CLOUD COMPUTING APPLICATIONS

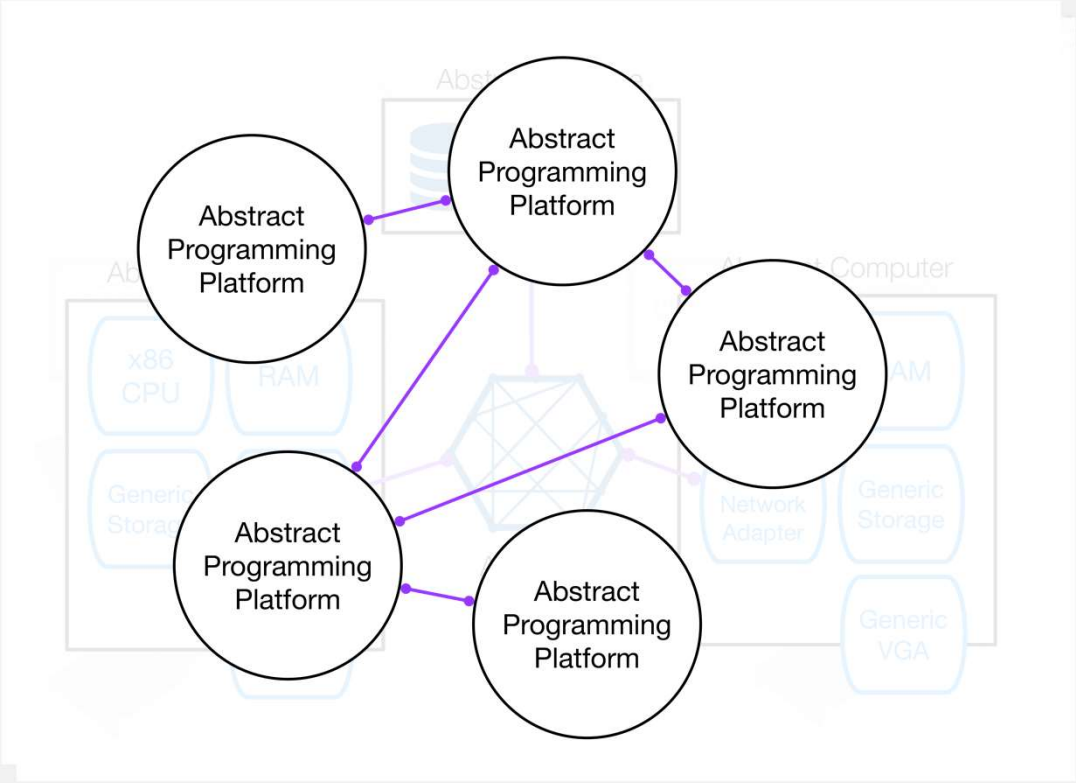
Virtualization

Prof. Reza Farivar

Sharing Resources

- Economics of Clouds requires sharing resources
- How do we share a physical computer among multiple users?
- Answer: Abstraction
  - Introduce an abstract model of what a generic computing resource should look like
  - The physical computer resource then provides this abstract model to many users

Layers of Abstraction

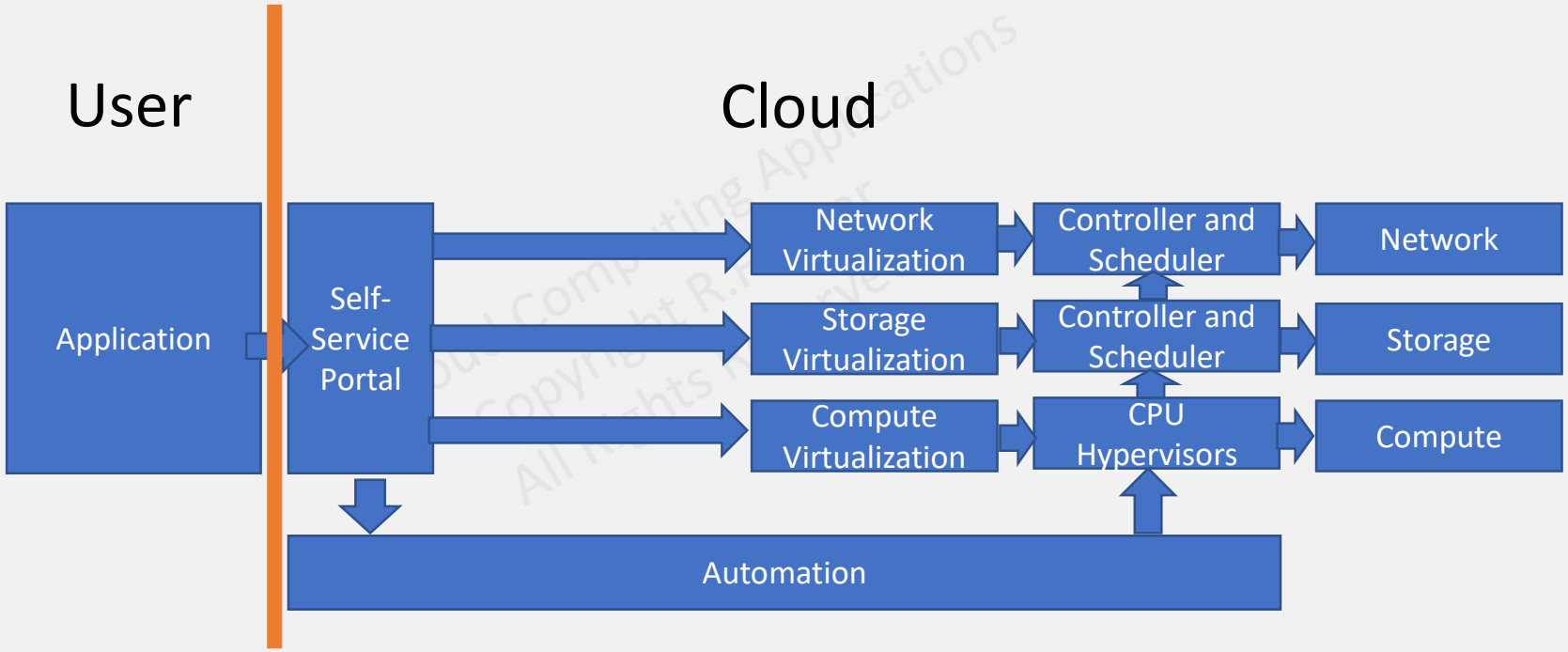


- Introduce an abstract model of what a generic computing resource should look like
- The physical computer resource then provides this abstract model to many users
- Virtualization avoids creating dependencies on physical resources

Virtualization: Foundation of Cloud Computing

- Virtualization allows distributed computing models without creating dependencies on physical resources
- Clouds are based on virtualization
  - Offer services based mainly on virtual machines, remote procedure calls, and client/servers
  - Provide lots of servers to lots of clients (e.g., phones)
- Simplicity of use and ease of programming require allowing client server paradigms to be used to construct services from lots of resources

Software-Defined Data Center



Types of Virtualization

- Emulation
- Full
  - Software
    - Binary Translation
    - Paravirtualization
  - Hardware assisted
- MicroVMs
- OS level
  - Containers

CLOUD COMPUTING APPLICATIONS

Virtualization: Background

Prof. Reza Farivar

Brief History Lesson

- Single program computers
  - VERY early mainframes (1950s)
  - MS-DOS
    - Single user program gets access to everything the hardware has
    - The OS is really a thin wrapper around BIOS
    - No real notion of process
- Multi-user / Multi-tasking
  - Need to isolate programs
  - Need to isolate users
  - Notion of Process
    - "executing program and its context”

World-view from a process

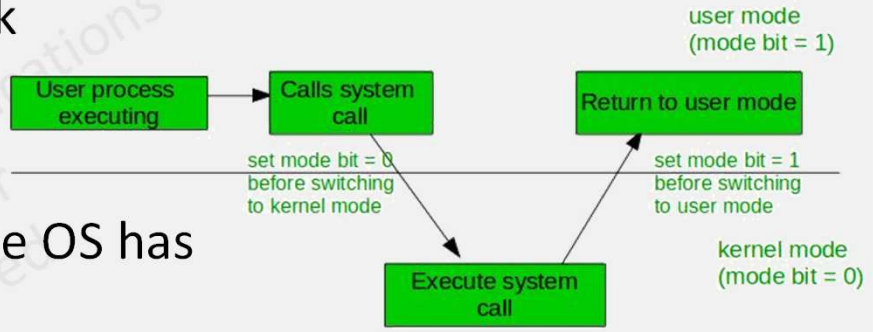
- An image of the program's executable machine code
- Memory
  - virtual address space → paging → a VM page is brought into memory when the process attempts to use it → managed by the OS
  - Process-specific data (input and output)
  - Stack: temp data, e.g. function parameters, local variables, return addresses, function call stack, and saved variables
  - Heap to hold intermediate data during run time
- OS resource descriptors: e.g. file descriptors, data sources and sinks
- Security attributes: e.g. process owner and the process' set of permissions (allowable operations)
- Processor state (context)
  - Program Counter
  - Content of registers and physical memory addressing

Process Isolation

- Need to isolate processes from each other
  - Virtualized, idealized, machine
  - A process is not capable of interacting with another process except through secure, kernel managed mechanisms
- User Processes should not be allowed to issue sensitive instructions
  - Things like loading memory mapping tables and accessing I/O devices.
- Normal applications better not use any of these instructions
- Imagine what would happen if a normal application like a word processor would suddenly be able to write to arbitrary memory locations, or get raw access to your hard drive.

Dual Mode Operations in OS

- The CPU and the Operating System work together to ensure process isolation



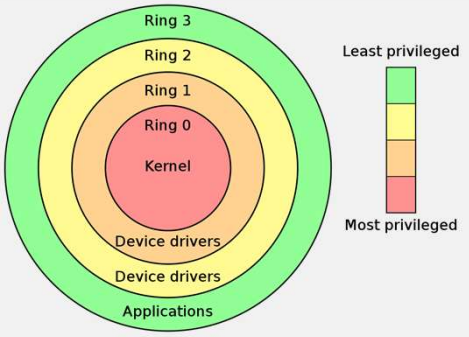
- To isolate processes from each other, the OS has two modes
  - User Mode
  - Kernel Mode

User and Kernel Modes

- User Mode
  - User processed operate in user mode
  - When the user application requests a service from the operating system, or an interrupt occurs, or a system call is made, there will be a transition from user to kernel mode to fulfill the requests
- Kernel Mode
  - When the system boots, hardware starts in kernel mode
  - privileged instructions which execute only in kernel mode
    - If user attempt to run privileged instruction in user mode then it will treat instruction as illegal and traps to OS
  - Example Privileged instruction: Input/Output management
  - Interrupt handling

CPU privilege protection

- When a privileged instruction is executed (or a safe instruction accesses a privileged resource), the CPU checks whether the process is allowed or not
  - Different mechanisms
  - x86 Example: Ring levels
  - Kernel mode code (OS, Device drivers, …) run in ring 0
  - User processes run in ring 3
- The CPU issues General Protection Fault (GPF) if a privileged instruction is executed in the wrong ring level



CPU + OS

- Certain operations are not allowed in user mode code
  - Read and write from a hardware device
  - Enabling/Disabling system interrupts
- Such operations only allowed in Kernel mode
- The task of enforcing this requirement is performed by the CPU
- Examples of privileged operations
  - HLT: Halt CPU till next interrupt
  - INVLPG: Invalidate a page entry in the translation look-aside buffer (TLB)
  - LIDT: Load Interrupt Descriptor Table
  - MOV CR registers: load or store control registers
    - In this case the MOV instruction (a non-privileged instruction on its own) is accessing a privileged register

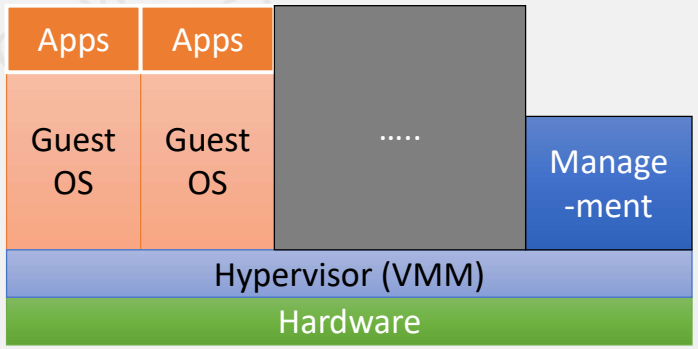
CLOUD COMPUTING APPLICATIONS

Virtualization: Full Virtualization

Prof. Reza Farivar

Full Virtualization

- The virtual machine simulates enough hardware to allow an unmodified "guest" OS (one designed for the same CPU) to be run in isolation
- The virtual machine looks and feels exactly like a real computer, up to the point where a guest operating system cannot tell the difference
- Examples:
  - VirtualBox
  - Virtual PC
  - VMWare
  - QEMU



Virtualization: Privileged and non-privileged instructions

- 1974 paper by Goldberg and Popek described criteria to make a system virtualizable
  - Trap and Emulate
- Safe Instructions
- Unsafe (sensitive) instructions
  - Privileged instructions a subset of unsafe
  - Privileged instruction should cause a trap
- The original X86 was not virtualizable according to the above paper
  - 17 unsafe instructions that were not privileged
    - Intel VT-x and AMD-V later made these privileged

Trap and Emulate

- The classical way to implement a hypervisor is using the "trap and emulate" approach
  - This approach was used by the very first hypervisor developed by IBM in the late 60s
    - IBM System 370
    - Used again today on 64-bit Intel and AMD systems
- The approach usually has good performance, because the majority of the instructions will not cause a trap, and will execute straight on the CPU with no overhead.

Trap and Emulate

- Executable code from the guest can execute directly on the host CPU by the hypervisor
- the hypervisor configures the CPU in such a way that all potentially unsafe instructions will cause a "trap”
- An unsafe instruction is one that for example tries to access or modify the memory of another guest.
- A trap is an exceptional condition that transfers control back to the hypervisor.
- Once the hypervisor has received a trap, it will inspect the offending instruction, emulate it in a safe way, and continue execution after the instruction

CLOUD COMPUTING APPLICATIONS

Virtualization: Paravirtualization

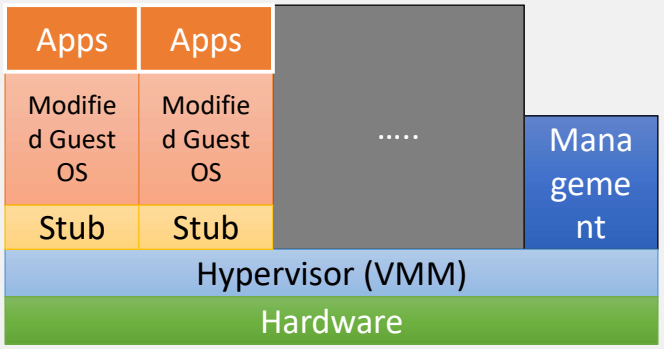
Prof. Reza Farivar

Software-only Virtualization

- Problem: x86 processors were not virtualizable until mid 2000s
- Software-only virtualization is a technique to go around the trap and emulate design of Popek and Goldberg
- Does not need special hardware support, e.g. the Intel "VT-x" or "AMD-V" features

Paravirtualization

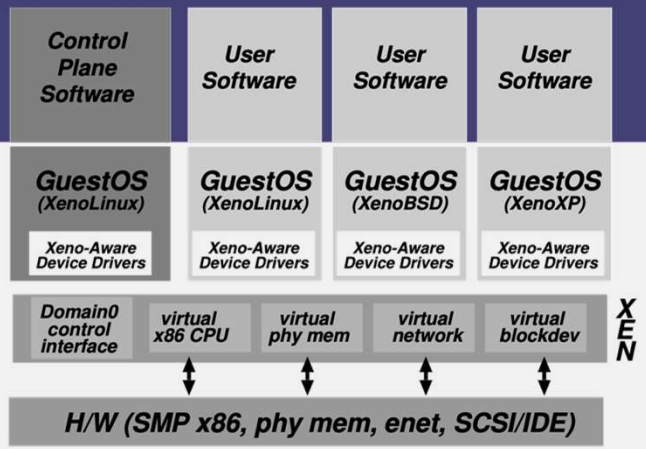
- First approach to software-only virtualization
- The virtual machine does not necessarily simulate hardware, but instead (or in addition), offers a special API that can only be used by **modifying** the "guest" OS
  - Paravirtualization is a technique in which a modified guest operating system kernel communicates to the hypervisor its intent to perform privileged CPU and memory operations
- The guest OS is specifically modified to run on a hypervisor
  - Windows 7 and newer
  - Linux Kernel version 3 and later
- Example:
  - XEN



Xen and the Linux Kernel

- Xen was initially a university research project
- Invasive changes to the kernel to run Linux as a paravirtualized guest
- Maintenance effort required on distributions
  - Support was added in mainstream Linux Kernel 3 (2012)
- Usually very fast → Trap and Emulate has overhead, paravirtualization eliminates traps
- Risk of distributions dropping Xen support

Xen Concepts



- Control Domain 0 a.k.a. Dom0
  - Dom0 kernel with drivers
  - Xen management tool stack
  - Trusted computing base
- Guest Domains
  - Your apps
  - For example, your cloud management stack
- Driver/Stub/Service Domain(s)
  - A "driver, device" model or "control service in a box"
  - De-privileged and isolated
  - Lifetime: start, stop, kill

CLOUD COMPUTING APPLICATIONS

Virtualization: Binary Translation

Prof. Reza Farivar

Binary Translation

- Binary translation modifies sensitive instructions on the fly to virtualizable instructions
  - we only need to translate kernel code that is executing in ring 0
    - Depending on the workload, this is a small fraction of the total code
  - Examine the executable code of the virtual guest for "unsafe" instructions
  - Translate these into "safe" equivalents
  - Execute the translated code
- Direct Execution
  - most code is executed directly on the CPU, and only the code that needs to be translated is actually translated
- Binary translation is performed on the binary code that gets executed on the processor it does not require changes to the guest operating system kernel

Binary Translation

- Binary translation was first described in a paper from 1992 by Digital Equipment Corporation (DEC)
- Original VMWare Workstation 1.0 in 1999
  - BT support deprecated since 2016
- Somewhat similar to Just in Time compilation for Java Virtual Machine (JVM), Javascript (V8 in Chrome), PHP 8 (since November 2020)

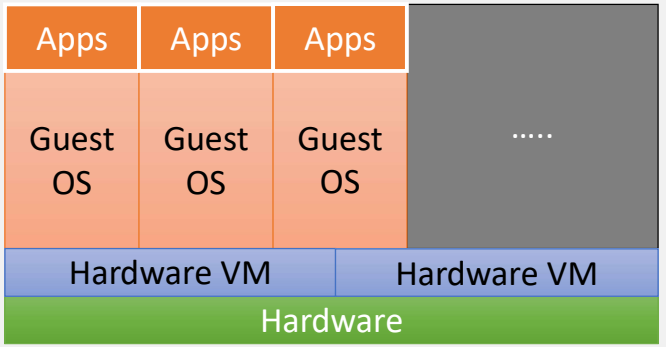
CLOUD COMPUTING APPLICATIONS

Virtualization: 1st Gen Hardware Virtualization

Prof. Reza Farivar

Hardware-Enabled Virtualization

- Intel VT (IVT)
- AMD virtualization (AMD-V)
- Allow “trapping” of sensitive instructions
  - Popek & Goldbreg → Trap and Emulate
- Examples:
  - VMWare Fusion, ESX
  - Parallels Desktop for Mac
  - Parallels Workstation



First Generation Hardware Virtualization

- First introduced in x86 in mid 2000s
- Intel VT-x, AMD-V
  - Virtual machine control block (`VMCB`).
    - in-memory data structure
  - The VMCB combines control state with a subset of the guest VCPU state
- A new, less privileged execution mode, guest mode, supports direct execution of guest code, including privileged kernel code

First Generation Hardware Virtualization

- A new instruction, `vmrun`, transfers from host to guest mode.
  - Upon execution of `vmrun`, the hardware loads guest state from the `VMCB` and continues execution in guest mode
  - Guest execution proceeds until some condition (set by VMM) is reached
    - The hardware performs an `exit` operation
    - `exit` is the inverse of `vmrun`
  - Guest state is saved to the `VMCB`, VMM state is loaded, and execution resumes in host mode, now in the VMM.

First Generation Hardware Virtualization

- First generation hardware support lacks explicit support for memory virtualization
  - The VMM must implement a software MMU using shadow page tables
  - → context switch on each `vmrun` and `exit`
    - VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, INVVPID, and VMFUNC
- With hardware-assist, the guest runs at full speed, unless an exit is triggered
  - Virtualization overheads are determined as the product of the exit frequency and the average cost of handling an exit

MMU in First Generation Hardware Virtualization

- First gen hardware virtualization does not virtualize MMU
- The VMM has to get involved on MMU
  - VMM write-protects primary page tables to trigger `exits` when the guest updates primary page tables so that the VMM can propagate the change into the shadow page tables (e.g., invalidate).
  - the VMM must request `exits` on page faults to distinguish between hidden faults, which the VMM consumes to populate shadow page tables, and true faults, which the guest consumes to populate primary page tables.
  - the VMM must request `exits` on guest context switches so that it can activate the shadow page tables corresponding to the new context.
- First generation hardware support often did not outperform a BT- based VMM, often slower

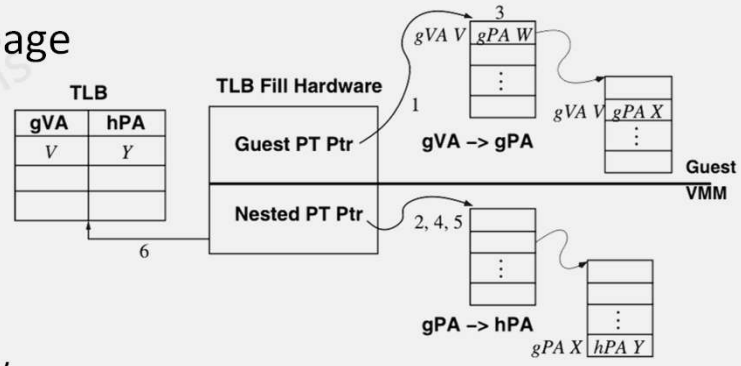
CLOUD COMPUTING APPLICATIONS

Virtualization: 2nd & 3rd Gen Hardware Virtualization

Prof. Reza Farivar

Second Generation Hardware Virtualization

- AMD's RVI and Intel's EPT (Extended Page Tables)
- The VMM maintains a hardware-walked "nested page table" that translates gPAs to hPAs
  - eliminating the need for VMM interposition
- Many issues of the first-gen are resolved
  - No trace-induced `exits`
  - no context-switch `exits`
  - no hidden/true fault `exits`
  - The VMM does not have to allocate memory for shadow page tables, reducing memory usage
- The cost to service a TLB miss will be higher with nested paging than without
  - TLB Caching helps a lot
  - Large Memory pages (1 GB vs 2 MB)



I/O Virtualization

- Most hypervisors “emulate” I/O devices
  - Generic display
  - Generic network
  - Generic storage
- Trap and Emulate idea
  - Or paravirtualization
- Cloud Data Center requirements necessitate optimal performance
  - Hardware-based I/O Virtualization

Third Generation Hardware Virtualization

- Since the Haswell microarchitecture (announced in 2013), Intel started to include VMCS shadowing as a technology that accelerates **nested virtualization** of VMMs
- Interrupt Virtualization (AMD AVIC and Intel APICv) 2012
- I/O MMU virtualization (AMD-Vi and Intel VT-d)
  - An input/output memory management unit (IOMMU) allows guest virtual machines to directly use peripheral devices, such as Ethernet, accelerated graphics cards, and hard-drive controllers, through DMA and interrupt remapping.
  - This is sometimes called PCI passthrough
- PCI-SIG Single Root I/O Virtualization (SR-IOV)

CLOUD COMPUTING APPLICATIONS

Virtualization: MicroVMs and Unikernels

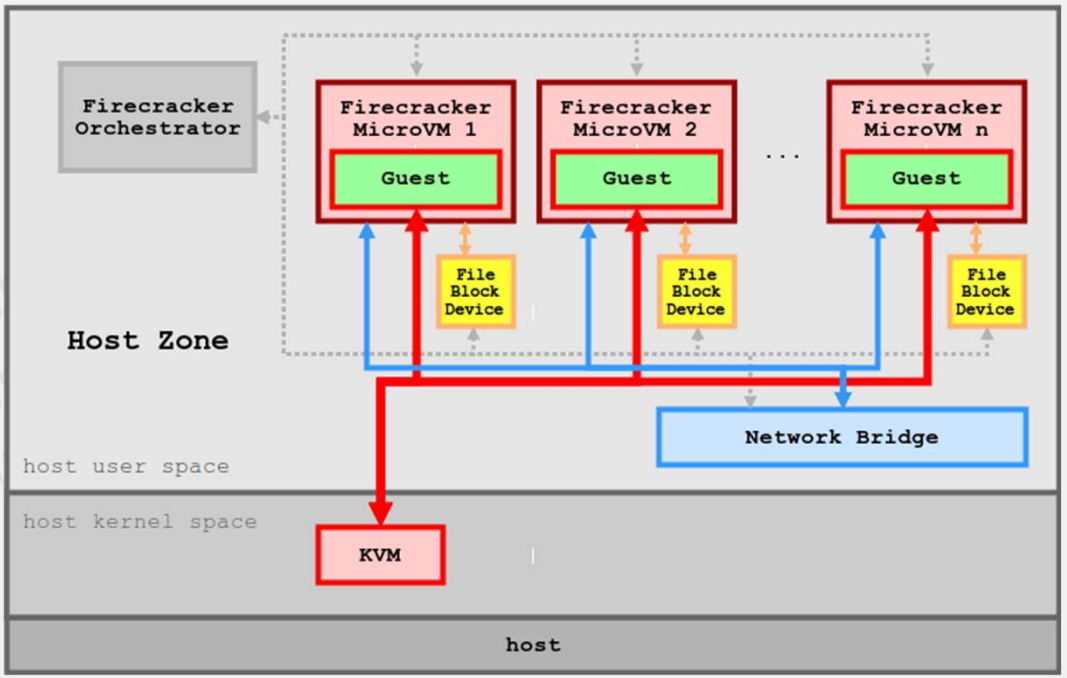
Prof. Reza Farivar

MicroVMs

- A typical virtual machine usually has many virtual I/O devices to make it usable
  - Think about an EC2 instance
    - Virtual storage
    - Virtual Network
    - Virtual Display
    - USB, Audio, …
  - The guest operating system supports device drivers, kernel modules, etc. for all
  - Typical load time in tens of seconds, if not minutes
- MicroVMs designed for cloud native
  - Serverless Computing use cases
  - Serverless containers
    - Fargate
  - Function as a Service
    - Lambda

FireCracker

- Open Source Project by Amazon
- Based on top of the Linux KVM
  - Similar to QEMU (VMM driver)
- Idea: as light weight as possible
  - VMM
  - Guest OS
- No support for graphics drivers
- The only virtual devices:
  - Paravirtualized virtio net
  - Paravirtualized virtio block
  - a one-button keyboard
    - To reset the VM
  - Interrupt controller
  - Timer
  - Clock

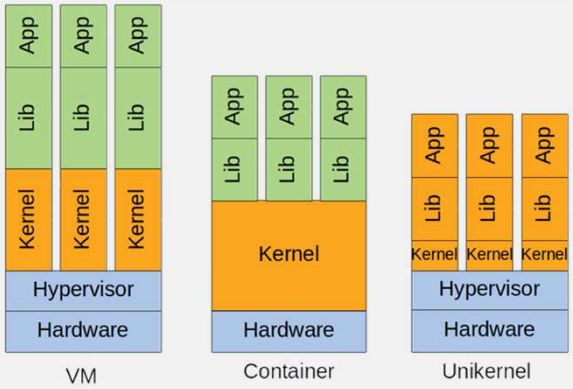


FireCracker

- VMM starts in 8 ms
- VM Start time less than 125 ms
  - Firecracker InstanceStart API call to the start of the Linux guest user-space /sbin/init process
  - Lightweight Linux guest, e.g. Alpine Linux
- Memory overhead less than 5 MiB
- OSv on FireCracker
  - Specialized OS
  - Boot time in less than 5 ms

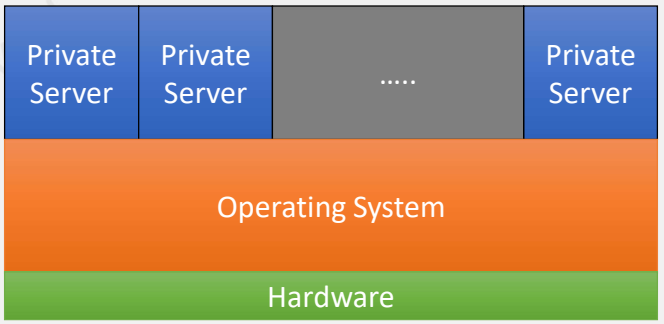
Unikernel

- Unikernels are a relatively new concept
- Software is directly integrated with the kernel it is running on
- Compiling source code, along with only the required system calls and drivers, into one executable program using a single address space
- Unikernels can only run a single process, thus forking does not exist
- The build process results in a complete (virtual) machine image of minimal size that only contains and executes what it absolutely needs
- Example: OSv → Can run on FireCracker in 5 ms (compare to 125 ms for Linux), 18 MiB memory overhead, can run any Linux Executable



Operating System-Level Virtualization

- Virtualizing a physical server at the operating system level, enabling multiple isolated and secure virtualized servers to run on a single physical server
- Examples:
  - Linux-Vserver
  - Solaris Containers
  - FreeBSD Jails
  - Chroot
  - CGroups



CLOUD COMPUTING APPLICATIONS

Containers

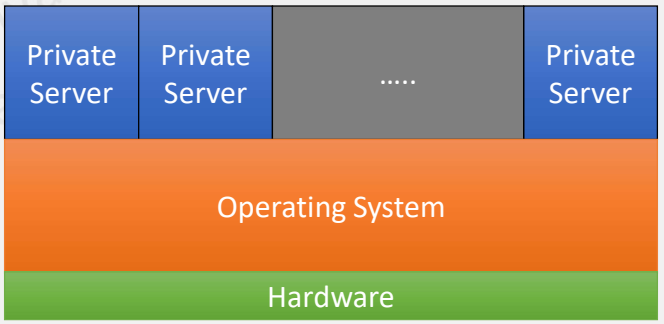
Prof. Reza Farivar

Isolation

- “I once heard that hypervisors are the living proof of operating system's incompetence”
  - -Glauber Costa, 2012
- hypervisors have indeed provided a remedy for certain deficiencies in operating system design
- for some cases, containers may be an even better remedy for those deficiencies

Operating System-Level Virtualization

- Virtualizing a physical server at the operating system level, enabling multiple isolated and secure virtualized servers to run on a single physical server
- Examples:
  - Solaris Containers (2004)
  - FreeBSD Jails (2000)
  - Linux Containers
    - Linux Vserver (2001)
    - OpenVZ (2005)
    - Process Container (2006) → cgroups
    - LXC (2008)
    - Docker (2013)



OS-Virtualization / Containers

- OS (operating system) virtualization is how we generally refer to this type of "light-weight" virtualization
- processes think they see a virtual kernel, but are all sharing the same real kernel under the hood
- kernel acts as a sort of hypervisor in ensuring that container/virtualization boundaries are not crossed
- The goal of containers is to support all of the resource-isolation use cases, without the overhead and complexity of running multiple kernel instance

Operating System-Level Virtualization

- Hypervisor (VM)
- One real HW, many virtual HWs, many OSs
- High versatility – can run different Oss
- Lower density, performance, scalability
- Performance overhead is mitigated by new hardware features (such as VT-D)

- Containers (CT)
- One real HW (no virtual HW), one kernel, many userspace instances
- Higher density, natural page sharing
- Dynamic resource allocation
- Native performance: [almost] no overhead

CLOUD COMPUTING APPLICATIONS

Pillars of Linux Containers

Prof. Reza Farivar

Three Pillars of Linux Containers

- cgroups
- Namespaces
- Unionfs

- Not chroot!

chroot

- In a Unix-like OS, root directory(/) is the top directory
  - All file system entries branch out of this root
  - 1979, Unix V7
- Each process has its own idea of what the root directory is
  - By default, it is actual system root
  - But we can change this by using chroot()system call

chroot

- `chroot` changes ***apparent*** root directory for current running process and its children
  - chroot() simply modifies pathname lookups for a process and its children
  - prepends the new root path to any name starting with /
  - Current directory is not modified and relative paths can refer any locations outside of new root
- **chroot() does NOT provide secure isolation**
  - **Docker uses mount namespace instead**

CLOUD COMPUTING APPLICATIONS

Containers: cgroups

Prof. Reza Farivar

Cgroups

- Control Groups
- Linux kernel feature which limits, isolates and measures resource usage of a group of processes
  - Since Linux Kernel 2.6.24
- Resources quotas for memory, CPU, network and IO
- Create a control group and assign resource limits on it:
  - e.g. 3GB of memory limit and 70% of CPU
- Add a process id to the group
- Process resource usage will be throttled
  - The application may exceed the limits in normal scenarios
  - it will be throttled back to pre set limits in case system is facing resource crunch

Cgroup Controllers (v2)

Controller	Brief Description
cpu	Cgroups can be guaranteed a minimum number of "CPU shares" when a system is busy. This does not limit a cgroup's CPU usage if the CPUs are not busy.
cpuset	Provides accounting for CPU usage by groups of processes
freezer	can suspend and restore (resume) all processes in a cgroup. Freezing a cgroup */A* also causes its children, for example, processes in */A/B*, to be frozen.
hugetlb	This supports limiting the use of huge pages by cgroups
io	The *io* cgroup controls and limits access to specified block devices by applying IO control in the form of throttling and upper limits against leaf nodes and intermediate nodes in the storage hierarchy. Two policies are available. The first is a proportional- weight time-based division of disk implemented with CFQ. This is in effect for leaf nodes using CFQ. The second is a throttling policy which specifies upper I/O rate limits on a device.
memory	The memory controller supports reporting and limiting of process memory, kernel memory, and swap used by cgroups.
Perf_event	This controller allows *perf* monitoring of the set of processes grouped in a cgroup.
pids	This controller permits limiting the number of process that may be created in a cgroup (and its descendants).
rdma	The RDMA controller permits limiting the use of RDMA/IB- specific resources per cgroup.

Detailed Documentation at: https://www.kernel.org/doc/Documentation/admin-guide/cgroup-v2.rst

Cgroup example

- Controllers mounted in the cgroups file system
  - /cgroup directory
    - `/sys/fs/cgroup/memory`
    - `/sys/fs/cgroup/cpu`
- Making a control group
  - `/cgroup/memory/mytestcgroup`
- Setting limits
  - `echo 2097152 > /sys/fs/cgroup/memory/mytestcgroup/memory.limit_in_bytes`
  - `echo 2097152 > /sys/fs/cgroup/memory/mytestcgroup/memory.memsw.limit_in_bytes`
    - Set both memory AND swap space limit to 2 MB
- Running a process
  - `cgexec -g memory:mytestcgroup ./<binary_name>`

Cgroup Scheduling

- When we think of containers as lightweight VMs, it is natural to think of resources in terms of discrete resources such as number of processors.
- However, the Linux kernel schedules processes dynamically, just as the hypervisor schedules requests onto discrete hardware.
- the cpu subsystem schedules CPU access to each cgroup using either the Completely Fair Scheduler (CFS)—the default on Linux and Docker—or the Real-Time Scheduler (RT).
- Scheduling cgroups in CFS requires us to think in terms of time slices instead of processor counts.
- CPU shares provide tasks in a cgroup with a relative amount of CPU time, providing an opportunity for the tasks to run.
- The file `cpu.shares` defines the number of shares allocated to the cgroup.

CLOUD COMPUTING APPLICATIONS

Containers: Namespaces

Prof. Reza Farivar

Namespaces

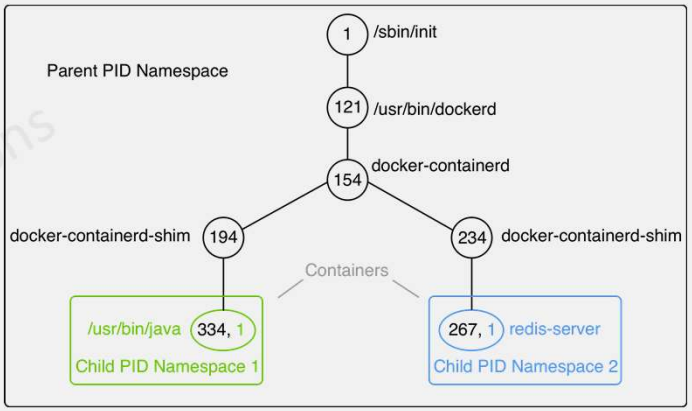
- A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource
- Linux processes form a single hierarchy, with all processes rooting at init.
- Usually privileged processes in this tree can trace or kill other processes.
- Linux namespace enables us to have many hierarchies of processes with their own “subtrees” such that processes in one subtree can NOT access or even know of those in another.

Namespaces

Namespace	Description (This namespace isolates …)
Cgroup	Cgroup root directory
IPC	Isolates System V IPC, POSIX message queues. The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames.
Network *	Network devices, stacks, ports, etc. each network namespace has its own network devices, IP addresses, IP routing tables, /proc/net directory, port numbers, etc.
Mount *	Mount points. processes in different mount namespaces can have different views of the filesystem hierarchy
PID *	Isolates Process IDs. In other words, processes in different PID namespaces can have the same PID.
Time	Boot and monotonic clocks
User *	User and group IDs. In other words, a process's user and group IDs can be different inside and outside a user namespace
UTS	Hostname and NIS domain name. Allows each container to have its own hostname and NIS domain name. Affects nodename and domainname—returned by the uname() system call.

PID Namespace example

- Without namespace, all processes descend hierarchically from PID 1(init).
- If we create a PID namespace and run a process in it, that first process becomes PID 1 in that namespace.
- The process that creates namespace still remains in parent namespace, but makes its child the root of new process tree.
- The processes within the new namespace can not see parent process but the parent process namespace can see the child namespace.
- The processes within new namespace have 2 PIDs: one for new namespace and one for global namespace.
- PID namespaces also allow each container to have its own `init` (PID 1), the "ancestor of all processes" that manages various system initialization tasks and reaps orphaned child processes when they terminate



Network Namespace

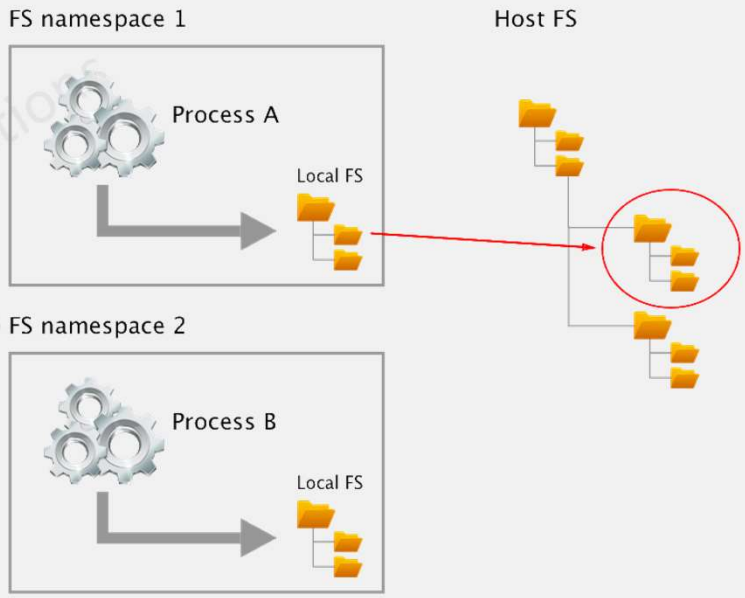
- Provide isolation of the system resources associated with networking
- each network namespace has its own network devices, IP addresses, IP routing tables, /proc/net directory, port numbers, and so on
- Network namespaces make containers useful from a networking perspective
  - Each container can have its own (virtual) network device and its own applications that bind to the per-namespace port number space
  - suitable routing rules in the host system can direct network packets to the network device associated with a specific container
- E.g. have multiple containerized web servers on the same host system, with each server bound to port 80 in its (per-container) network namespace

User Namespace

- process's user and group IDs can be different inside and outside a user namespace
- The most interesting case here is that a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace.
  - This means that the process has full root privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.
- From a security perspective this is a great feature as it allows our containers to continue running with root privileges, but without actually having any root privilege on the host.
  - Docker since 2016

Mount Namespace

- Mount namespaces were the first type of namespace to be implemented on Linux, appearing in 2002
  - by contrast with the use of the chroot() system call, mount namespaces are a more secure and flexible tool for this tas
- Isolate the set of filesystem mount points seen by a group of processes
- Processes in different mount namespaces can have different views of the filesystem hierarchy



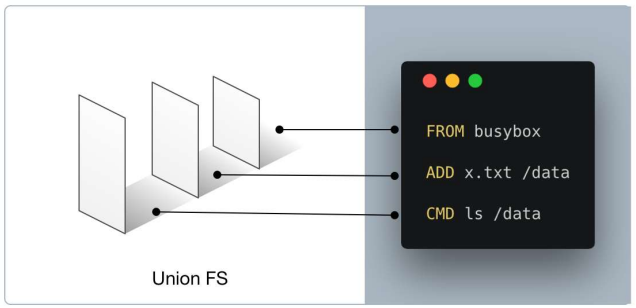
CLOUD COMPUTING APPLICATIONS

Containers: Union Filesystem

Prof. Reza Farivar

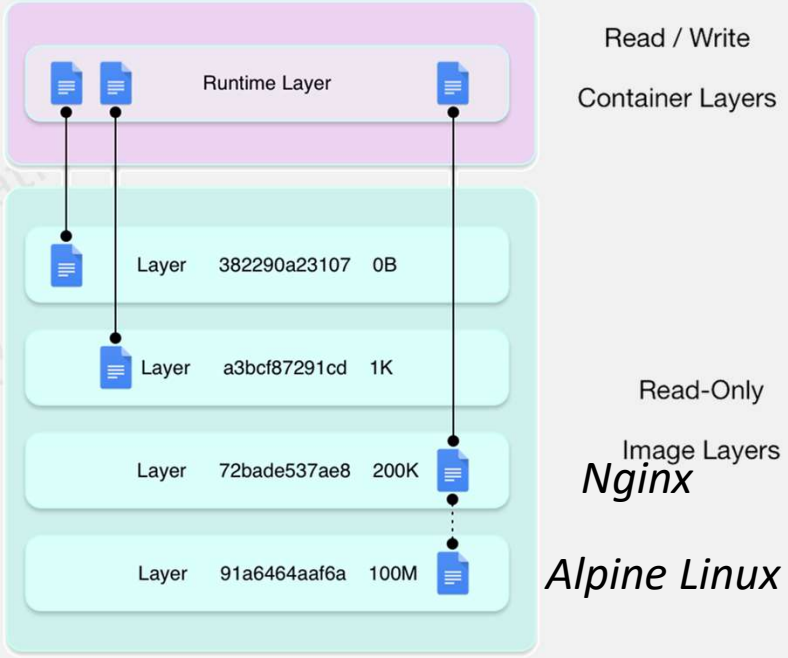
Union File System (Unionfs)

- Backbone of container images
- A stackable unification file system, which can appear to merge the contents of several directories (branches), while keeping their physical content separate
- overlays several directory into one single mount point
  - Contents of directories that have the same path within the merged branches will be seen together in a single merged directory, within the new virtual filesystem



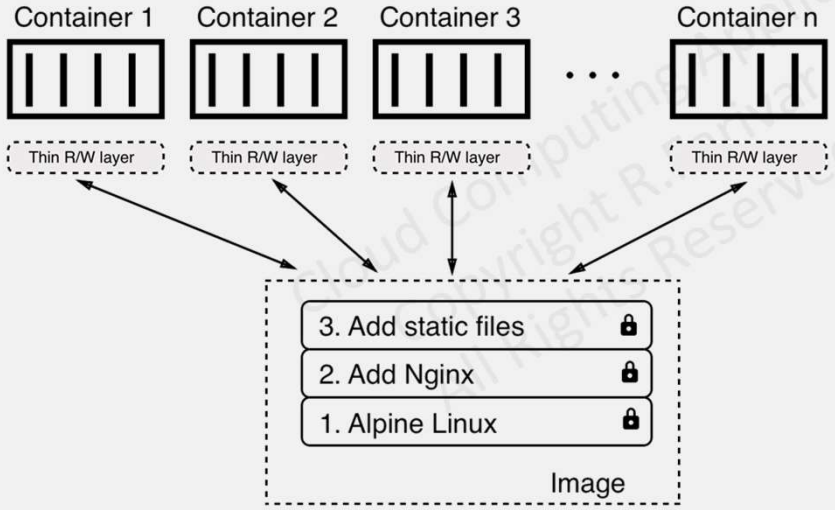
Union File System (Unionfs)

- With union mount, the directories in the file system from the underlying layer are getting merged with those from the upper layer file systems
- To access a file: first tries to access the file on the top branch and if the file does not exist there, it continues on lower level branches
- **copy-on-write (cow)**: If the user tries to ***modify*** a file on a lower level read-only branch the file is ***copied*** to to a higher level read-write branch
- the program running inside the container doesn't care which layer the files and directories comes from



Docker images

- A container image is made of a stack of immutable or read-only layers
- In run-time, the docker engine adds a R/W layer on top of the stack of immutable layers



Dockerfile

```
FROM node:12-alpine
WORKDIR /app COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
```

Graph Driver

- A local instance of a Docker engine has a cache of Docker image layer
- This cache of layers is built up as explicit `docker pull` commands are executed, as well as `docker build`
- The driver to handle these layers is called a "graphdriver"

Graph Driver Options

- graphdriver options: vfs, aufs, overlay, overlay2, btrfs, zfs, devicemapper, and windows
- vfs: Naïve implementation, does **not** use a union filesystem or CoW technique
- Overlay, overlay2, aufs: unionfs on top of a real filesystems
  - ext4, xfs
- btrfs, zfs, devicemapper, windows: the underlying real filesystem performs the task of union

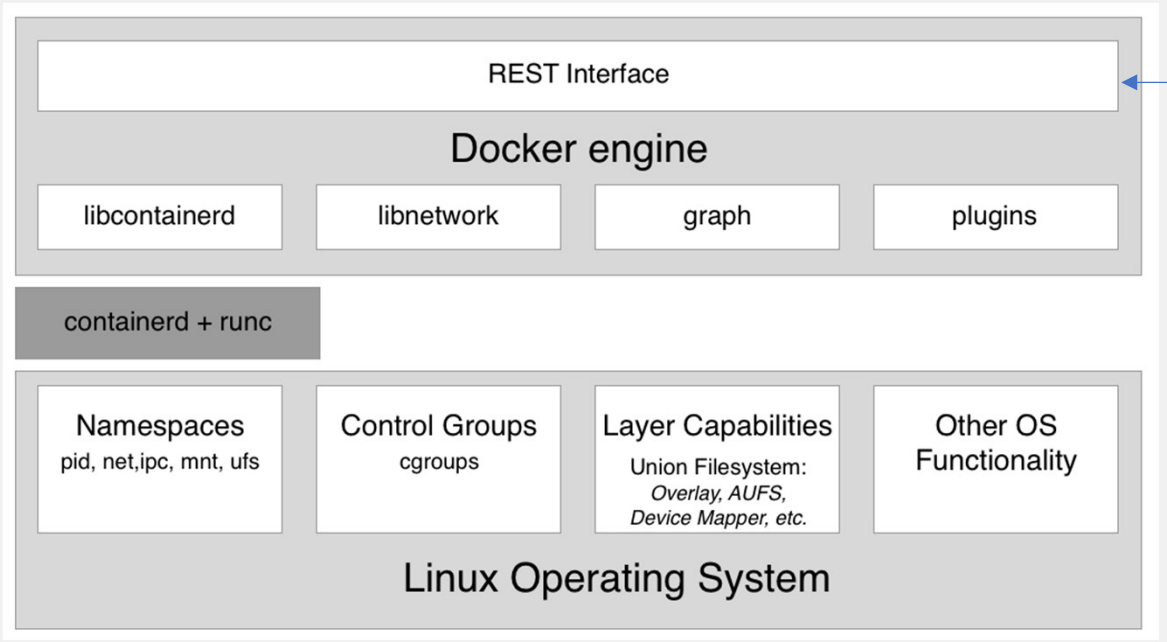
More details: https://integratedcode.us/2016/08/30/storage-drivers-in-docker-a-deep-dive/

CLOUD COMPUTING APPLICATIONS

Containers: Docker Architecture

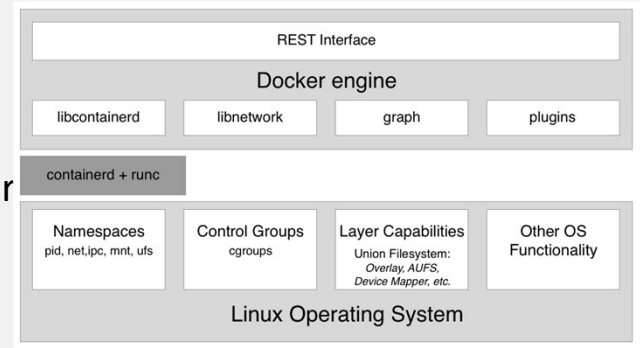
Prof. Reza Farivar

Docker Architecture



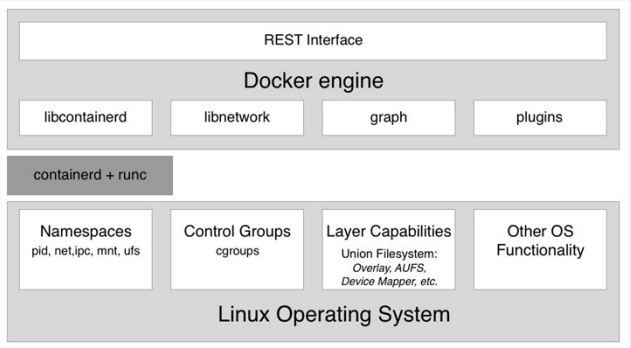
Container Runtime

- Docker was originally monolithic
  - Later the runtime was separated
- Responsible for the whole life cycle of a container
- pulls a container image (which is the template for a container) from a registry
- Creates a container from that image
- Initializes and runs the container
- Eventually stops and removes the container from the system
- The container runtime on a Docker host consists of `containerd` and `runc`
- Both are open source and have been donated by Docker to the CNCF
  - CNCF: Cloud Native Computing Foundation, a Linux Foundation project



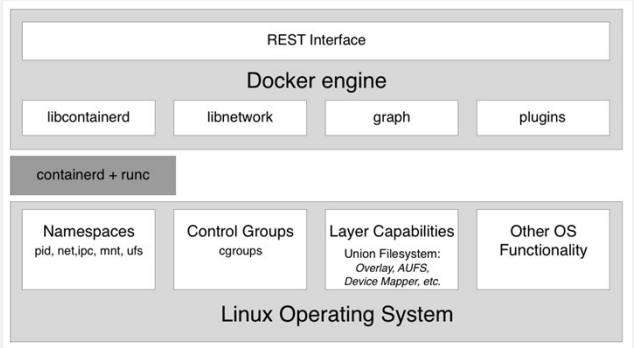
Container Runtime: `containerd`

- `containerd` is based on `runc`, provides higher-level functionality
  - Image push and pull
  - Managing of storage
  - executing of Containers by calling **runc** with the right parameters to run containers
  - Managing of network primitives for interfaces
  - Management of network namespaces for containers to join existing namespaces
- reference implementation of the OCI specifications
  - OCI: Open Container Initiative, a Linux Foundation Project
- `containerd` was donated to and accepted by the CNCF in 2017



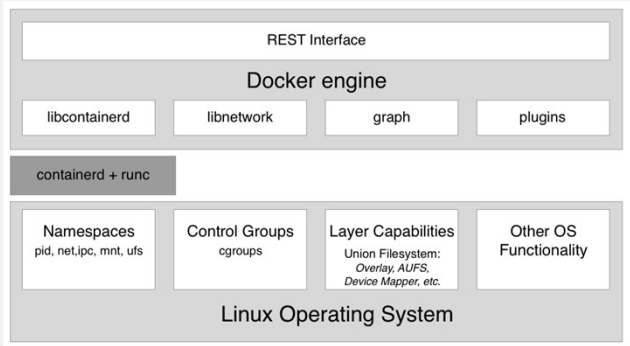
Container Runtime: `runc`

- `runc` is the low-level functionality of the container runtime
  - full support for Linux namespaces
  - native support for all security features available on Linux
    - SELinux
    - AppArmor
    - Seccomp
    - `cgroups`
- Spawns and runs containers according to the **Open Container Initiative** (**OCI**) specification
  - Containers are configured using bundles
  - A bundle for a container is a directory that includes a specification file named "config.json" and a root filesystem
  - The root filesystem contains the contents of the container



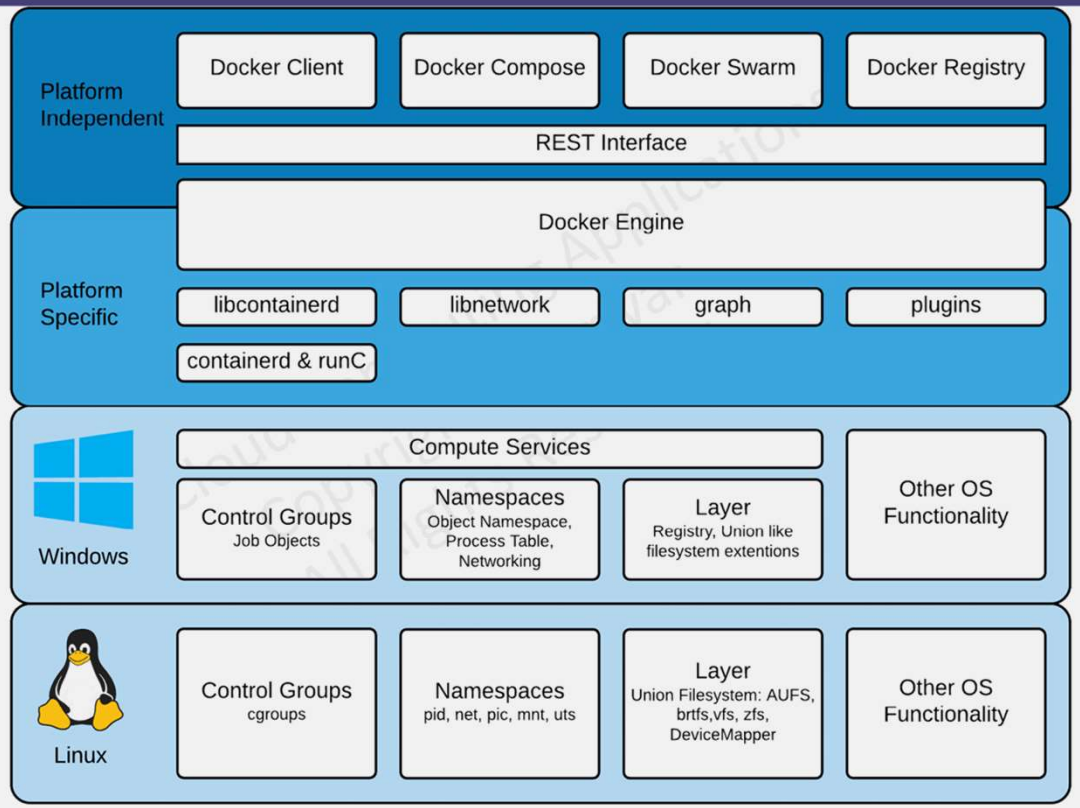
Docker Engine

- **Docker engine** provides additional functionality on top of the container runtime
  - E.g. network libraries or support for plugins.
- Provides a REST interface over which all container operations can be automated
  - The Docker command-line interface is one of the consumers of this REST interface



Docker Overview

- Container Orchestration
- Windows Containers



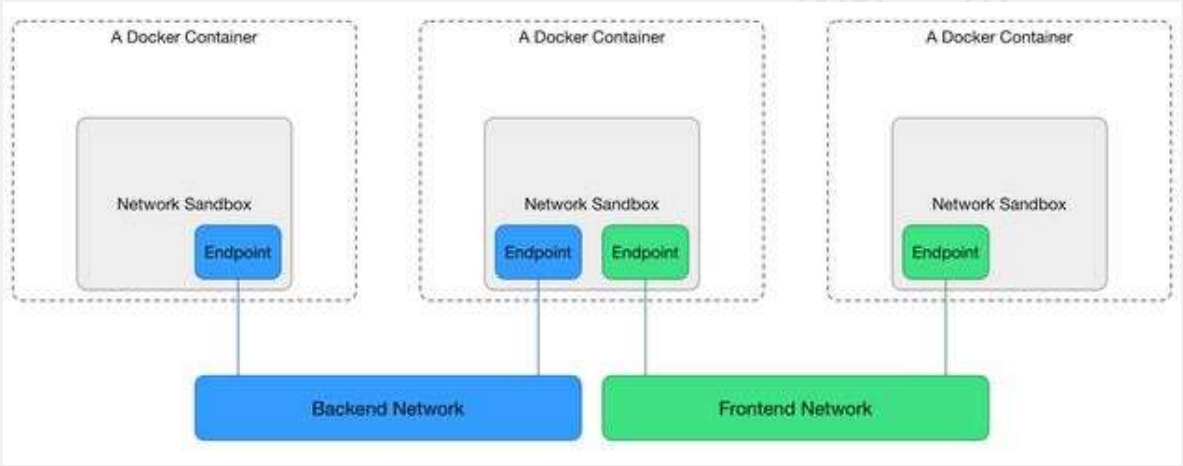
CLOUD COMPUTING APPLICATIONS

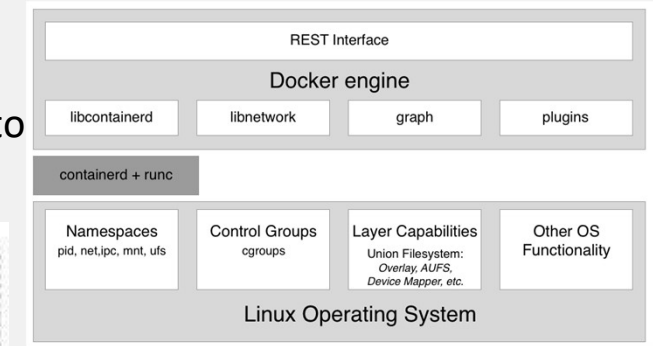
Containers: Networking

Prof. Reza Farivar

Container Network Model

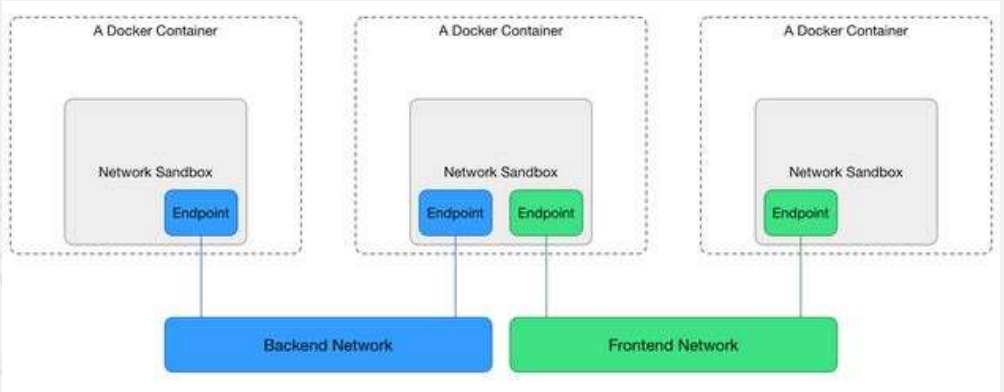
- Libnetwork implements Container Network Model (CNM)
- Formalizes the steps required to provide networking for containers while providing an abstraction that can be used to support multiple network drivers





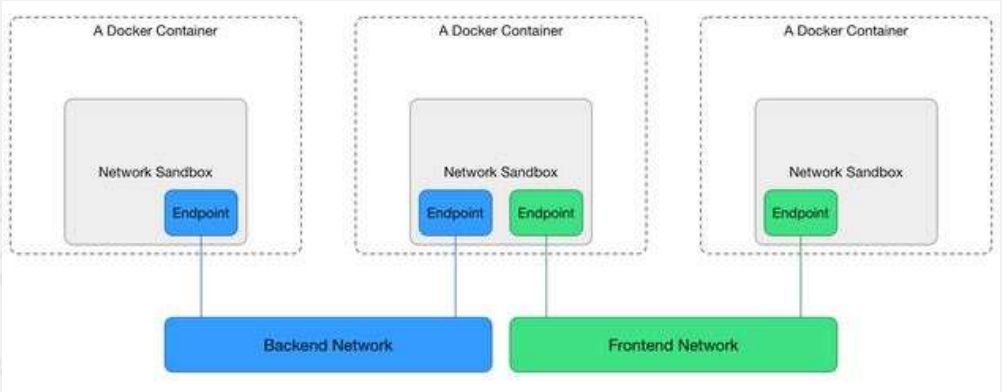
Container Network Model: Sandbox

- A Sandbox contains the configuration of a container's network stack.
- This includes management of the container's interfaces, routing table and DNS settings.
- A Sandbox may contain *many* endpoints from *multiple* networks.
- An implementation of a Sandbox could be a Linux Network Namespace, a FreeBSD Jail or other similar concept.
- Libnetwork implements sandbox in Linux through network namespace
- It creates a Network Namespace for each sandbox which is uniquely identified by a path on the host filesystem.



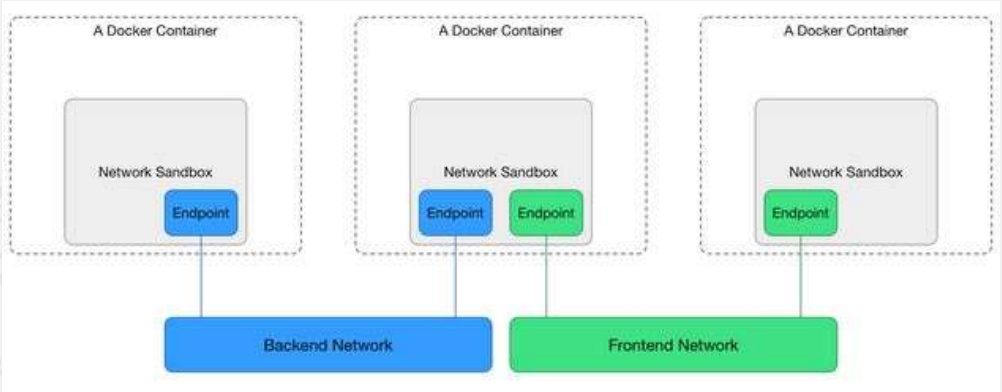
Container Network Model: Endpoint

- An Endpoint joins a Sandbox to a Network.
- An implementation of an Endpoint could be a `veth pair`, an Open vSwitch internal port or similar.
  - The **veth** devices are virtual Ethernet devices.
  - They can act as tunnels between network namespaces to create a bridge to a physical network device in another namespace
  - Can also be used as standalone network devices
  - **veth** devices are always created in interconnected pairs
    - One end is paced in one network namespace, and the other end in another namespace
- An Endpoint can belong to only one network and it can belong to only one Sandbox, if connected.
- Libnetwork delegates the actual implementation to the drivers which realize the functionality



Container Network Model: Network

- A Network is a group of Endpoints that are able to communicate with each-other directly.
- An implementation of a Network could be a Linux bridge, a VLAN, etc.
- Networks consist of *many* endpoints.
- Libnetwork delegates the actual implementation to the drivers which realize the functionality



Driver packages

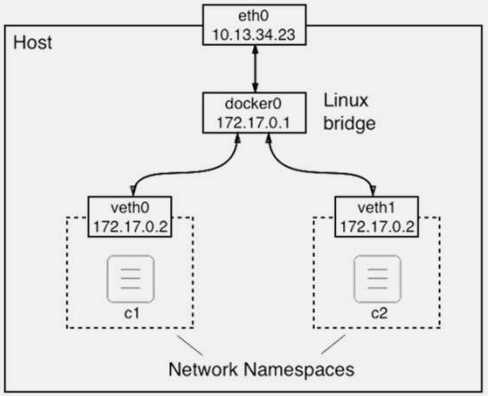
- Extension of libnetwork and provide the actual implementation of API
  - driver.Config
  - driver.CreateNetwork
  - driver.DeleteNetwork
  - driver.CreateEndpoint
  - driver.DeleteEndpoint
  - driver.Join
  - driver.Leave

Default Drivers in Docker Libnetwork

- Bridge: uses Linux Bridging and iptables to provide connectivity for containers
  - It creates a single bridge, called `docker0` by default, and attaches a `veth pair` between the bridge and every endpoint.
- host: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly
- Overlay: networking that can span multiple hosts using overlay network encapsulations such as VXLAN
  - Enable swarm services to communicate with each other
- macvlan: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network.
  - Docker daemon routes traffic to containers by their MAC addresses
- None
- Note: The type of network a container uses is transparent from within the container

Bridge Networks

- Bridge networks are usually link layer devices that forward traffic between networks
- In Docker, bridge network uses a software bridge allowing containers connected to the same bridge network on the same host
  - Isolating containers from other containers not connected to the bridge
  - For communicating with containers in other hosts, use overlay network
- The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other
  - `iptables` rules on Linux



Default Bridge Network

- When you start Docker, a default bridge network (also called `bridge`) is created automatically, and newly-started containers connect to it unless otherwise specified.
- Containers on the default bridge network can only access each other by IP addresses
  - User-defined bridges provide automatic DNS resolution between containers.
- The default bridge network is considered a legacy detail of Docker and is **NOT recommended** for production use

User-defined Networks

- Use --network to attach a container to a specific network
- Better isolation
- DNS resolution
  - On a user-defined bridge network, containers can resolve each other by the *container name* or alias
  - Much better than messing with /etc/hosts
- Containers can be attached and detached from user-defined networks on the fly
- Containers connected to the same user-defined bridge network effectively expose all ports to each other

Publishing ports

- From the container's point of view, it has a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details
- For a port to be accessible to containers or non-Docker hosts on different networks, that port must be *published* using the -p or --publish flag.

- ```
  $ docker create --name my-nginx  -
  -network my-net --publish 8080:80
  nginx:latest
  ```

Host : Container
  ```

IPAM: IP Address Management

- IPAM tracks and manages IP addresses for each network
  - Subnet
    - E.g. 172.17.0.0/16
    - All containers attached to this network will get an IP address taken from this CIDR range
  - Gateway
    - E.g. 172.17.0.1
    - Router for this network
- By default only egress traffic is allowed
  - Containerized applications can reach the internet, but they cannot be reached by any outside traffic

RFC 1918

```
~ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "232cda23c82c663ab4ea5297eb5e1e7a57b91cdd37d8ee63eb217
        "Created": "2021-02-28T03:04:28.953451723Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
```

Containers in the same namespace

- We can have multiple containers in the same namespace
- Processes in two containers in the same namespace can communicate through localhost
  - Compare to bridge networking, with two containers connected to the same network, where each host gets its own IP address
- Note that a sandbox (aka the Linux namespace) is connected to a network
  - We typically run each container in its own sandbox
  - But multiple containers can run in the same sandbox