are competing technologies
  - Clear protection model and independence from host favors VMs
  - Efficiency, tuning a similar base environment, and reasonable isolation favors Containers
- But, they are also cooperative technologies
  - VMs can contain containers
  - Common model:
    - VM provides hardware independence (for sharing, fungibility, efficiency, isolation, robustness, etc)
    - Containers provide application environment (for deployability and further management)