Department of Information Engineering and Computer Science

Master's Degree in
Information and Communications Engineering

Fog and Cloud Computing

Lecturer

Prof. Domenico Siracusa

Students

Mattia Meneghin

Davide Zordan

Alessio De Paoli

Brando Chiminelli

Davide Parpinello

Academic Year 2021/2022

Index

1 Cloud Ecosystem

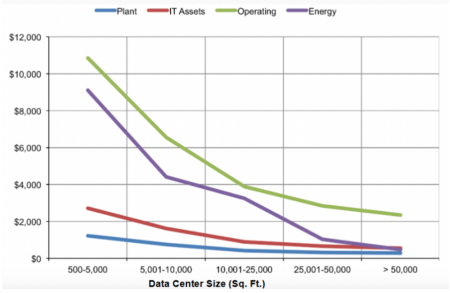
1.1 Introduction to cloud computing and fog computing

Global spreading of cloud computing: more need to store, access, process and share big amount of data. Examples from real-world: Walmart analyzed data for hurricane prediction; Local governments in US used data analysis for resource allocation decisions.

We need a safe and standardized way to access data.

How can we treat them? Using **distributed and scalable systems**.

Thus, large data centers are growing at a rapid rate (e.g. Microsoft Azure). Related to this, we can refer to economies of scale.
Simplifying, the **larger** the data center, the **less** management costs there are.



Many challenges arise while handling large distributed systems:

- lengthy procurement cycles
- lengthy deployment effort
- costly power and cooling
- costly system administration

and so on. So, we need to move from **Innovation to service** (IPS)
Cloud computing is the transformation of IT from **product** to **service**

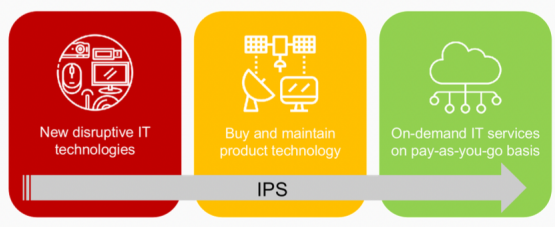


Figure 1.1: Cloud computing IPS

Most common requirements of a cloud computing system are:

- connectivity (move data around)
- interactivity (seamless interfaces)
- reliability (users not affected by failures)
- performance (not worse than what is already available)
- pay-as-you-go (no upfront fee or service)
- programmability (ease development of complex services to users)
- data management (large amounts of data)
- efficiency (cost and power)
- scalability and elasticity (flexible and rapid response to user needs)
- security (secure communications and services

Definition

Cloud computing could be seen as "computation done over the Internet"

A more formal definition (from NIST) is: Cloud computing is a model for **enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources** (e.g., networks, servers, storage, applications and services) that can be rapidly **provisioned** and **released** with minimal management effort or service provider interaction.

Five key cloud characteristics:

1. shared/pooled resources (resources are retrieved from a common pool)
2. broad network access (Available from anywhere with an internet connection using any platform)
3. On-demand automated reservation (Consumer can reserve resources as needed)
4. Rapid elasticity (Resources can be rapidly and automatically scaled up and down to satisfy customer demands)
5. Pay by use (Users pay only for used services, that are metered like a utility)

History

Some key events:

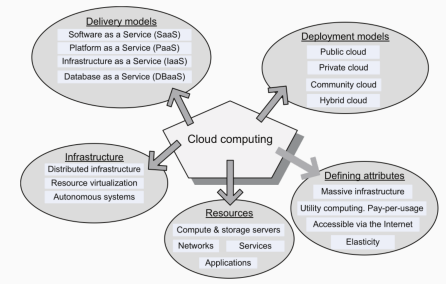
- 1960: John McCarthy wrote that "Computation may someday be organized as a public utility"
- 1990: Grid computing idea
- 1999: Salesforce introduced CMR
- 2002: Amazon web services
- 2006: Amazon's Elastic Compute cloud (EC2)
- 2009: Microsoft Azure

So, is cloud computing new? No, because technology is not new, but lately it is deeply changing thanks to new consumption and delivery model.

Obstacles and opportunities

1. Availability
   - Obstacle: business continuity is strategical to invest on cloud; causes of unavailability can be related to security attacks
   - Opportunity: Multiple cloud providers to provide continuity; elasticity and threat mitigation against attacks
2. Data lock-in
   - Obstacle: Many proprietary APIs for cloud computing
   - Opportunity: API standardisation
3. Data confidentiality
   - Obstacle: confidentiality of corporate data and auditability
   - Opportunity: encyrpted storage, firewall, auditability with additional layer beyond virtual OS
4. Data transfer capability
   - Obstacle: data-intensive applications, need of bandwidth
   - Opportunity: cost of WAN decreasing, keep data in cloud

1.2 Cloud deployment models and delivery models



NIST Reference Model For Cloud Computing

NIST is a model that defines how cloud computing is provided.

1. **Service Consumer**: Whoever accesses to a cloud services.
2. **Service Provider**: Entity responsible making a cloud service available.
3. **Broker**: Not Always necessary, it has the role of intermediary between who provides the service and those who access it. It also manage the use, performance and delivery of services.
4. **Carrier**: Who allows you to get your service from your service provider, so who provides connectivity and transport of services.
5. **Auditor**: Who conducts independent assessment of cloud services, information system operations, performance and security of the cloud implementation.

1.2.1 Virtualisation

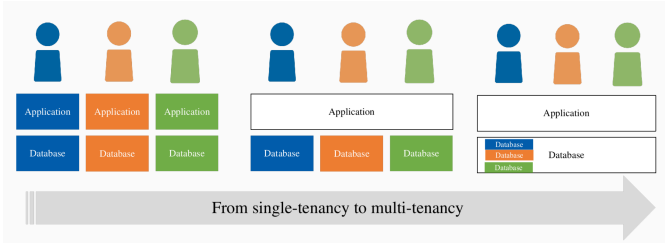
- **Abstraction of computing resources:** It hides the physical characteristics of computing resources: in which systems, applications, or end users interact with those resources. The main difference is the isolation level of the environment, every OS is isolated from each other.
- **Tenant:** Group of users sharing a common access with specific privileges to the sw instance

Single-Tenancy

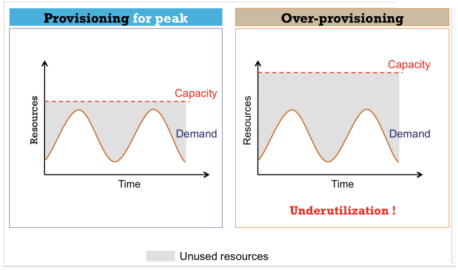
- Each customer has its own software instance
- Requires a dedicated set of resources to fulfill the needs of a customer
- n users $\rightarrow$ n instances

Multi-Tenancy

- A single instance of software runs on a server and serve multiple tenants
- Instance includes data, configuration, user management, and more
- Resources management and costs shared among tenants
- *1* instances $\rightarrow$ *multiple* users



- **Elasticity**
  - Capability to add and remove resources as you need, without losing time and money (*pay as you go* model)
  - It's difficult to predict peak utilisation, but with cloud computing it is possible to remove the risk of underprovisioning for expected peak demand (*transference of risk*)
  - Reduction of the risk of: **overprovisioning** (*underutilisation*) and **underprovisioning** (*saturation*)



- **Underprovisioning VS Overprovisioning**
  - **Underprovisioning**: Rejected user generate zero revenue and may not come back due to poor service
  - **Overprovisioning**: You spend more for what you actually need

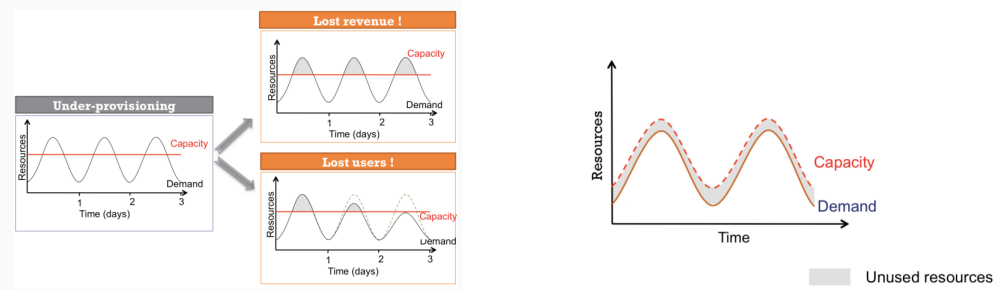


Figure 1.2: Elasticity and resource provisioning

1.2.2 Deployment Models

Deployment models represent a specific type of cloud environment (A deployment is a "*structure*" of your infrastructure), distinguished by ownership, size and access.

	Public Cloud	**Private Cloud**	**Community**	**Hybrid**
Consumer	General public or a large industry group	Exclusive use of an organisation	Organisations sharing common concerns (mission, policies)	Composition of two or more clouds (private, community or public)
Service provider	Organisation selling cloud services which manages the infrastructure	Consuming organisation or another party	either the organisations or a third party	
Resource location	All within the premises of the cloud provider	*On-premises*: Local cloud *Off-premises:* Remotely accessible private cloud offered by third party	either on-premises or off-premises	Remain unique entities but are bound together by standardised technology that enables data and application portability

1.2.3 Delivery Models

- Software-as-a-Service (**SaaS**): End-user has just to run the software for me (i can do some configuration, no more)
- Platform-as-a-Service (**PaaS**): Application developer gives me nice API and takes care of deployment
- Infrastructure-as-a-Service (**IaaS**): SysAdmin provides me computing resources to be used

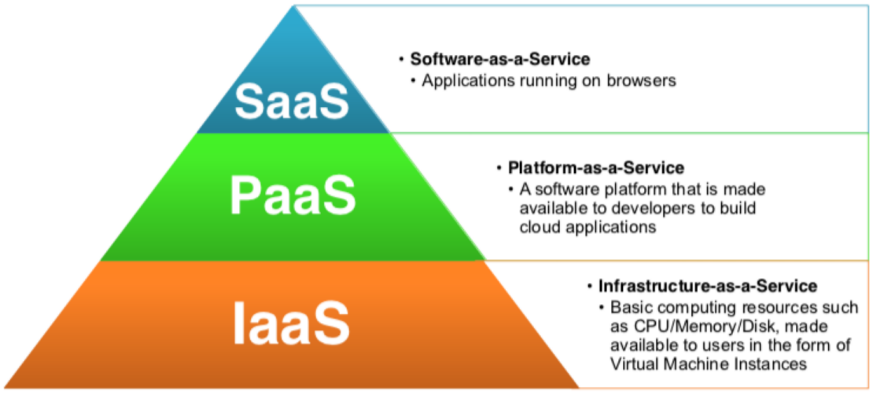


Figure 1.3: Delivery Models

Software-as-a-Service (SaaS)

- Applications are supplied by the service provider
- The user **does not manage or control** the underlying Cloud infrastructure, just customize the interface
- Services offered include:
  - Enterprise services such as: workflow management, communications, digital signature, customer relationship management (CRM), desktop software, financial management, geo-spatial, and search
- Not suitable for real-time applications or for those where data is not allowed to be hosted externally
- *Examples*: Gmail, Dropbox, Office 365, Netflix

Platform-as-a-Service (PaaS)

- Allows a cloud user to deploy consumer-created or acquired applications using programming languages and tools supported by the service provider
- User:
  - **Has control** over the deployed **applications** and, possibly, **application hosting environment configurations**
  - **Does not manage or control** the underlying Cloud **infrastructure** including network, servers, operating systems, or storage
- *Examples*: Aws Elastic Beanstalk, Windows Azure, Sap Cloud, Google App Engine

Infrastructure-as-a-Service (IaaS)

- Infrastructure is compute resources: CPU, VMs, storage, etc
- User:
  - Able to **deploy** and **run** arbitrary software, which can include **OS and applications**
  - Does not **manage or control** the underlying Cloud **infrastructure** but **has control over operating systems, storage, deployed applications**, and possibly limited control of some networking components, e.g., host firewalls
- Services offered by this delivery model include: server hosting, storage, computing hardware, operating systems, virtual instances, load balancing, Internet access, and bandwidth provisioning
- *Examples*: OpenStack, Amazon EC2, Google Compute Engine

Responsibility Sharing Between User and CSP

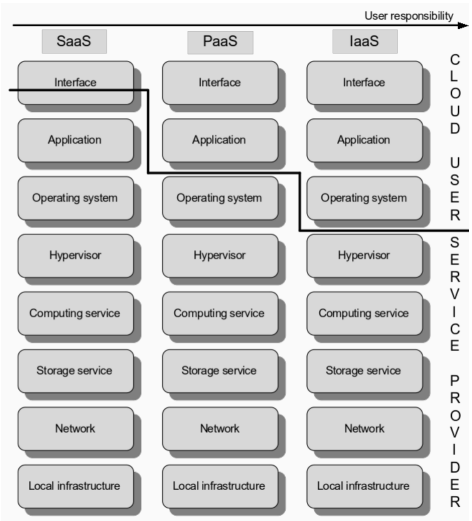


Figure 1.4: Responsibility Sharing Between User and CSP

1.3 The market of cloud/fog solutions

- **Amazon** is leader in IaaS
- **Google** focuses on SaaS and PaaS
- **Microsoft** main involvment is with PaaS
- Many open-source cloud computing platform (Eucalyptus, **OpenStack**..)

1.3.1 AWS

AWS is a IaaS cloud computing service launched in 2006 by Amazon.

- The infrastructure consists of compute and storage servers interconnected by high-speed networks and supports a set of services (user can install application of his choice)
- In each region there are several availability zones interconnected by high-speed networks. An availability zone is a data center consisting of a large number of servers. Regions do not share resources

Instances

An instance is a virtual server with a well specified set of resources User choses:

1. Region and the availability zone where this virtual server should be placed
2. Instance type (from a limited menu)

An instance is provided with a **DNS name**, mapped to:

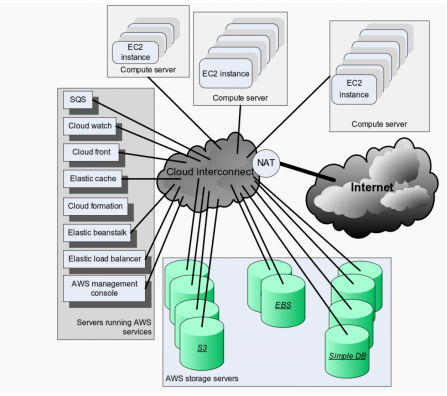
- private IP address for internal communication
- public IP address for communication outside the internal Amazon network (it is assigned for the lifetime of an instance
- NAT maps external IP addresses to internal ones
- an instance can request **elastic** IP rather than public (elastic: static public IP address allocated to an instance from the available pool of the availability zone)

What AWS does

1. Retrieves the user input (choice) from the front-end
2. Retrieves the disk image of a VM from a repo
3. Locates a system and requests hypervisor to setup a VM
4. Invokes DHCP and sets up MAC and IP addresses for the VM

User can interact with AWS through AWS Management Console, SDK libraries or raw REST requests.

AWS infrastructure



Examples of AWS

- **AWS Management Console** (provides access services offered by AWS)
- Elastic Cloud Computing (**EC2**): web service for launching instances of an application under several operating systems
  - EC2 imports VM images from user environment to ain instance through VM import
  - EC2 instances boot from an Amazon Machine Image stored in S3
  - User can both access images provided by Amazon, customize and store them in S3.
  - Instance types differ from memory, CPU, cost.
- Simple Storage System (**S3**): service designed to store large objects (stored in a bucket and retrieved via unique key). It supports minimal set of function (write, read, delete). There are authentication and MD5 mechanisms.
- Elastic Block Store (**EBS**): provides persisten block level storage volumes for use with EC2. Suitable for databases, file system, raw data. EC2 may mount different EBS but a EBS volume cannot be shared among different E2x instances
- **SimpleDB**: non-relational data store, supports high-performance web applications
- Simple Queue Service (**SQS**): Hosted message queues are accessed through standard SOAP and Query interfaces
- **CloudWatch**: monitoring infrastructure to collect and track metrics for optimizing the performance of applications and for increasing the efficiency of resource utilization

1.3.2 Google SaaS

- Gmail (hosts Emails on Google servers and provides a web interface to access the Email)
- Google Docs (web-based software for building text documents, spreadsheets and presentations)
- Google Calendar (a browser-based scheduler)
- Google Maps (web mapping service) and so on

1.3.3 Google PaaS

- AppEngine - a developer platform hosted on the cloud
- Google Co-op - allows users to create customized search engines
- **Google Drive** - an online service for data storage
- Google Base - load structured data from different sources to a central repository

1.3.4 Microsoft PaaS and SaaS services

- **Windows Azure** - an operatying system with 3 components:
  - Compute - provides a computation environment
  - Storage - for scalable storage
  - Fabric Controller - deploys, manages, and monitors applications
- SQL Azure (cloud-based version of SQL)
- Azure AppFabric, formerly .NET Services - a collection of services for cloud applications

1.3.5 Open-source platforms for private clouds

- Eucalyptus - paid and open-source computer software for building Amazon Web Services-compatible private and hybrid cloud computing environments
- Open-Nebula - single enterprise-ready cloud computing platform for managing heterogeneous distributed data center infrastructures (more enterprise cloud model)
- **OpenStack** - free and open-source software platform for cloud computing, mostly deployed as IaaS, whereby virtual servers and other resources are made available to customers

2 Cloud Virtualisation

2.1 Virtualisation

Initially, servers have been installed across the company; then, moved to datacenters in order to preserve data and make management easier. It has been discovered that there were so many servers around, mostly idle and that we cannot consolidate multiple apps on a single server.

"One application per server rule"

- Due to the failure of popular OSes to provide:
  1. **Configuration/shared components full isolation** (for example app A requires library v1.0 and app B v2.0, or A certified on OS version X and B on version Y)
  2. **Temporal isolation for performance predictability** (for example if A consumes lot of CPU, performance of B will be affected, same with network traffic)
  3. **Strong spatial isolation for security and reliability** (for example if A crashes, it may compromise B)
- Another reason: Moore Law valid with number of CPU cores and not anymore with operating frequency (clock)
- **Results:** Huge amount of servers, massively underutilized, and consuming a lot of electrical power (A lot of CPU cores available, but not used because of the "one app per server" rule)

Computing virtualisation in a nutshell

Computing Virtualisation is a **flexible way to share hardware resources** (e.g. CPU, memory, I/O) **between different (un-modified) operating systems**

Virtualisation broadly describes the **seperation of a service request from the underlying physical delivery** of that service (VMware definition)

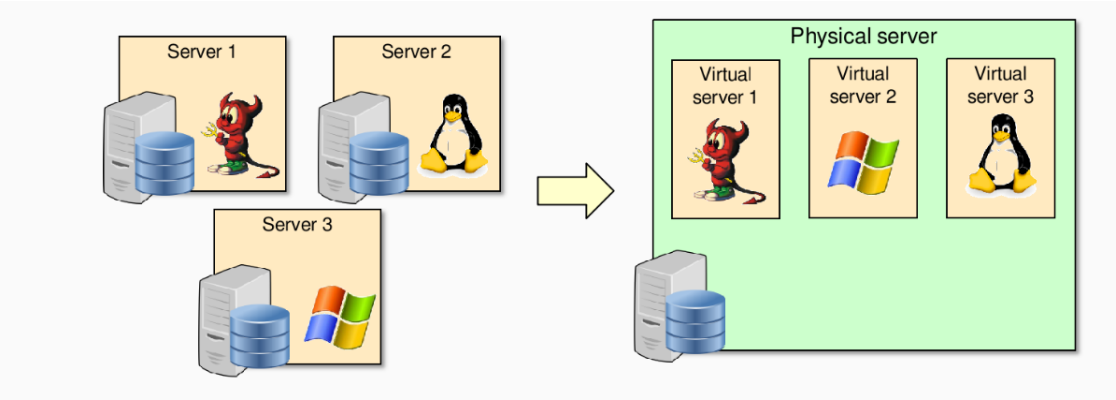


Figure 2.1: Computing virtualisation

It was proposed for the first time around 60's e.g with mainframes; then back in 90's as a new way to address new needs of reliability (datacenters). Currently it focuses on x86 architecture.

Advantages:

- **Isolation** (critical apps could run in different and isolated OS, malicious apps cannot compromise services running in other VMs)
- **Consolidation** (different OSes could run on the same HW at same time, saving HW resources and then minimizing operating costs)
- **Optimize energy consumption**
- **Flexibility and agility** (complete control over the execution information of all VMs, possibility to pause, restart, migrate, consolidate or give more resources)
- Possibility to duplicate running VM
- Disaster recovery
- Rapid spawn of new instances

Limitations:

- **Additional overhead** in running the same application (more requirements in terms of disks, memory and CPU for each App; however, usually acceptable)
- More difficult to handle heterogeneous Hardware (e.g offering the access to special components such as powerful GPUs to some applications

Usage scenarios

- **Serves virtualisation:** Several services, each one with their environment (e.g., their own OS) sharing the same hardware, with a given degree of isolation. **More important economically**. Main products: Vmware ESXi, Microsoft Hyper-V, Citrix XenServer, Linux KVM
- **Workstation/desktop virtualisation:** Possibility to reproduce the same hardware/-software configuration on several physical machines; easy to share HW facilities. Most of the products are **free**. Main products: Virtualbox, Vmware workstation, Linux KVM

Common Off The Shelf (COTS) Hardware

- Focus has now shifted from real servers to virtual servers (We can create dynamically a virtual server with the requested characteristics based on the actual necessities).
- **Consequence:** (corporate) users can buy tons of equivalent servers, with exactly the same hardware characteristics and aggregate them in datacenters or virtualize their resources in order to create specific set of virtual servers
- Computing hardware becomes a commodity

2.1.1 Initial Definitions

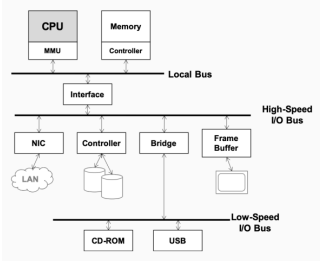


Figure 2.2: Computer System Organization

Layering in complex systems i

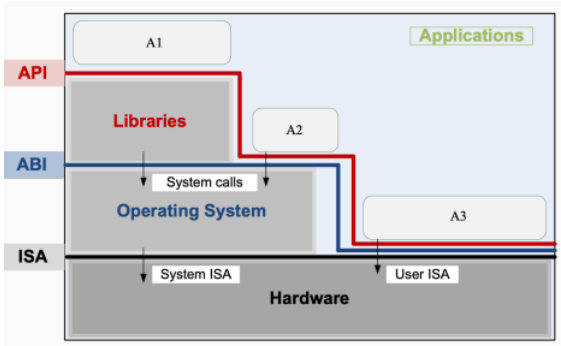
Layering – a common approach to manage system complexity

- Minimizes the interactions among the subsystems of a complex system
- Simplifies the description of the subsystems; each subsystem is abstracted through its interfaces with the other subsystems
- We are able to design, implement, and modify the individual subsystems independently

Layering in a computer system

- Hardware
- Software
  - Operating system
  - Libraries
  - Applications

Application complex system



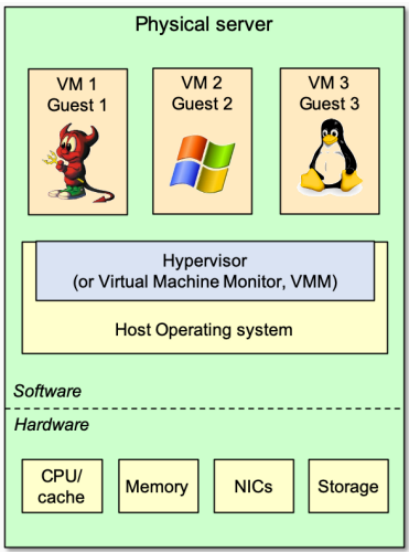
An application

- Uses library functions (A1)
- Makes system calls (A2)
- Executes machine instructions (A3)

Interfaces

1. Instruction Set Architecture (ISA)
   At the boundary between hardware and software
2. Application Binary Interface (ABI)
   Allows the ensemble consisting of the application and the library modules to access the hardware
   - ABI does not include privileged system instructions, instead it invokes system calls
3. Application Program Interface (API)
   defines the set of instructions the hardware was designed to execute and gives the application access to the ISA
   - Includes HLL library calls which often invoke system calls

Phisical Server



2.1.2 Definitions

1 - Virtual Machine (VM) Software emulation of a physical machine that executes OS+apps such as being in a physical one

2 - Host OS OS running on the physical machine, in charge of the virtualisation of the hardware, It can include the Hypervisor

3 - Guest OS OS running in the VM, which should not be aware of running in a virtualized environment

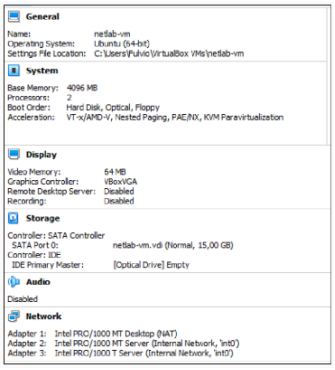
4 - Hypervisor (VMM)

- The Hypervisor (or Virtual Machine Monitor, VMM) is the software in charge of the virtualisation process It has to virtualize the hardware resources, such as CPU, memory, and other devices (e.g., NICs)
- Virtualisation means
  - Assigning a distinct set of resources to each VM (when possible) and guaranteeing that each VM cannot get access outside its boundaries
    - E.g., the memory can be partitioned in disjoint spaces and assigned to different VMs
  - Arbitering the access to shared resources, in case those cannot be partitioned
    - E.g., unique NIC, shared among all VMs
- In practice, often a stripped-down OS
  - Often Linux-based
  - A limited set of native drivers to manage the hardware
  - A virtualisation layer exports a set of "standard" devices to the upper-layer OS
    - Usually, we do not virtualize the latest video card
    - However, most important characteristics of the hardware can be exploited "natively"
    - Enable hosted OS to support a limited set of hardware
- The hypervisor may be attacked
  - Much smaller and more defendable than a conventional OS

5 - Virtual Hardware

- The VMM has to provide a "virtual hardware" to the guest VM, with the exact characteristics specified in a given hardware profile
  - The real hardware may have a little to do with the virtualized hardware
  - E.g., the server has one NIC from vendor X, and the VM profile specifies two NICs from vendor Y
- There is no straight relationship between how many virtual NICs you ask for and how real NICs you have.
- You could have n phisical NICs and n virtual NICs and they will not depend each other

Virtual Hardware



6 - Virtualised components

- Computing virtualisation represents the most common approach for Cloud Computing, according to the IaaS model
  - A flexible way to share hardware resources between different (un-modified) operating systems
- This requires the VMM to virtualize:
  - CPU
  - Memory
  - I/O (e.g., NICs, storage, video, etc.)

2.2 CPU Virtualization

Introduction

VMM assigns to the VM one or more CPUs to execute Guest OS

- Assume CPU architecture of *physical machine* is the **same** of the *virtualized one* (e.g., Intel x64)
- The Instruction Set Architecture (ISA) of the virtualized machine is the same (or a subset) of the physical ISA

In case the ISA is different we need to rely on "**emulation**" instead of "**virtualisation**"

- It requires the binary translation between two really different ISA, which is usually rather slow
  - e.g. Apple "Rosetta" transitioning from MIPS to intel CPUs
- Although some particular CPU architectures are able to perform this binary translation automatically, this feature is not available on mainstream architectures (e.g., x64)

Definitions

1. **Virtual machine**: "A virtual machine is taken to be an efficient, isolated duplicate of the real machine" (Popek and Goldberg, 1974)
2. **Virtual Machine Monitor (VMM)** must satisfy three characteristics:
   (a) It exports execution environments, VMs, essentially identical to the real machine
      - Applications and OSes can run unmodified

   (b) It efficiently executes the virtualised system
      - A "statistically dominant portion of the virtual processor instructions should be executed by the real processor", allows to be **efficent**

   (c) The VMM should have the **complete control** of real system resources
      - Each virtualised system should be able to access only hardware resources under the authorisation of the VMM

Types of CPU virtualisation in the x86 world

1. **Full Virtualisation** (Vmware workstation, Virtual Box) - 1999
    - Guest OS can run unmodified in the hypervisor
2. **Paravirtualisation** (Xen, Oracle OVM) - 2003
    - Guest OS need to be modified in order to be executed
3. **Hardware assisted Virtualisation (KVM)** - 2007
    - Hypervisors exploit hardware features now included in the CPU/chipset

x86 Hardware Privilege Ring

x86 defines 4 rings with different privilege levels:

- Ring 0: designed for the most privileged part of code, the OS kernel
- Ring 1-3: designed to run application and services with different levels of trustworthiness

Currently, only Rings 0 and 3 are used in modern OS.
The lower you go down less powerful you are. (at ring 3 very low amount of istructions avaiable)

x86 Hardware Privilege Ring and CPU virtualisation

Virtualisation can use "ring de-priviledging", a technique that runs all guest software at priviledge level greater than zero, but:

- 0/1/3 model: in the x86 architecture, some privileges with respect to memory accesses are granted to 0-2 rings, hence the guest OS can interfere with the VMM.
  You put VMM at ring 0, Guest OS at ring 1 and application on ring 3
- 0/3/3 model ("ring compression"): the above problem is solved, but the GuestOS is no longer protected from malicious applications, both Guest OS and Application on ring 3

There are 2 problems:

- When they thought of moving the Guest OS to the ring 1, they found that there are some privileged instruction that dont't allow the OS to run
- If the Guest OS is at the same ring of Application is no longer protected

Definition: priviledged instruction

- A **privileged instruction** is a CPU instruction that needs to be executed in a privileged hardware context
- It generates a trap if called when the CPU is running in the wrong context (e.g, at ring3)
- Examples of instructions that cannot be allowed in user-level applications
    - "HALT" instruction: a user-defined applications cannot halt a computer from running
    - I/O instructions: a user-defined application cannot directly interact with a specific I/O device, unless the OS guarantees exclusive access to it
- A privileged instruction can't be executed by a guest OS running at a ring greater than 0

Definition: sensitive instruction

- A **sensitive instruction** is CPU instruction leaking info about physical state of processor
- In order to be virtualizable, all sensitive instructions of a CPU must be privileged

2.2.1 Traps

What is trap?

- When CPU is running in user mode, some internal or external events, which need to be handled in kernel mode, take place
- Then CPU will jump to hardware exception vector (you have an interrupt), and execute system operations in kernel mode (Move out all the data).

When does a trap occur?

- **Exception**
  - Invoked when *unexpected error* or system malfunction occur
  - For example, execute privilege instructions in user mode
- **System Call**
  - Invoked by *application* in user mode
  - For example, application ask OS for system I/O
- **Hardware Interrupts**
  - Invoked by some *hardware events* in any mode
  - For example, hardware clock timer trigger event

A note on system calls implementation

- Traditional OS:
  - When application invokes a system call:
    - CPU will trap to interrupt handler vector in OS
    - CPU will switch to kernel mode (Ring 0) and execute OS instructions
- When hardware event:
  - Hardware will interrupt CPU execution, and jump to interrupt handler in OS
- Traditional way:
  - Userland code (e.g. glibc) generates a software interrupt (instruction INT xx)
  - The generic interrupt routine of the OS is started, which determines where to jump in the OS code to serve the above interrupt
  - Kernel jumps to the requested code, serves the interrupt, then returns to the caller (instruction IRET)
    - Requires to load and parse the content of several memory locations (rather slow)
- New way: use SYSENTER/SYSEXIT (or SYSCALL/SYSRETURN in x64)
  - Userland code writes the memory address of the target kernel routine in a specific register, then calls SYSENTER and kernel runs in a very fast transition

2.2.2 Trap & Emulate (T&E) paradigm

Model proposed by Popek & Golberg in 1974

- Guest OS executed in an unpriviledged domain
- When Guest OS executes a privileged instructions, processor launches a trap that is intercepted by the VMM
- VMM emulates the instructions for the guest OS (if legitimate) and than gives control back to guest OS execution

What does VMM do with traps that occur within the virtual machine?

- If trap is caused by an application, pass trap to the guest OS
- If trap is caused by guest OS, handle the trap by adjusting the state of the VM

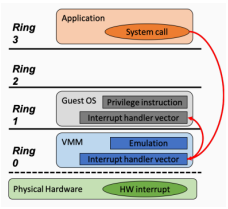
Need to require operating system support

- All traps and exceptions originating inside the VM must be handled by the VMM
- Most of the time guest apps and guest OS simply use the physical processor normally

When someone tries to run any instruction at the privilege ring, and this instruction is not available, it happens a **TRAP**, something like an interrupt. A context switch is when you move from a ring to another.

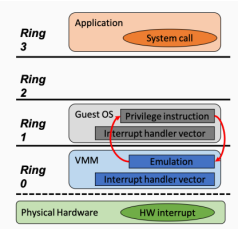
System Call

- CPU will trap to interrupt handler vector of VMM
- VMM jump back into guest OS
  - extra context switch
  - gets worse if the guest isn't able to handle the interrupt routing
  - time to execute single syscall could be 10x for the same call in the host OS



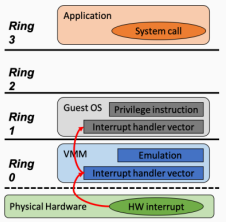
Privileged Instruction

- Running privileged instructions in guest OS will be trapped to VMM for instruction emulation
- After emulation, VMM jump back to guest OS



Hardware Interrupt

- Hardware make CPU trap to interrupt handler of VMM
- VMM jump to corresponding interrupt handler of guest OS



Traditional x86 architecture is not easily virtualisable

- In the T&E, each privileged instruction, if executed in an unprivileged domain, has to generate a (time-consuming) trap detectable by the hypervisor
- This is not always true for all CPU architectures
  - In particular, for those, like x86, conceived with a minimalistic approach and when the virtualisation was not popular at all
  - x86/x64 presents some sensitive instructions that do not trap if executed in a unprivileged level, such as POPA, POPF
  - The x86/x64 architecture is said to be "non-virtualizable"
- Therefore:
  - Several sensitive instructions are not detectable by the VMM
  - Consequentially, the VMM cannot emulate the correct behavior during the execution

Possible solutions

1. Parse the instruction stream and detect all sensitive instructions dynamically
   - **Interpretation (BOCHS, JSLinux)**: Old and slow technique,that emulating a single ASM instruction may originate an overhead of at least one order of magnitude
   - **Binary translation (VMWare, QEMU)**: No OS source modification, but performance overhead
2. Change the operating system: paravirtualisation (Xen, L4, Denali, Hyper-V)
   - Near-native performance, but OS modifications
3. Make all sensitive instructions privileged
   - Hardware supported virtualisation (Xen, KVM, VMWare): Intel VT-x, AMD SVM

Dynamic Binary Translation (DBT)

- Fully virtualized approach
  - Guest OS does not need to be modified
  - Enables x86/x64 virtualisation without HW assistance or modification to source code
- Idea: VMM dynamically translates x86 "non-virtualizable" ISA to virtualizable one at run-time
  1. **Dynamic**: translation is done on the fly during execution and interleaved with code execution
  2. **Binary**: VMM translates the binary code, not the source code
     - For better performance translation is done on blocks of code and not on single instructions
- Compatibility
  - No need of a specific HW support
  - No need of an OS modification
- Performance
  - Virtualisation overhead is significantly higher than other techniques

  - Translation speed can be improved by adopting caching techniques to recognize (and translate) significant instruction patterns
  - Several instructions or execution patterns (system calls) are significantly slower than real execution

Vmware workstation: T&E + DBT

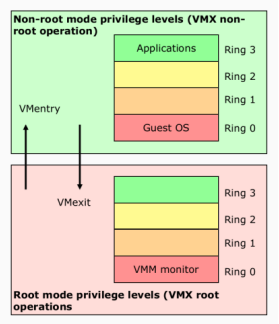
- Original VMware VMM combined a T&E engine with a system-level DBT
- Guest OS is executed in Ring 1 and VMM at Ring 0
- VMM inspects dynamically the code of the Guest OS and replaces non-trappable portions of the code (sensitive instructions) with "safe" instructions
  - In some cases, the resulting code requires the intervention of the VMM (e.g., interacting with the hardware)
  - In other cases, the resulting code is able to perform all its operations without the VMM intervention
    * e.g., access to GuestOS structures, while "normal" (privileged) instructions would return the equivalent structure present at VMM level (ring 0)

Para-Virtualisation (PV)

- Main idea: let the guest know that it is running in a VM
  - Guest OS knows that, in some cases, it has to give the control to the VMM
  - Para-virtualisation is no longer Full Virtualisation
- OS Guest is explicitly modified to be virtualizable, changing the interface provided to make it easier to implement
  - System calls are replaced with specific hypervisor calls (hypercall)
  - Non-virtualizable instructions are replaced with hypercalls (They do not produce traps)
  - No more trap&emulate (Also access to the page tables is para-virtualized)
  - ABIs of guest OS will not change, hence application could be executed without any modification
- Guest OS is explicitly deprivileged (It is executed in RING 1)
- Efficient mechanisms are introduced in the Guest OS kernel to ease the communication with the hypervisor:
  - Guest-to-Hypervisor: privileged instructions are replaced with synchronous para-virtualized equivalents (hypercalls)
  - Hypervisor-to-Guest: hypervisor could notify certain events asynchronously to the guest (Like interrupts, Guest OS could defer their handling)
- Performance
  - PV does not need to emulate motherboard or device bus
  - No longer dynamic translation; the OS has to be patched ahead of time
  - Implementation is simpler, hence much faster, than DBT
- Compatibility
  - Only modifiable OS could be virtualized
  - No Windows
  - OK Linux and BSD

Hardware-assisted virtualisation (HVM)

- Two problems still open:
  - Make easier the implementation of a hypervisor: even in PV, the VMM is still complex and tricky
  - Provide efficient x86 Full Virtualisation: several OS could not be modified and DBT is slow
- HVM aims at providing a solution to those problems by proposing an efficient "Trap & Emulate" approach to virtualisation thanks to an additional hardware support
- First implementation of Intel VT-X / AMD SVM (now AMD-Vi) in 2005
  - Functionally equivalent implementation in Intel and AMD CPUs, but incompatible
- Idea: avoid sensitive instructions, either because are "promoted" as privileged, or because the VMM can dynamically configure which instructions have to be trapped
  - Some instructions (e.g., INVD, Invalidate CPU Internal Caches) cause VM exits unconditionally and therefore can never be executed in VMX non-root operation.
  - Other instructions (e.g., INVLPG, Invalidate TLB Entries) and all events can be configured to do so conditionally, using the VM execution control fields in the VMCS
  - Processor is provided with a new running mode (VMX, Virtual Machine eXtensions)
- When VMX is turned on CPU enables two different running modes called "Operating Levels", equivalent to a new set of rings
  - Root Mode: higher class of privilege
  - Non-Root Mode: keeps existing four rings



- The VMM runs in fully privileged Root Mode
- Guest OS run in VMX non-root operating level in Ring 0
- Applications runs in VMX non-root operating level in Ring 3
- Intel's VTx allow a VMM to specify several conditions
  - According to this configuration, the actions attempted by a Guest VM will get "trapped"
- When a trapping condition is triggered, VMM takes the execution control and it emulates the correct behavior
- Transition between Root and Non-Root levels
  - VM entry: VMM to guest transition
  - VM exit: guest to VMM transition
- Registers and address space swapped in a single atomic operation
- Such transitions are main source of overhead

Detail: VMX instructions

- If sytem code tries to execute instruction violating isolation of the VMM or that must be emulated via software, hardware traps it and switch back to the VMM
- CPU enters non-root mode via the new VMLAUNCH and VMRESUME instructions, and it returns to root mode for a number of reasons, collectively called VM exits
- VM exits should return control to the VMM, which should complete the emulation of the action that the guest code was trying to execute, then give control back to the guest by re-entering non-root mode. All the new VM instructions are only allowed in root/system mode
- e.g, while in non-root mode, the 2 INT istruction may cause a switch from non-root/user to non-root/system, and the IRET instruction may return from non-root/system to non-root/user

Virtual Machine Control Structure (VMCS)

VT-x (SVM) introduces a new memory structure, VMCS (VMCB)

- It mirrors all register modifications needed to set a certain configuration in the guest OS
- Concretely it represents the control panel of the VM storing information about:
  - Guest state (*Load at VMEntry*)
  - Host state (*Load at VMExit*)
  - Control data (*What event to TRAP*)
- Introduced dedicated instructions to modify it (VMWRITE / VMREAD)

HVM: summary

- Hardware Assistance technology greatly simplifies VMM implementation
  - Transparent way to make "Full virtualisation"
  - HVM mode minimizes VMM intervention
  - No VM Entry/VM Exit for system call triggered by user applications
- HVM reduces virtualisation overhead
  - Round-trip time for VM Entry/VM Exit represents the most important source of virtualisation overhead
  - CPU manufactures are pushing for more efficient VM entry/exit to reduce the cost
- KVM provides hardware-assisted virtualisation

CPU virtualisation: summary

- DBT features the best compatibility but it is slow
  - Used mainly for legacy hardware without HVM
  - Still used nowadays e.g., in desktop virtualisation (Virtualbox)
- Paravirtualisation could achieve great performances but it could virtualize only a subset of OS
  - Fully implemented only for Linux and BSD guests
- HVM is the most used technique nowadays
  - Performance could vary a lot according to CPU generation

2.3 Memory Virtualization

Virtual memory

- Processes use virtual addresses
  - Addresses start at 0
  - OS lays processes down on pages
- MMU (Memory management unit)
  - Translates virtual to physical addresses → Maintains page table (big hash table) for the mapping
  - TLB (Translation lookaside buffer): cache of recently used page translations

Mapping virtual to physical

- OS maintains mapping of logical page numbers (LPNs) to physical page numbers (PPNs) in page table structures
- When logical address is accessed, hardware walks these page tables to determine the corresponding physical address
- For faster memory access, x86 hardware caches the most recently used LPN → PP mappings into TLB

Why virtual memory

- **Simplicity** - Every process gets illusion of whole address space
- **Isolation** - Every process is protected from every other
- **Optimization** - reduces space requirements

2.3.1 Translation Lookaside Buffer (TLB)

- TLB: fast, fully associative memory
  - Caches page table entries
  - Stores page numbers (key) and frame (value) in which they are stored
- TLB Sizes: 8 to 2048 entries

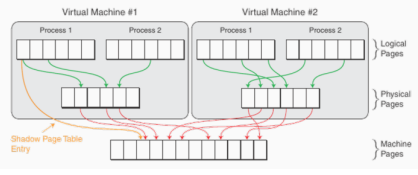
2.3.2 Memory Virtualisation: Shadow Page Table

An extra level of translation is required with VMs

- Guest virtual address → Guest physical address → Machine physical address

To avoid two address translations, a "Shadow Page Table" is introduced

- The shadow page table stores and keep track of the mapping between Guest logical address and Machine physical pages
- It is invisible from the guest point of view and it will be used by the CPU for a translation, when the guest is active
  - PPN→MPN maintained by VMM in internal data structures
  - LPN→MPN stored by VMM in shadow page table exposed to HW
  - Most recently used LPN→MPN in HW TLB



- The VMM is in charge of keeping the Shadow Page Table synchronized with the guest OS page table
  - OS will modify its ordinary page tables
- An important extra overhead is introduced in presence of page faults
  - An extra page table has to be kept synchronized when Guest update its own page tables→overhead for apps for which the guest frequently updates its page table

2.3.3 Memory Virtualisation: EPT/RVI

- In order to avoid Shadow Page Table overhead, Intel/AMD introduces Extended Page Table / Rapid Virtualisation Indexing
  - Different technologies implementing the same concept
- Traditional page tables translate LPN → PPN (guest table walk)
- VMM maintains PPN → MPN mappings in an additional level of page tables, called nested
  - Both the traditional (guest) page tables and the nested page tables are exposed to the CPU
  - When a logical address is accessed, the hardware walks the guest page tables as in the case of native execution (no virtualisation), but for every PPN accessed during the guest page table walk, the hardware also walks the nested page tables to determine the corresponding MPN

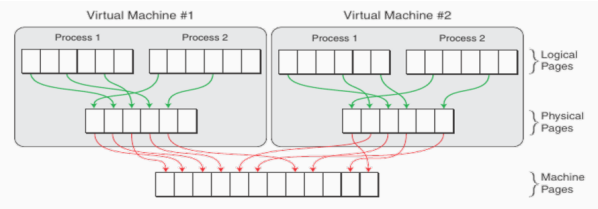


Figure 2.3: EPT/RVI

2.3.4 Memory virtualisation - Tagged TLBs

- Additional optimization of EPT
- Each TLB line has an identifier (Virtual Processor ID) for each virtual processor
- When a line has to be accessed, the ID prevents wrong access to other virtual processors cache lines
- This technique allows several virtual processors coexist on the TLB at the same time
  - The tagged TLBs eliminate the need for TLB flushes on every VMEntry and VMExit
  - Great importance with EPT/RVI where TLB page miss cause an important overhead

2.4 I-O Virtualisation

- Several techniques could be adopted for virtualizing I/O devices
  - Device emulation
  - Para-virtualized device
  - Direct assignment
- The choice of the technique that has to be used depends on:
  - The type of device
  - Shared/Dedicated to a single Guest OS

2.4.1 I-O virtualisation: device emulation

- VMM proposes to the Guest OS an emulated device, which implements in software an hardware specification
- Guest OS completely ignore that the device is emulated
- Guest OS will use the same driver used with an equivalent physical device
- VMM has to remap device communication with the physical device in real OS

Device emulation: pros and cons

- An approach simple and easy to set up
  - No need to install any dedicated driver to handle device I/O operations
  - A single physical device could be multiplexed with several emulated device
- However:
  - I/O operations are generally slower than physical ones, with higher latency
    - Particularly critical in case of devices with high I/O (NIC, disk)
  - I/O operations may increase substantially the CPU load
    - This represent an extra-work that is completely absent in case of a physical device

2.4.2 I-O virtualisation: para-virtualized device

- Guest OS is enriched with dedicated drivers
- Similar to para-virtualisation
  - The main difference is that para-virtualisation needs core modules of the kernel to be modified
    - This requires the kernel to be patched
- Instead, para-virtualized drivers could be added as external modules to the most part of modern OS
  - Here we require dedicated device drivers to be added/replaced

Para-virtualized: status

- PV drivers are provided from modern hypervisors for several kind of devices, such as:
  - Network cards
  - Disks
  - Graphical video cards
- In addition, special devices could be conceived to optimize resource consumption in virtualized OS
  - Memory Ballooning driver

A possible PV driver: memory ballooning

- The VMM, during the initialization of a VM, defines its memory size
- Normally, the VMM needs to allocate statically all the memory assigned to the guest
  - The actual utilization in the guest OS is unknown to the hypervisor
- In order to avoid to reserve memory that is not used by the guest, the hypervisor exploits a "Memory Ballooning" PV driver installed in the Guest OS
  - This driver provides to the hypervisor the information about current memory occupation of the Guest
  - This allows the hypervisor to over-commit memory allocation to several guests

2.4.3 I-O virtualisation: direct assignment

- Direct assignment allows the VM to directly communicate with the physical device
  - Also called "Device pass-through"
- Guest OS will handle the device with the traditional (dedicated) driver
  - No need for an additional driver in the host OS, as the device will be totally handled by the guest OS
- This technique requires the exclusive assignment of a device to a VM
  - No device multiplexing over several VMs
- "PCI passthrough" is an example of this technique

Direct assignment: problems

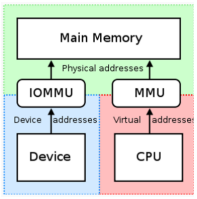
Despite its apparently simple approach, the "direct access" to hardware devices is very complex indeed

- Direct Memory Access (DMA) cycles have to be performed on guest physical addresses
- Devices do not know the mapping between guest and host physical addresses for a particular VM
- This could potentially lead to a memory corruption
- Memory corruption can be avoided if the VMM or host OS intercepts the I/O operation and performs the correct translation
- But this is slow and can introduce a significant overhead in I/O operations

2.4.4 IOMMU

- Dedicated extension introduced by hardware manufacturers to boost and make direct assignment easier to be implemented in hypervisor

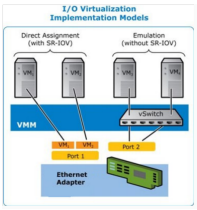
- Like the MMU, the IOMMU:
- Remaps the addresses accessed by the hardware according to the same table used to map guest-physical address to host-physical addresses
- DMA cycles will safely access the correct memory locations



2.4.5 SR-IOV (Single Root Input/Output Virtualisation)

- The PCI-e standard defines SR-IOV as a mechanism to allow several directly assigned devices to be shared among VMs (Specially designed for network cards)

- SR-IOV defines the possibility from the device to present several virtual devices, "virtual functions" to the OS
- The VMM will directly assigning a virtual function to a VM
- The hardware will handle itself the device multiplexing



2.4.6 I/O virtualisation: summary

- Device emulation is slow and computationally expensive but offers a great flexibility
  - Used for devices that are not critical in terms of performance or legacy OS
- Para-virtualisation is flexible and faster even if still computationally expensive
- Direct assignment is the fastest and less expensive technique but requires exclusive allocation of the device
  - In case of a network card, SR-IOV mitigates the problem of exclusive allocation of the device

2.5 Hypervisor Architectures

Traditionally, hypervisors presents mainly two architectures, type-1 and type-2:
They follows different design criteria:

- Performance
- Easiness of deployment & utilization
- Hybrid approaches are possible as well

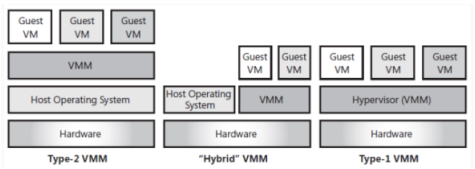


Figure 2.4: Hypervisor Architecture Type 1 VS Type 2

2.5.1 Hypervisor Architecture: Type 1

The hypervisor runs directly on bare metal:

- No extra layers between Hypervisor and hardware
- Normally able to provide the best performance
- Installation and deployment may be cumbersome
  - The hypervisor is basically an Operating System where all the unnecessary components (e.g., printing, etc.) are stripped down
  - The problem is having the proper drivers for all your hardware

Type 1 - Popular implementations

- Vmware ESXi
- Xen
- Microsoft Hyper-V
  - When enabling Hyper-V on your Windows 10 desktop, your operating system will no longer executed on the bare hardware, but as VM
  - Hence, running a further hypervisor (e.g., VirtualBox) in Windows would not be a great choice, as we are asking the system to use nested virtualisation
  - Instead, when Hyper-V is enabled, any further virtualized OS executed by Hyper-V would work in parallel with your Windows 10

2.5.2 Hypervisor Architecture: Type 2

- The hypervisor runs on a host OS, as a normal application
  - Installation and VM creation are simple tasks
  - Normally less performing than type 1
- Available implementations:
  - VirtualBox
  - VMware workstation

Hypervisor architectures - Hybrid approach

- The hypervisor is now implemented in the OS kernel
  - The host OS is itself the hypervisor, thanks to a specific component
- Normally simple to install and deploy
  - Drivers and support coming from the mainstream OS
- Very good performance, KVM is an example of hybrid approach

2.6 OS Level Virtualisation

2.6.1 Full virtualisation with VMs: pro and cons

Advantages

- Compatible with existing applications
- Support for different operating systems
- Each application (i.e., VM) can have its own execution environment(Kernel version, libraries)
- Excellent isolation, backed by hardware mechanisms (CPU, memory)

Problems

- Overhead for the necessity to execute the guest OS
  (Memory [hundred of MB], CPU)
- Necessity to configure and keep up-to-date each instance of guest OS
  ("Operations" are a non-negligible cost in real life)
- OS booting time (tens of seconds or more) may not be acceptable

2.6.2 Going cloud native

A recap on the historical perspective:

- First, we had individual servers (with Operating Systems, applications, et)
- Then, we had datacenters, with many servers that look like "cattle"
- But do we really need different operating systems?
  - Required for historical reasons
  - Required if we want to support real users (desktop environment)
  - Required if we want to support different hardware (e.g., peripherals, etc.)
- In cloud, we want to have the same commodity hardware, we don't have desktop environments, hence we can achieve greater operational efficiency if we reduce the number of OS to just one: Linux

2.6.3 Lightweight Virtualisation

Lightweight virtualisation - Idea behind

- Create a system that can guarantee the nice properties of computer virtualisation (scalability, elasticity, isolation) but that consumes less resources
- Lightweight virtualisation is appropriate when:
  - No need for a classical virtual machine
  - The overhead of a classical VM is not acceptable
  - We would like to have an isolated environment that is quick to deploy, migrate and dispose with possibly little or no overhead at all
  - We would like to scale both vertically (thousands of "lightweight VMs" on the same machine) and horizontally (deploy the "lightweight VMs" on many different machines available in a data center)

Lightweight virtualisation - What is it

- Use Operating System-level virtualisation or Application-level virtualisation instead of full hardware virtualisation
  - Both are software virtualisation technologies
- In case of OS-level virtualisation, the "hypervisor" is the Linux kernel itself
  - No longer required a dedicated hypervisor
- Virtual environments, replace classical VMs
  - Virtual environments are also know as virtual private servers, jails, containers
  - Virtual environments feature a given extent of resource management and isolation, usually less than what is achievable with VMs
  - Apps are executed inside these virtual isolated environments

Lightweight virtualisation - Requirements

- Fine-grained control of resources of the physical machine
  - Possibility to partition and control resources (e.g., CPU cores, RAM) among the different environments
- Security and isolation guarantees
  - Each virtual environment should be assigned to a different app/user, and avoid that a misbehavior in a virtual environment affects the others
- Possibility to manage the entire datacenter as an unique entity, such as with cloud toolkits
  - Even better, capability to integrate lightweight virtualisation with a cloud toolkit in order to have the flexibility to deploy VM, containers, etc., upon request

2.6.4 Possible technologies

- In theory, several different choices, but, in the end, a few answers
  - Linux containers (LXC) and LXC-based software
  - Other Operating Systems look irrelevant here
- Technologies actually used
  - Linux cgroups and namespaces
  - Linux Containers (LXC)
  - Docker

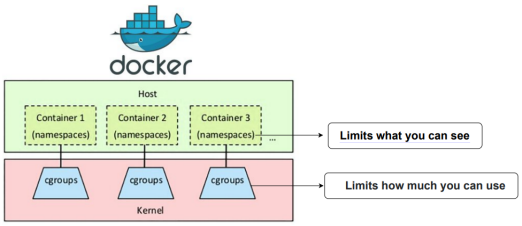
2.6.5 Process isolation: motivation

Community recognised need to implement strong process isolation in Linux Kernel

- Server running multiple services wants to be sure that possible intrusion on a service does not compromise the entire machine
- Necessity to safely execute arbitrary or unknown programs on your server
  - e.g., Amazon Lambda
  - e.g., code submitted by third parties, such as students, coding events (hackathon), continuous integration (automatic testing of new software releases)
- Requires strong process isolation, without adopting techniques such as hw virtualisation
  - e.g., a different operating system is not needed

2.7 Cgroups and Namespaces

- Originally developed to strengthen isolation between processes
  - Nothing to do with full virtualization
- Cgroups and namespaces can be leveraged to create a new form of lightweight virtualization, without the overhead associated to full virtualization
- Controllable properties:
  - CPU and Disk quotas
  - I-O rate and Memory limit
  - Network isolation
  - Check-pointing and live migration
  - File system isolation
  - Root privilege isolation



2.7.1 Linux Cgroups

- Linux kernel feature to limit, account, isolate or deny resource usage to processes or groups of processes
  - Represents the elementary brick that enables a fine-grained control to single processes and resources, providing a way to implement OS-level virtualisation
- Consists of two parts: **kernel feature** and **user-space tools** that handle the kernel control groups mechanism
- Example: CPU quota
  - The advantage of control groups over nice or cpulimit is that the limits are applied to a set of processes, rather than to just one
  - For instance, CoreOS, the minimal Linux distribution designed for massive server deployments, controls the upgrade processes with a cgroup: downloading and installing of system updates does not affect system performance

Linux Cgroup - Features

Resource limiting

- Groups can be set to not exceed a configured memory limit, which also includes the file system cache

Prioritization

- Some groups may get a larger share of CPU utilization, Disk I/O throughput, or Network bandwidth

Accounting

- Measure group resource usage, e.g. for billing purposes

Control

- Freezing groups of processes, their checkpointing and restarting

2.7.2 Linux Namespaces

- Feature of the Linux kernel, technically not part of cgroups, but highly related
  - Process groups separated to not "see" resources of a given class in other groups
  - Initial release in 2002, kernel 2.4.19, with "mount" namespaces and growing ever since
- Create distinct virtual environments e.g., networking, file system, and more
  - e.g., two namespaces have completely independent networking stacks, e.g., virtual interfaces, IP addresses associated to it, etc.
  - e.g., two namespaces have visibility on completely independent file systems, such as different the /etc folder, etc.

Linux Namespaces - Example

- Kernel has to create different independent instances of the same data structure
  - E.g., routing table for the network namespace
- Each object (e.g., process) is associated to a given namespace and can access to objects belonging to the same namespace
  - When the access to a given data structure is requested, the kernel uses the "namespaceid" to retrieve data from the proper structure

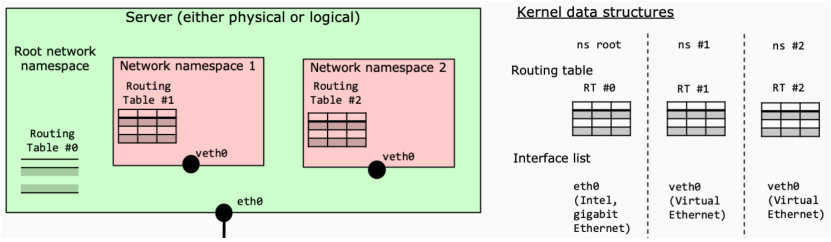


Figure 2.5: kernel partitioning among namespaces

Linux Namespaces - 7 Avaiable Types

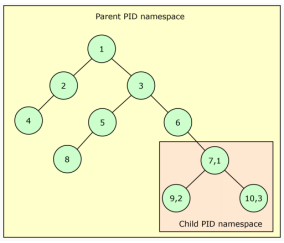
Within each type of namespace, we can create N different namespaces, e.g. within the network namespace, there can be namespace root, ns1, ns2, etc

Namespace	Constant	Isolates
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues
Network	CLONE_NEWNET	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name
Cgroup	CLONE_NEWCGROUP	Control groups

Figure 2.6: Namespace Avaiable Types

Linux Namespaces - Process Namespace

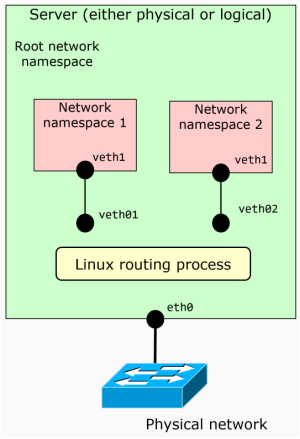
- In Linux, processes originate a single process tree
  - Each process has a parent. . . till init(1)
  - Processes may have children (when fork() is called)
- A process with enough privileges and under some conditions can inspect another process by attaching a tracer to it or may kill/suspend it
- PID namespace enables multiple "nested" process trees
  - Each process tree represents an entirely isolated set of processes
  - Processes belonging to a given process tree do not even know the existence of other parallel process trees, hence cannot inspect or kill processes in other process trees
- PID namespace allows a process to create a new tree, with its own PID 1 process
  - The first process remains in the original tree and knows about its child;
    in fact, processes in the parent namespace have a complete view of processes in the child namespace, as if they were any other process in the parent namespace
  - However, child becomes the root of its own process tree and it does not know anything about the originating process tree



- A single process can now have multiple PIDs associated to it, one for each namespace it falls under

Linux Namespaces - Network Namespace

- Network namespace allows two processes to perceive a completely different network setup
  - Interfaces, routing table, firewalling rules, etc.
  - Even the loopback interface is different
- Once a network namespace is created, we should create additional "virtual" network interfaces which span multiple namespaces
  - Virtual interfaces ("veth") are a network abstraction of a wire with two ends, in different namespaces
  - Veth allows traffic to cross the namespace borders and be delivered to another namespace
  - Bridging/routing process in root namespace enables traffic delivery to destination



- These commands establish a pipe-like connection between these two namespaces
  - Parent namespace retains the veth0-root device, and passes the veth0-ns device to the child namespace
  - Anything that enters one of the ends, comes out through the other end, just as we expect from a real Ethernet connection between two real nodes

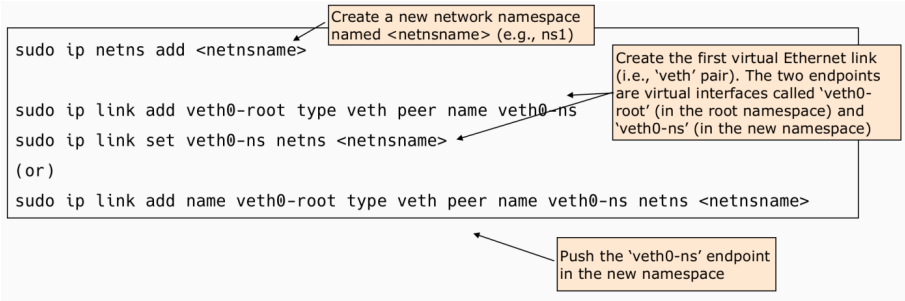
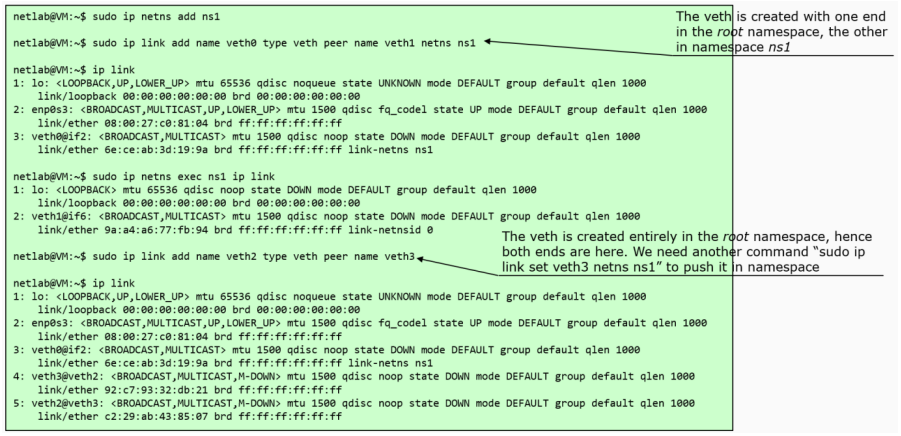
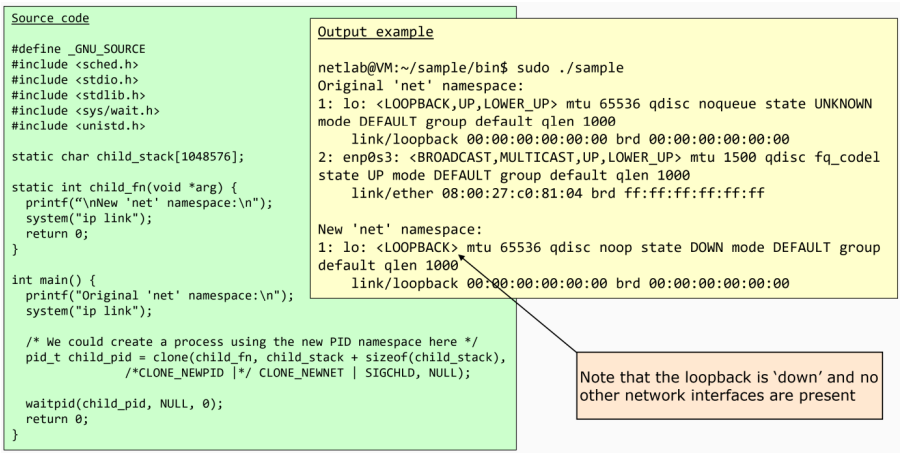


Figure 2.7: Network Namespace Cli

  - Both sides of this virtual Ethernet connection can be provided with IP addresses

- Names are arbitrary; names can be the same in different namespaces (e.g., two veth0 in two different namespaces)
- All network commands are still available in network namespaces, although with visibility limited to the namespace resources
- Relevant command:
  - *$ [sudo] ip netns exec namespace_name [command]*
- Example
  - *$ [sudo] ip netns exec* ***ns1 tcpdump***





Other namespaces in short

- Mount namespace enables the creation of a completely new file system, with the desired structure
  - E.g., disk partitions (and permissions, e.g., read-only), mount points, folders
  - Similar to chroot (empty root to populate), but more flexible (cloned data structure)
- UTS namespace is a very simple namespace that provides isolation of two system identifiers: the hostname and the NIS domain name
- User namespace allows a process to have root privileges within the namespace, without giving it that access to processes outside of the namespace
- IPC namespace creates private inter-process communication resources (e.g., System V IPC, POSIX messages) for the isolated process
- cgroup namespace provides a mechanism to virtualize the view of the "/proc/$PID/cgroup" file and cgroup mounts

2.7.3 Beyond process isolation: limitation of cgroups and namespaces

- cgroups and namespaces show some limitations beyond process isolation
- In fact, they provide an answer to the "virtualization" on a single server
  - Cannot be used, as is, to handle an entire datacenter (need to be integrated with CMS)
- Flexible (each feature, e.g., namespace can be tuned) but difficult to use
  - A lot of commands/scripting required to turn a full container on
- Cannot guarantee application portability, such as in case of VMs
  - VMs can be packaged and started on any server
  - Not an easy way to "package" an isolated app and make it running on another server
- Cgroups and Namespaces should be extended to handle a datacenter at scale

2.8 Containers

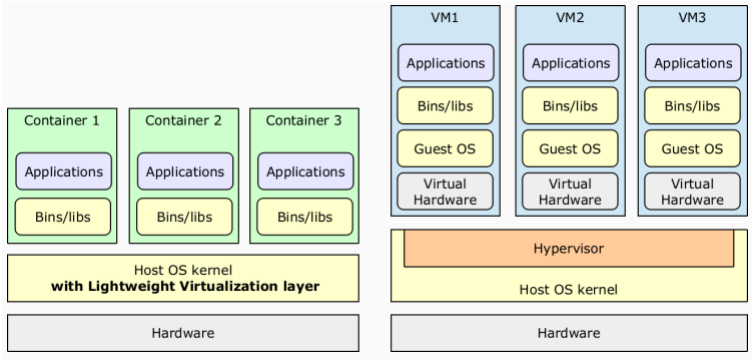


Figure 2.8: Containers VS HyperVisor

2.8.1 Containers - What They Do

- They group processes together inside isolated containers (to which different resources can be assigned)
- They share the same operating system as the host
- Inside the box they look like a VM (or better, a machine)
- Outside the box they look like normal processes (in the host, I see processes of all containers)

2.8.2 Containers - What They Don't Do

- They do not emulate hardware
- They do not run different kernels or OSs
- Security is not an out-of-the-box feature and it must not be taken for granted
  - still an appropriate level of security can be achieved

2.8.3 Containers vs VMs

- Depending on your view, containers are at the same time:
  - Infrastructure primitives (aka "lightweight VMs")
  - Application management and configuration systems
- An Infrastructure engineer will see them as the former; a developer will see them as the latter

Containers are:

- Faster than real VM (to boot, freeze, dispose and to orchestrate)
- Lighter than VM: less CPU, less memory, no virtualisation overhead (e.g., instruction emulation)
- Denser than VM: due to the inferior resource consumption
- Container virtualisation technology can practically achieve the same performances of native execution

VMs are:

- Provide better isolation (e.g., protect also from kernel exploits)
- Better security, due to the limited points of attacks (the hypervisor is usually very tiny)
- Enable the usage of different OSs

2.8.4 Containers - Characteristics

Agile application creation and deployment Increased ease and efficiency of container image creation compared to VM images

Cloud and OS distribution portability Run on Ubuntu, RHEL and anywhere else

Application-centric management abstraction: from OS on virtual hardware to application on OS using logical resources

Resource isolation Predictable application performance

Resource utilization High efficiency and density

2.9 Linux Containers LXC

2.9.1 Linux Container - Overview

- Containers provide **lightweight virtualisation** that allows to isolate processes and resources without the complexities of full virtualisation
- Linux container is an **OS-level virtualisation** method for running multiple isolated Linux systems (containers) on a single control host
  - The Linux kernel is shared across all containers
- LXC = **cgroups** + **namespaces** (+ some other stuff)
- No kernel patches required, vanilla kernel is ok
- Collection of kernel features that can be used to isolate processes in different ways and a userspace tool to use all of these features to create full-fledged containers
- Elementary features are still usable on their own, without LXC

2.9.2 Linux Container - Features

- Kernel namespaces (ipc, uts, mount, pid, network and user)
- Chroots
- Control groups (cgroups)
- Kernel capabilities
- Apparmor and SELinux profiles
- Seccomp policies

2.9.3 Linux Container - Why LXC?

Because it is easier than each single elementary component

For instance:

- Cgroups provides only **resource management**
- If we want also **isolation**, we need to add also *namespaces*
- If we want also **security**, we need to add *Apparmor* and/or *SELinux*
- But what about also live migration, etc.

LXC does not provide everything, but it provides several features:

- LXC allows to configure each container with the list of features it needs, specified in its configuration file
- We can assign different resources to different containers
- We can have different levels of isolation between different containers (and the host)

2.9.4 Linux Container - Limitation

Checkpointing and migration Missing features in vanilla linux kernel, we need to user other tools (CRIU), which are not yet 100% working

Resource isolation

- We must configure containers not to "see" resources
- Not always guaranteed; e.g., the resource quota of a container could be affected by the behavior of others

Portability A LXC created on a first server cannot be ported to other servers, while VMs are instead portable (at least across servers that have the same hypervisor), so portability isn't guaranteed.

2.10 Docker

2.10.1 Docker - Focuses on applications

- So far, developers experience an hard way to package applications for deployment
  - Package your app in multiple formats (e.g., .debian, redhat, etc)
  - Create multiple packages for each OS version (e.g., Ubuntu 19.10, Ubuntu 19.04, etc), as dependencies may be different and required libraries may be present in different versions
  - Be ready to create new packages as soon as a new OS flavor / version is released
  - Still, users may complain that your app does not start because of some mistakes in the packaging process
- Not even considering that there are different OSs out there (Windows, MacOS, etc)
- Docker focuses on applications, simplifying their deployment and execution by creating a lightweight, portable, self-contained "package" that runs anywhere
  - Similar to what "intermodal shipping containers" did in the area of goods transportation (ships, trucks, etc.)

2.10.2 Docker - In a Nutshell

Objectives

- Clean separation of environments (of different processes)
- Sandbox with defined resources (e.g., through cgroups and namespaces)
- Unified environment to handle applications (e.g. "run" command for all apps)
- Transparent and seamless networking (Docker bridge, DHCP, transparent NAT, etc)

What Docker is not

- Not a virtualisation engine (leverages existing primitives such as cgroups and namespaces) and no support for different kernels
- Does not leverage hardware primitives (e.g., CPU extensions)

Docker focuses on applications simplifying their deployment and execution

2.10.3 Docker - Deploys apps reliably and consistently

- If an app works locally, it works on the server (and in any other place)
  - Run side-by-side containers with their own versions of dependencies
- Your app runs everywhere, with the exact behavior
  - Regardless of the kernel version
  - Regardless of the host distro
  - Physical or virtual, cloud or not

Separation of concerns

Developer (inside the container)

- App
- Libraries and dependencies
- "Manifest" (e.g., used TCP/UDP ports)
  - Optional and not always embedded in the container
- Code
- Data

DevOps (outside the container)

- Logging
- Remote access
- Network configuration
- Monitoring

2.10.4 Docker - Docker vs LXC

- Docker is optimized for the deployment of applications, as opposed to machines
  - LXC focuses on containers as lightweight machines, basically servers that boot faster and need less RAM
- Docker containers are portable across machines, while LXC need to be rebuilt on the target machine
  - A Docker can run "anywhere" (although CPU architecture must match) with just one build, and is isolated from the host
- Docker has with a simplified (command line) interface and more powerful management tools
- Docker supports resource management and isolation, versioning, component re-use, automatic build and sharing

2.10.5 Docker - Additional features compared to LXC

- Application portability
- Union file system
- Automatic build
- Focus on running applications
- Versioning
- Component re-use
- Sharing (Docker server)
- Better documentation and ecosystem
- Integration with OpenStack, Kubernetes and all cloud vendors

2.10.6 Docker - Image and containers

Docker Image

- An immutable template for containers
- Can be pulled and pushed towards a registry
- Image names have the form [registry/] [user/] name [:tag]

Docker Container

- An instance of an image
- Can be started, stopped, restarted, etc
- Maintains changes within the filesystems
- New image can be created from current container state (although not recommended, use Dockerfile instead)
  - The default for the tag is *latest*

2.10.7 Docker - Registry

- Entity similar to a software repository, that keeps the available Docker Images
- Can be either public (e.g., Docker Hub, https://hub.docker.com/) or private
  - Images can be private (e.g., associated to a given user/group), even on a public repository
- Some commands:
  - Searching in the registry:
    - *docker search <term>*
  - Download or update an image from the registry (and cache locally):
    - *docker pull <image>*
  - Upload an image to the registry (keep it private or make it public):
    - *docker push <image>*

2.10.8 Docker - Images (locally)

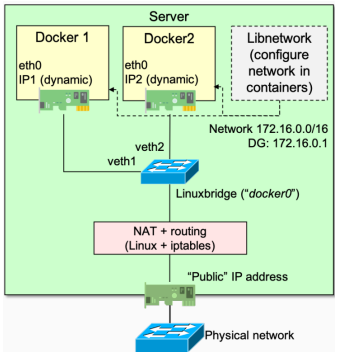
- "Pulled" images are cached locally and can be executed
- Some commands:
  - **List** downloaded images:
    - *docker images*
  - **Run** an image
    - *docker run [options] <image>*
  - **Delete** a local image:
    - *docker rmi <image>* (or)
    - *docker image rm <image>*

2.10.9 Docker - Docker run very common options

- When running a container, two options may be very useful:
  - Expose a TCP/UDP port of the container on the host
  - Mount a folder on the host (e.g., with persistent data) in the container file system (consider also network shares)
- Those are simple options of the docker run command
  - Publish port 80 from container nginx as port 8080 on host command:
    - *docker run -p 8080:80 nginx*
  - Mount local directory /html as directory /usr/share/nginx/html in the container nginx:
    - *docker run -v /html:/usr/share/nginx/html mynginx*
    - **Note 1**: /usr/share/nginx/html is where nginx expects HTML files
    - **Note 2**: "Mount" just means "make available"

2.10.10 Docker - Network

- By default, all containers are attached to a docker0 bridge (linuxbridge)
- Docker automatically implements a private network, with its IP address space and the appropriate routing table + iptables rules to enable Internet connectivity (outbound)
- Inbound connectivity is automatically provided to all connections started inside and if a port of the container is exposed outside
- Network operations are carried out through the "libnetwork" component, which implements the Container Network Model (CNM)
- This component decides the IP address to use within the container, and configures the inside endpoint of the veth with the above address (Address assignment doesn't use DHCP)



2.10.11 Docker Container

1 - Docker Container - some common commands

- Show running containers:
  - *docker ps*
- Show all containers (also terminated):
  - *docker ps -a*
- Show metadata of a container
  - *docker inspect <container>*
    - Returns JSON with all the info about the container
    - (e.g. current IP/MAC address, log path, image name, etc)
  - *docker inspect –format='.Image' <container>* (to show a specific metadata)
    - Shows only a specific metadata from the above JSON

2 - Docker Container - Commands to handle container lifecycle

Control your containers:

- *docker run <image..>* (start a new container from an image)
- *docker start <container..>* (start stopped container)
- *docker stop <container..>* (stop gracefully a container with SIGTERM signal)
- *docker kill <container..>* (kill running container by sending a SIGKILL signal)
- *docker rm <container..>*
  - Remove container from *docker ps -a list* State of container is lost, preventing user to save it to another container
  - docker start does not work on a "removed" container
  - Does not remove image, *docker run* still works

3 - Docker Container - Commands for interactions and debugging

- Run a command in an existing container, e.g start a shell:
  - *docker exec <container> <command>*
  - *docker exec -it <container> bash*
- See the logs (stdout) of the container:
  - *docker logs -f <container>*
- Copy files from and to Docker container:
  - *docker cp <source> <destination>*
  - *docker cp my_webserver:/etc/nginx/nginx.conf ∼/*

4 - Docker Container - Automatic build

- Docker allows to automatically create a container starting from its composing elements
  - E.g., a container can be created by compiling the application source code from scratch
  - E.g., another container can be created by using the package manager of Linux distro
- The recipe used to create the container is stored in a special file, called "**Dockerfile**"
- Developers are free to use make, maven, salt, debian packages, rpms, source tarballs, bash scripts, or any combination of the above, regardless of the configuration of the machines
- Note: if we start the build process with a strictly reduced base image (e.g., stripped down OS image), we can create an image that has only the software we need, avoiding the "default" software that is always installed in a vanilla copy of the operating system.

5 - Docker Container - File system

- Containers must replicate entire file system required for them to run
  - Isolation requirement: prevents container to access some files already on the host
  - Portability requirement: container should containing all files required to run
- This means that the size of a container (even minimal) can be rather huge, i.e., in the order to 100MB or more
  - Remember: only OS kernel is shared with the host; all the rest is duplicated!
- **Starting a process** (i.e., container) already present on the host may require a few milliseconds
- **Transferring the image** from a remote location can require several seconds (100MB is ∼1s on a 1Gbps network) => non-negligible delay in the container startup time

5.1 - Union file system works on top of the other file-systems

- It gives a single coherent and unified view to files and directories of separate file-system
- In other words, it mounts multiple directories to a single root (It is more of a mounting mechanism than a file system)
- UnionFS, AUFS, OverlayFS are some examples of the union file system (Docker, by default, uses OverlayFS)

5.2 - Properties of a Union File System

- Logical merge of multiple layers
- Read-only lower layers, writable upper layer
- Start reading from the upper layer then defaults to lower layers
- Copy on Write (CoW)
  - If process wants to modify existing data, OS copies data only for that process to use (all other processes continue to use the original data)
- Simulate removal from lower directory through whiteout file
  - File removed from the union mount directory would directly remove file from "upper" directory, simulate removal from "lower" directory by creating "whiteout" file
  - This file exists only in "union" directory, without physically appearing in either the "upper" or "lower" directories

5.3 - Union File Systems: advantages

- Only differences are stored
  - E.g., in case an application is modified, we do not need to copy also shared bin/libs
  - E.g., two apps can be created using same set of base "layers" (only apps are copied)
- Reduces
  - Disk footprint on the target host
  - Loading time when launching containers (each individual layer can be cached)

6 - Docker Container - Upgrading

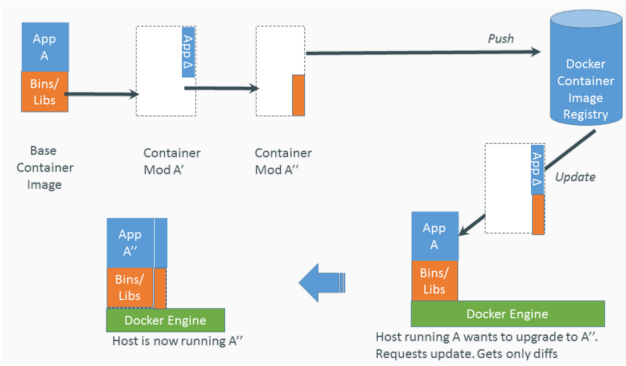
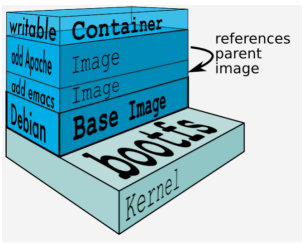


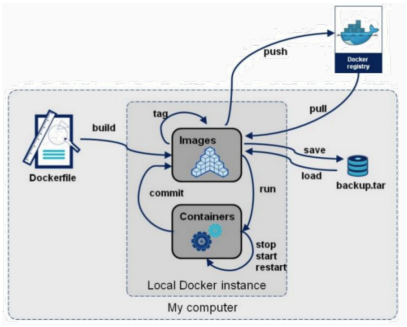
Figure 2.9: Upgrading a container

2.10.12 Docker - File System

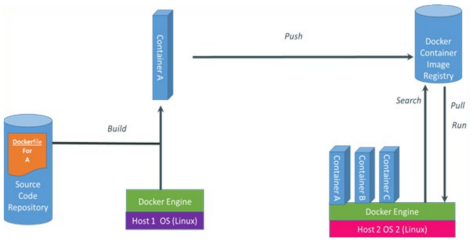
- Docker uses a layered file system
  - Default is OverlalyFS, through storage driver "overlay2"
- A complex image can result from the composition of different layers
  - Each layer can be either read-only or read/write
- Many containers can share the same set of base images



Docker - Lifecycle



Components and deployment workflow



2.10.13 Docker - Why is it used (and appreciated)

- Continuous delivery
  - Deliver software more often and with less errors
  - No time spent on dev-to-ops handoffs
- Improved security
  - Containers help isolate components of your system and provides control over them
- Run anything, anywhere
  - All languages, all databases, all operating systems
  - Any distribution, any cloud, any machine
- Reproducibility
  - Reduces the times we say "it worked on my machine"

2.10.14 Docker - Orchestration

DockerSwarm: orchestrator for Docker:

- It turns a pool of Docker hosts into a virtual, single host, handling the resources of the entire datacenter
- Loosing attraction in favor of Kubernetes

2.10.15 Docker - Compose

- Compose is a tool for defining and running multi-container Docker applications
- A specific YAML file is used to configure the application individual services
  - E.g., web frontend, database engine, storage
- Compose enables to create and start all the services from the same configuration, with a single command
- Works better if coupled with Docker Swarm
  - Also in this case, its functions are often carried out with Kubernetes

OS-level (lightweight) virtualisation: summary

- Lightweight virtualisation is definitely important and it is increasingly used in big enterprises
  - More efficient use of computing resources
  - Reduced operating costs (e.g., update of each individual kernel in different VMs)
  - Enable Multiple processes to coexist, with (strong) isolation properties
  - Several cloud companies are already offering commercial services
- Docker is a well-established technology for container virtualisation
  - Although it is more appreciated because has revolutionized the way software is packaged than for lightweight virtualisation

3 Cloud Networking

3.1 Data Center Networks

3.1.1 Clouds and Networks

- **Interconnectivity** supported by a continually evolving Internet made cloud computing feasible, **Communication** is at the heart of cloud computing
  - Cloud is built around a high-performance interconnect
  - Servers of a cloud infrastructure communicate through **high-bandwidth and low-latency networks**

Cloud workloads fall into four broad categories based on their dominant resource needs:

- CPU-intensive
- I/O-intensive
- Memory-intensive
- Storage-intensive

Communication bandwidth goes down and the communication latency increases the farther from the CPU data travels

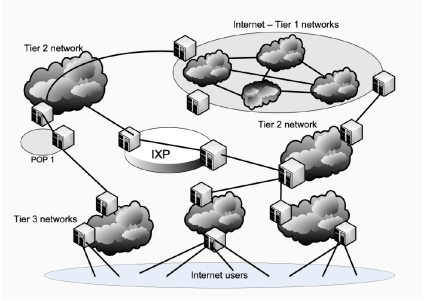
3.1.2 Relations between Internet networks

Three type of relations:

1. Peering - **two networks exchange traffic** between each other's customers freely
2. Transit - a network **pays** to another one to access the Internet
3. Customer - a network **is paid** to allow Internet access

Networks have been commonly classified as:

1. Tier 1 - can reach every other network on the Internet without purchasing IP transit or paying settlements (TOP LEVEL)
2. Tier 2 - Internet service provider who engages in the practice of peering with other networks, but who still purchases IP transit to reach some portion of the Internet
3. Tier 3 - purchases transit rights from other networks (typically Tier 2 networks) to reach the Internet (INTERNET USER LEVEL)



Three classes of networks, Tier 1, 2, and 3;
An IXP is a physical infrastructure allowing ISPs to exchange Internet traffic

3.1.3 The Transformation of the Internet

- Web applications, cloud computing, and content-delivery networks reshaped the definition of a network
- **Data streaming consumes** an increasingly larger fraction of the **available bandwidth** as high definition TV sets become less expensive and content providers, such a Netflix and Hulu, offer customers services that require a significant increase of the network bandwidth
- The **"last mile"** - the link connecting the home to the Internet Service Provider (ISP) network **is the bottleneck**
- Several initiatives (e.g. Google Fiber Project) to bring 1Gb/s access speed to individual households through **FTTH**

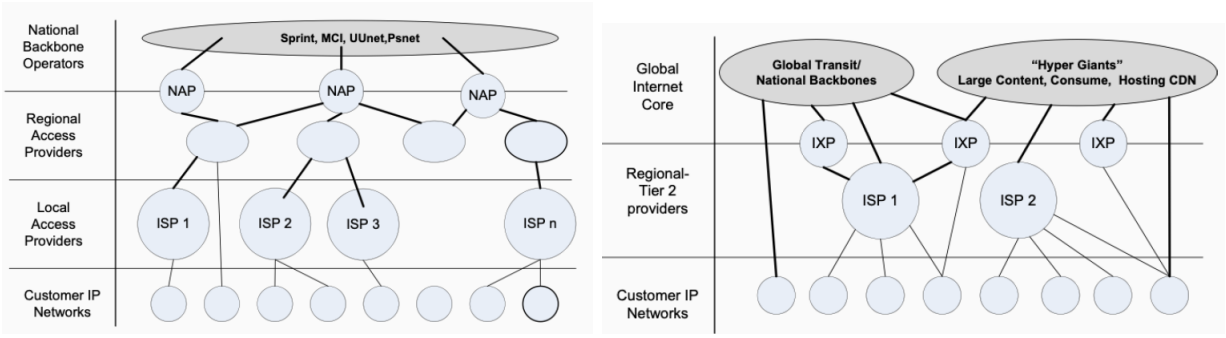


Figure 3.1: Transformation of internet: (a) prior to 2007, tier 1 networks; (b) CDNs and IP hosting

3.1.4 Data center networks: characteristics

- Huge scale (20k switches/routers)
- High bandwidth (10/40/100G and more)
- Very low round trip time (RTT, 1-10 microseconds) (really important)
- Limited geographic scope (little area)
- Limited heterogeneity (all the same kind of routers, switches, etc)
- Regular/planned topologies

Degrees of freedom

Single administrative domain: Control over network and endpoints, Control over placement of traffic source/sink.

In short: **full control**

- Can deviate from standards
- Can change addressing/-congestion control
- Can control routing (what traffic crosses which links)

3.1.5 Data center architecture: requirements

Applications:

- Online analysis of data (Tons of small flows (e.g. 55% of flows 3 % of bytes, 5% of flows 35% of bytes)).

  [mice flows (little) elephant flow(big)]
- Low-latency user interaction

Service model:

- One specific application (e.g. Google, Facebook)
- Cloud (computation/storage/infrastructure as a service)

Traffic directions:

- Outward (e.g., serving web pages to users)
- Internal computations (e.g. MapReduce for web indexing)

Workloads often unpredictable:

- Multiple services run concurrently within a DC
- Demand for new services may unexpectedly spike

3.1.6 Interconnection networks: basic concepts

- A network consists of nodes and links or communication channels
- An interconnection network can be:
  - **Non-blocking** if it is possible to connect any permutation of sources and destinations at any time
  - **Blocking** if this requirement is not satisfied
- Switches and communication channels are the elements of the interconnection fabric
  - **Switches**: receive data packets, look inside each packet to identify the destination IP addresses, then use the routing tables to forward it to the next hop towards its final destination
  - An n-way switch has n ports that can be connected to n communication links
- The degree of a node is the number of links the node is connected to
- **Nodes**: could be processors, memory units, or servers
- **Network interface of a node:** Hardware that connect it to the network
- Interconnection networks are distinguished by:
  1. **Topology** - is determined by the way nodes are interconnected
  2. **Routing** - routing decides how a message gets from source to destination
  3. **Flow control** - negotiates how the buffer space is allocated

1 - The topology of an interconnection network determines:

- **Network diameter** - the average distance between all pairs of nodes
- **Bisection width** - the minimum number of links cut to partition the network into two halves
  - When a network is partitioned into two networks of the same size the bisection bandwidth measures the communication bandwidth between the two
  - Full bisection bandwidth: one half of the nodes can communicate simultaneously with the other half

Static networks In the network there are direct connections between servers

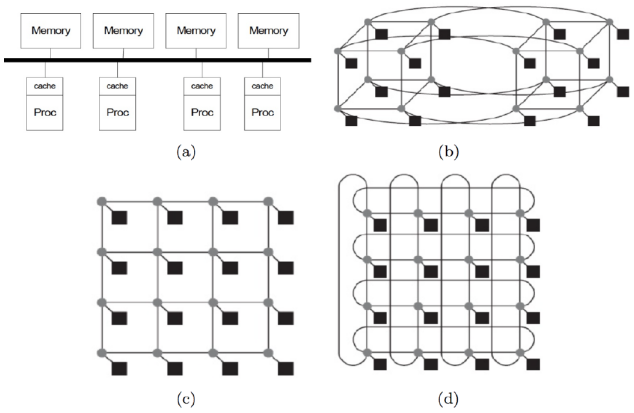


Figure 3.2: Static networks: (a) Bus; (b) Hypercube; (c) 2D-mesh; and (d) Torus

Switched networks Networks using switches to interconnect the servers

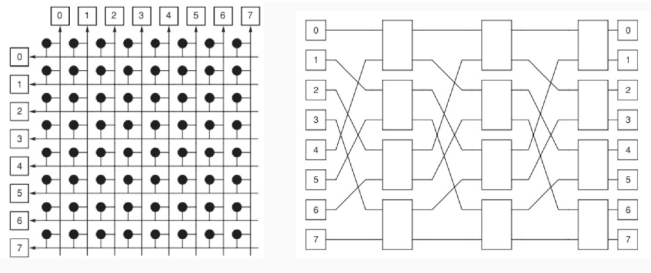


Figure 3.3: Switched networks

(Left) An 8 x 8 crossbar switch

- 16 nodes are interconnected by 49 switches represented by the dark circles

(Right) An 8 x 8 Omega switch

- 16 nodes are interconnected by 12 switches represented by white rectangles

3.1.7 Cloud interconnection networks

- 100000s **servers** placed in racks interconnected by
- 1000s **ToR switches** (Top-Of-Rack) interconnected by
- 1000s **aggregation switches** interconnected by
- 100s **core switches** interconnected to
- Internet/backbone network

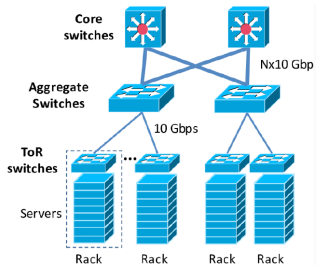


Figure 3.4: Network Inerconnection

Requirements for cloud interconnection:

- Scalability
- Low cost
- Low-latency
- High bandwidth
- Location transparent communication

3.1.8 Location transparent communication

- Every server should be able to communicate with every other server with similar speed and latency
  - Applications need not be location aware
  - Reduces the complexity of the system management
- In a hierarchical organization true location transparency is practically not feasible
  - Cost considerations ultimately decide the actual organization and performance of the communication

3.1.9 Costs

Amortized Cost	Component	Sub-Components
∼45%	Servers	CPU, memory, disk
∼25%	Power infrastructure	UPS, cooling, power distribution
∼15%	Network	Switches, links, transit
∼15%	Power draw	Electrical utility costs

Server costs

Uneven application fit

- Each server has CPU, memory, disk: most applications exhaust one resource, stranding the others

Long provisioning timescales

- New servers purchased quarterly at best

Uncertainty in demand

- Demand for a new service can spike quickly

Risk management

- Not having spare servers to meet demand brings failure just when success is at hand

Routers and Switches Costs

- The cost of routers and the number of cables interconnecting the routers are relevant components of the cost of interconnection network
- Better performance and lower costs can only be achieved with innovative router architecture
  - *Wire density* has scaled up at a slower rate than processor speed and wire delay has remained constant
- Router – switch interconnecting several networks
  - *low-radix* routers: have a small number of ports
    - Divide the bandwidth into a smaller number of wide ports
  - *high-radix* routers: have a large number of ports
    - Divide the bandwidth into larger number of narrow ports
- The number of intermediate routers in high-radix networks is reduced
  - Lower latency and reduced power consumption

3.1.10 Conventional data center network

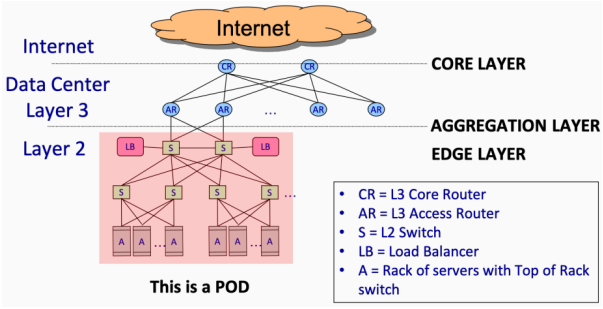


Figure 3.5: Conventional data center network

3.1.11 Layer 2 vs. Layer 3

- Ethernet switching (layer 2)
  - Pros
    - Fixed IP addresses and auto-configuration (plug & play)
  - Cons
    - Broadcast limit scale (ARP)
    - Spanning Tree Protocol
- IP routing (layer 3)
  - Pros
    - Scalability through hierarchical addressing
    - Multipath routing through equal-cost multipath
  - Cons
    - More complex configuration
    - Cannot migrate without changing IP address

3.1.12 Topologies: Clos networks, InfiniBand, Myrinet, Fat trees

Butterfly network

The name comes from the pattern of inverted triangles created by the interconnections, which look like butterfly wings

- Transfers the data using the most efficient route, but it is blocking, it cannot handle a conflict between two packets attempting to reach the same port at the same time

Clos

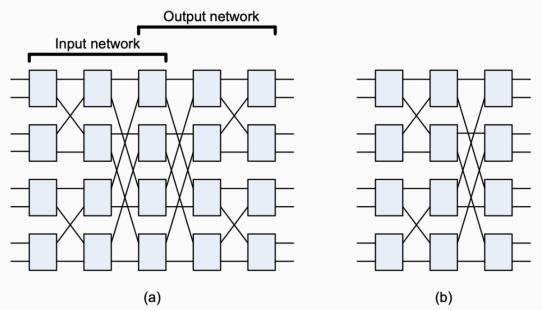
Multi-stage non-blocking network with an odd number of stages

- Consists of two butterfly networks
  - The last stage of the input is fused with the first stage of the output
- All packets overshoot their destination and then hop back to it
  - Most of the time, the overshoot is not necessary and increases the latency, a packet takes twice as many hops as it really needs

Folded Clos topology

The input and the output networks share switch modules

- Such networks are called **fat trees**:
  Myrinet, InfiniBand, and Quadrics implement a fat-tree topology



- (a) A 5-stage Clos network with radix-2 routers and unidirectional channels;
  The network is equivalent to two back-to-back butterfly networks
- (b) The corresponding folded-Clos network with bidirectional channels; the input and the output networks share switch modules

3.1.13 Fat Trees

Optimal interconnects for large-scale clusters and for Warehouse-Scale Computing (See 3.4 Picture)

- Servers are placed at the leafs
- Switches populate the root and the internal nodes of the tree
- Have additional links to increase the bandwidth near the root of the tree

Fat-tree DCNs

Fat Tree Data Center Networks design principles:

- The network should scale to a very large number of nodes
- The fat-tree should have multiple core switches
- The network should support multi-path routing
  - The equal-cost multi-path (ECMP) routing algorithm which performs load splitting among flows should be used

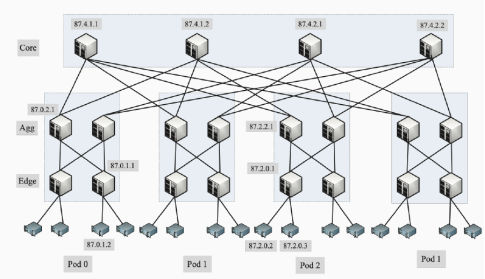


Figure 3.6: Fat-Tree DCN

Fat-tree interconnect consists of k pods

- Each pod has two layers and k/2 switches at each layer
- Each switch at the lower layer is connected directly to k/2 servers
  - Other k/2 ports are connected to k/2 of the k ports in the aggregation layer
- # switches: k(k+1), # connected servers k^2
  - There are $(k/2)^2$ paths connecting every pair of servers

Fat-tree interconnection network for k=4

- Core, aggregation and edge layers populated with 4-port switches
- Each core switch connected with one switch at the aggregation layer of each pod

The network has four pods

- Four switches at each pod, two at aggregation layer and two at the edge
- Four servers are connected to each pod

IP addresses of edge/agg switches: 87.pod.switch.1

- Switches are numbered left to right, and bottom to top
- Pod 2 switches (4 in number) have IP addresses 87.2.0.1 to 87.2.3.1

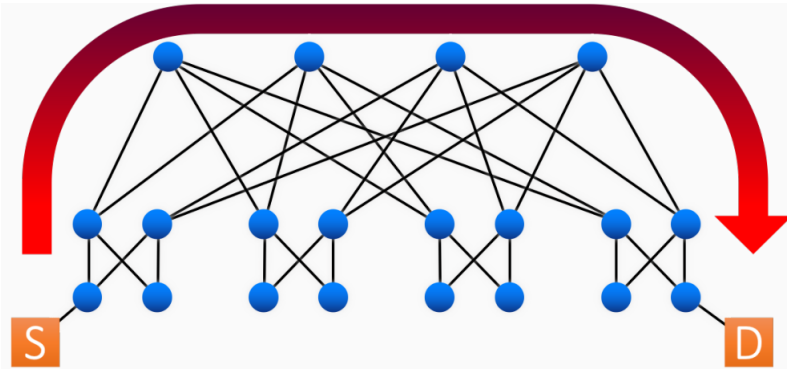
Core switches IP addresses: 87.k.j.i where k=4, j and i denote the coordinates of the switch in the $(k/2)^2$ core switch grid starting from top-left.
Servers have IP addresses of the form 87.pod.switch.serverID

- ServerID is the server position in the subnet of the edge router starting from left to right; e.g. IP addresses of the two servers connected to the switch with IP address 87.2.0.1 are 87.2.0.2 and 87.2.0.3

3.1.14 Traffic Balancing

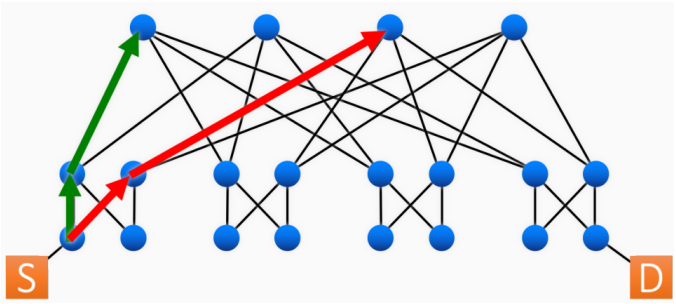
How to balance traffic flows? i

- Let us assume many flows from S to D
- Many equal cost paths going up to the core switches
- Only one path down from each core switch



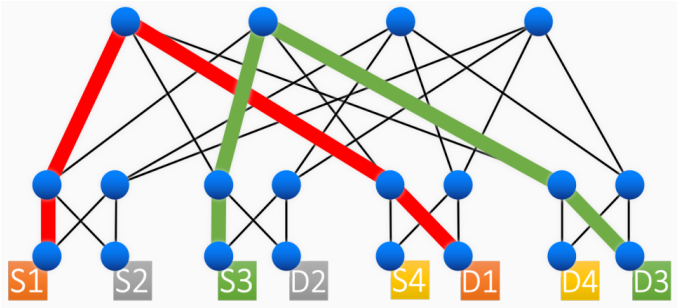
How to balance traffic flows? ii

- Equal-Cost Multi-Path routing (ECMP)
- Randomly allocate paths to flows using hash of the flow
- Agnostic to available resources
- Long lasting collisions between elephant flows



How to balance traffic flows? iv

- This is the perfect way



How to balance traffic flows? v

- This is not a good way, collisions can take place on upward or downward path

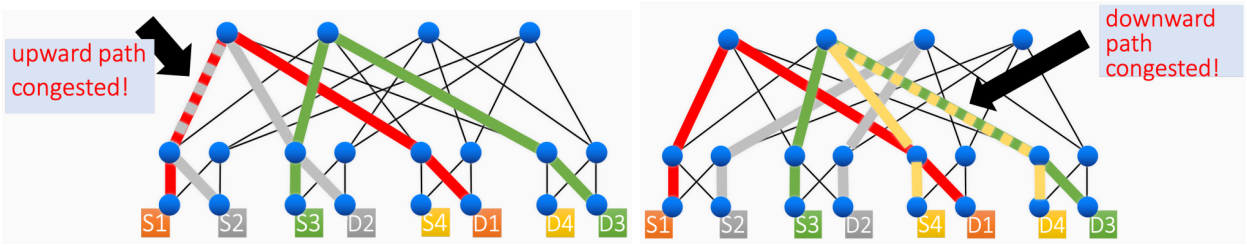


Figure 3.7: Traffic balance collisions

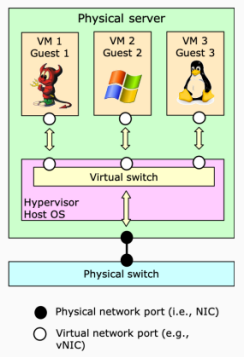
3.1.15 Summary

- Interconnection networks plays an essential role in the design of a data center
  - Layered structure: edge, aggregation, core
  - Many topologies: fat tree quite relevant
- Data center networks
  - Complex requirements from applications, service models, traffic directions
  - Full control helped defining ad-hoc architectures
- Open issues (research): achieve the following requirements together
  - Low latency (short messages, queries)
  - High throughput (continuous data updates, backups)
  - High burst tolerance

3.2 Network in Virtualized Environments

3.2.1 Computing virtualization in brief

Capability to run multiple virtual machines on the same physical host. Main software components are:

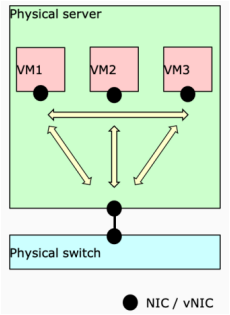


- Virtual machines
- Host OS / Hypervisor
- Software (virtual) switch (or software bridge)

All those components (including the virtual switch) are created by OS people

3.2.2 Virtualization and networking

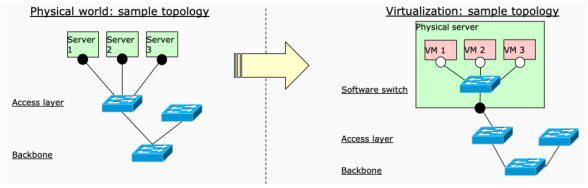
Virtualization introduces additional complexity for networking



- Communication requirement:
  - Deliver traffic from VMs/LXCs to the physical network (and vice versa)
  - Deliver traffic between VMs/LXCs within the same server
- Additional requirements:
  - Assign IP addresses to VM/LXCs
  - Provide advanced features such as load balancing, firewall, etc.
- Note: no difference between VMs or LXCs from networking perspective

Main idea

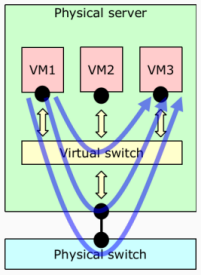
- Physical world: servers are connected to a switch, from here to the backbone
- Virtual world: vservers are connected to a switch, from here to the physical network



3.3 North/South and East/West communications on single server

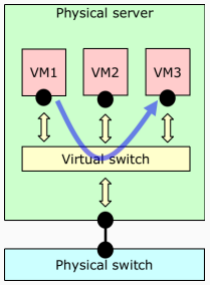
To solve this problem there are many possibilities, for example with a software switch (softswitch), but there are three main options:

1. Virtual switch in the host OS / hypervisor
2. In the NIC (using virtual queues)
3. In the external switch (requires special support)



3.3.1 Host Based Switching

Historically, the first solution used;
In some cases the software switch is in the Host OS, in some other in the Hypervisor (e.g., in case the virtualization layer sits on top of the operating system such as in Virtualbox).



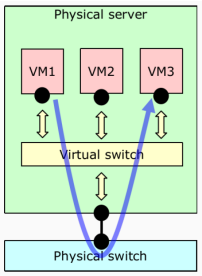
- Strengths
  - High bandwidth (and reduced overhead) for inter-VM traffic
  - Enforces policies early
- Weaknesses
  - Additional processing overhead
  - Additional component (vswitch) to configure and monitor
  - Not clear who controls the switch: IT or networking people?
- Also called "software bridge" or "softswitch"

Other features:

- Centralized management
- Can implement the same features of hardware switches
  - Packet visibility, ACLs, Quality of Service
- Can support a large number of policy rules
- Consumes hypervisor CPU cycles
- Physical NIC to operate in promiscuous mode (packets to VMs may be dropped because the destination MAC address does not match the MAC of the physical NIC)
- Possible hardware offloading can provide big wins
- Sometimes, even controlled by the same commands (e.g., Cisco IOS) and through the same management system that controls the rest of the infrastructure
  - vSwitch can be seen as a native piece of the infrastructure
- Examples:
  - VMware vSwitch, Cisco Nexus 1000V, Open vSwitch

3.3.2 Hairpin switching

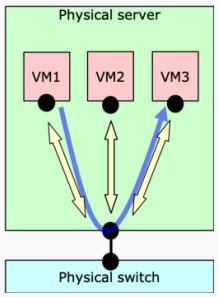
"hairpinning": traffic makes a U-turn : "hairpin"



- Use hardware that is already present
  - However, hardware must be compatible with hairpin switching: standard 802.1D bridges never forward a frame on the same port on which it has been received
- No longer used
- Performance
  - Can leverage the power of the hardware (i.e., forwarding speed)
  - Not consume CPU cycles, consumes PCI bus BW (smaller than CPU-to-MEM one)
  - Traffic crosses twice intermediate components (hypervisor, NIC, physical link)
  - Best when having little VM-to-VM traffic
- Attractive when applying similar policies over all nodes
  - Clear separation between computing and networking domains

3.3.3 NIC switching

Intermediate solution between host and hairpin switching (also about strengths and weaknesses)



- Several NICs available supporting this feature
- Most important advantage: avoids the problem about who controls edge switches
  - NICs are not under the control of the Operating System, hence can be considered part of the network infrastructure
- More and more SmartNICs from datacenter vendors (e.g., Broadcom, Mellanox, Netronome, Pensando, etc.)

SmartNIC is an **intelligent server adapter** (ISA) (e.g., NIC) that:

- Can implement complex server-based networking data plane functions
  - e.g. multiple match-action processing, tunnel termination and origination, metering and shaping and per-flow statistics
- Supports a fungible data plane either through updated firmware loads or customer programming, with little or no predetermined limitations on functions that can be performed
- Works seamlessly with existing open source ecosystems to maximize software feature velocity and leverage

3.3.4 Smart NIC - Why a SmartNIC?

- When amount of CPU spent in a server due to networking tasks becomes large, leaving little or no CPU left over to run applications (e.g., VMs/containers), consider offloading that computation to a discreet component
  - A similar pattern was observed with GPU, which were intended to offload graphical tasks in computers in which that load was significant
  - As a consequence, some servers may not benefit from SmartNICs (e.g., when the processing load due to networking is not significant), hence can stay with the traditional software switch
- The networking may be consuming a significant amount of CPUs
  - Widespread used overlay tunneling protocols (e.g., VxLAN) introduce a noticeable processing cost
  - Detecting and blocking DDoS attacks (which are often handled by datacenter servers; a "centralized" firewall may be difficult to scale) may consume a significant amount of CPU
  - Complex functions (switching, load balancing, etc) may consume a lot of CPU as

Hardware vs Software switching: performance

- Software switches have been demonstrated at Nx10Gbps
  - Although with significant CPU cost, stolen from VM processing
    - But. . . are we creating this infrastructure to achieve fast switching processing or to support many VMs?
- Software switches can drop traffic closer to the source
  - Important in clouds with over-subscribed links and untrusted sources
- Hardware offloading for software switches can provide big wins
  - Checksum offloading
  - SR-IOV may be even better
- Software switches are better for local VM-to-VM traffic

Summary

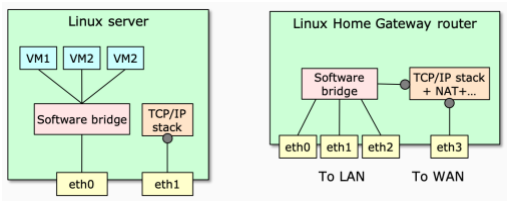
- Hardware switches attractive when applying similar policies over all nodes or in aggregate with little local VM-to-VM traffic
  - Clear separation between computing and networking domains
- Host switches provide more flexibility and fine-grained control at cost of hypervisor CPU cycles
  - Not clear separation between computing and networking domains
- Host switches is currently the most common solution
  - Easy to add new features (e.g., added value services such as mobility), without requiring explicit support from the network
  - Commercial interests are pushing for this solution, adding the most intelligence we can in them, leaving the network as a dumb pipe (just provide basic connectivity)

3.4 Software Bridges in Linux (Virtual Switch)

3.4.1 Introduction

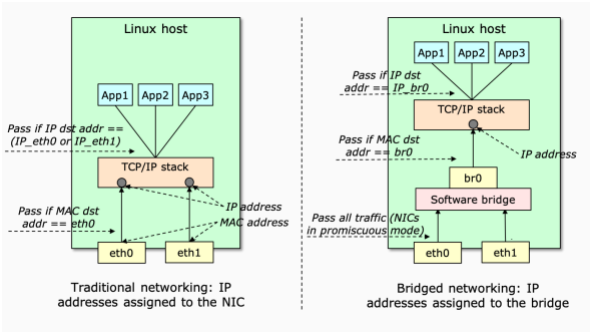
- Why do we need a software bridge?

  1. Provide intra-host connectivity to execution containers (e.g., different network namespaces, VMs, Docker, Linux Containers, etc)
  2. Handle IP addresses differently from the "official" IP model
     - "One IP address per NIC" rule
     - Some examples:



3.4.2 Basic bridging networking in Linux

Linux allows to create a bridge across the interfaces, and one unique (virtual) interface that is valid for the entire host



In the past: The setup of a bridging process in Linux was a way to have an 802.1D bridge, running in software, that was part of a bigger network

- We can modify the bridge code
- We can play with a bridged network without having to buy a physical device

Now: it represents a way to overcome one of the limitations of the IP protocol, which requires a distinct IP address on each interface

- Since the default gateway is unique for the entire host, it is hard to use multiple paths
  - It requires an ICMP redirect or the manual definition of a specific route
- If allowed by the STP, all the NICs can be used to rx/tx traffic

Software bridges in Linux: Linux has three main types of software bridges

1. Linuxbridge
2. macvlan
3. Open vSwitch

3.4.3 Software bridges in Linux - LinuxBridge

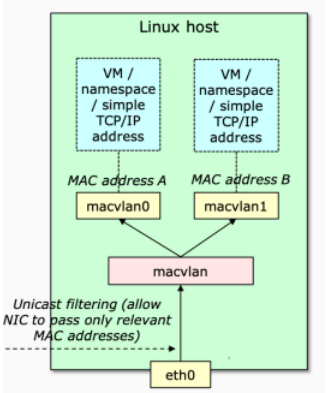
It is the most common software bridge

- Behaves like a traditional hardware switch (i.e., IEEE 802.1D)
  - Has filtering DB (FDB), spanning tree (STP), etc.
- NICs in promiscuous mode allow to receive all the packets
  - Filtering based on the MAC address of the br0 virtual device
- Traffic can be sent to the network using both network interfaces
- The IP address configured on the bridge is reachable from both interfaces
  - Looks like a "loopback" address

3.4.4 Software Bridges in Linux - Macvlan

Implements a VLAN-like behavior by using MAC addresses instead of 802.1Q tags

- Looks like a (L2) subinterface derived from the main NIC
- Each macvlan interface
  - Has its own MAC address
  - Can have a distinct IP address
- Applications can bind to a specific interface / IP address
  - LXC guests can use one side of that interface, by moving the macvlan interface in their namespace

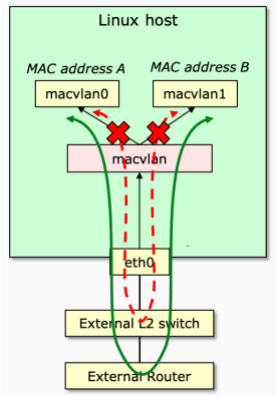


- Supports four operating modes
  - Private
  - Virtual Ethernet Port Aggregator (VEPA) - Default mode
  - Bridge
  - Pass-through
- Can configure the NIC to filter unicast packets based on MAC address instead of running in promiscuous mode (except for pass-through)
  - Requires explicit NIC support

Macvlan - Private Mode

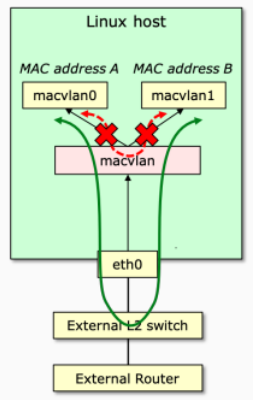
In the private mode:

- VLAN-like behavior
- Forbids any inter-macvlan traffic
  - I.e., MAC A cannot send data to MAC B
- Traffic will be delivered only if it comes with a different source MAC address
  - E.g., from MAC A to a router and then to MAC B



Macvlan - VEPA Mode

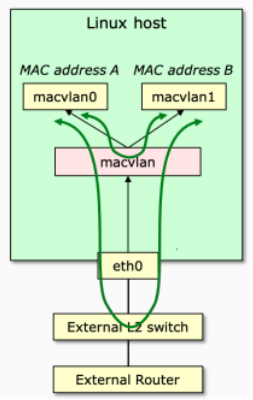
In the VEPA mode:



- In this case MAC A is allowed to talk with MAC B only if the traffic comes from the external world
  - i.e., the Macvlan driver cannot send traffic from MAC A to MAC B
  - However, if the same frame is received from an external switch (it may also be the L2 switch on the NIC), the packet is delivered correctly
- The problem is that the external switch cannot be a traditional 802.1D bridge
  - 802.1D does not allow a frame to be sent back to the same interface on which it was received

Macvlan - Bridge Mode

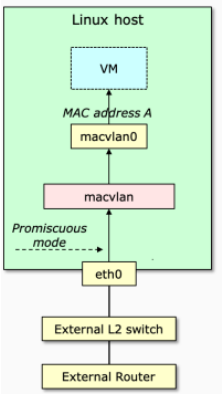
In the Bridge mode:



- Allows forwarding of all the traffic
- The Macvlan driver acts as a (simple) bridge
  - No filtering database (i.e., MAC learning)
  - No STP
  - Only one uplink

Macvlan - Pass-Through Mode

In the Pass-Through mode



- Supports only one macvlan device (single VM)
- Allow all traffic to be sent to the upstream component (e.g., VM)
  - Used mainly for VMs (as macvtap)
- NIC has to work in promiscuous mode
- Allow the upstream component to use any MAC address
  - No MAC filtering in the macvlan driver, nor in the NIC
  - Traffic is always sent upstream, and that component (e.g., the VM vNIC driver) will be in charge of the filtering
- MAC is no longer used

3.4.5 Bridge vs Macvlan

- Linux bridge
  - Behaves like a traditional hardware switch
- Macvlan: trivial bridge → simple and fast
  - No need to do learning (or STP) as it knows every mac address it can receive
  - Uses less host CPU and provides slightly better throughput.
- Use Bridge if:
  - Need to connect VMs or containers on the same host
  - For complex topologies with multiple bridges and hybrid environments
    - e.g. hosts in the same Layer2 domain both on the same host and outside the host
  - Advanced flood control, FDB manipulation, etc.
- Use Macvlan if:
  - Just need to provide egress connection to the physical network to your VMs or LXCs

3.4.6 Software Bridges in Linux - Open vSwitch (OvS)

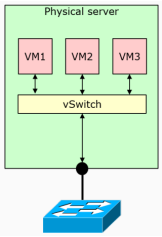
- Open vSwitch (OVS) is a software implementation of a virtual multilayer network switch, designed to enable effective network automation through programmatic extensions
- Supports standard management interfaces and protocols such as NetFlow, sFlow, and more
- Designed to support transparent distribution across multiple physical servers by enabling creation of cross-server switches in a way that abstracts out the underlying server architecture
- Can operate both as
  - software-based network switch running within a virtual machine (VM) hypervisor
  - control stack for dedicated switching hardware

OvS - Implementation

- Ported to multiple virtualization platforms, switching chipsets, and networking hardware accelerators
- Integrated into cloud computing software platforms and virtualization management systems (e.g. OpenStack, OpenNebula)
- Implemented in Kernel

3.5 Single Server: Complex Services

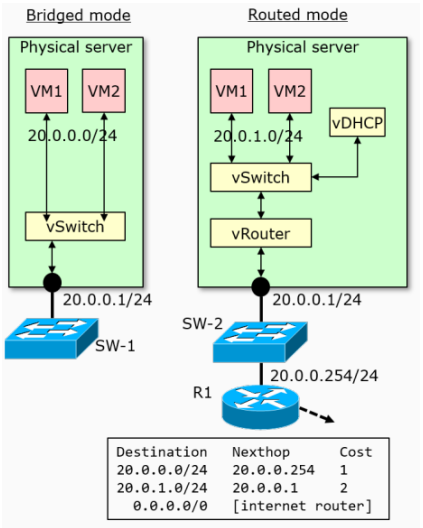
Problems:



1. Need to determine who provides IP address to VMs, and which addresses can be assigned
2. Providing connectivity may require more network services than just setting up a bridge (softswitch)
3. Multi-tenancy (i.e., strong network isolation) is a "must" in a virtual environment
4. Tenants may require additional network services within their own virtual setup
5. The server has its own TCP/IP stack (and applications, e.g., for SSH access, management, etc) that has to coexist with virtual network services.

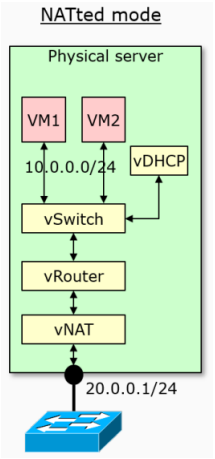
1 - IP address assignment

Who provides IP addresses for VM/dockers?
Option (1) 2 modes (bridged and routed, the first one not used anymore): Use addresses known by the physical network (**direct routing**), the VDHCP provides the IP to the VMs



- Use the same addresses of the physical network (e.g., VMs attached in "bridged" mode, i.e., with pure L2 connectivity)
- **Pros:** seamless inbound/outbound connections towards VMs
- **Cons:** require the coordination of the (physical) network manager
  - This may limit the agility of the virtualization

Option (2) **NATted mode:**

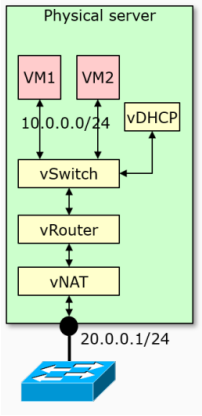


- Use private addresses and connect to the Internet through NAT (to reach a VM i'm using 2 different IPs inside and outside the machine
  - **Pros:** Work with any IP address range
    Requiring no coordination with the (physical) network manager
  - **Cons:** Seamless outbound connectivity, but a NAT static rule is required to support inbound traffic
  - **Cons:** VMs reached with different IP addresses, depending if the sender is inside or outside the private network

2 - Providing feature-rich network connectivity

Additional network services may be required to connect VM/Docker to the physical infrastructure.

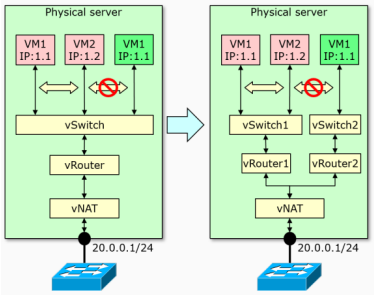
- If private addresses are used for VMs, hypervisor must also provide Router, NAT and DHCP [usually provided "transparently" by the infrastructure (e.g., hypervisor)]
- Implementation:
  - Leverage Linux kernel whenever possible (e.g., for bridging, routing, firewall/NAT (iptables))
  - Additional LXC/Docker can be used for other services



3 - Multi-tenancy

Multi-tenancy (i.e., strong network isolation) is a "must" in a virtual environment.

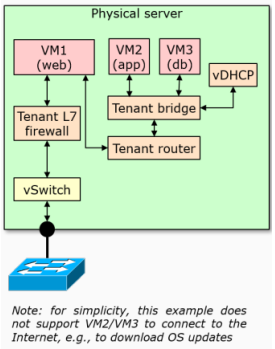
- To protect communications (A cannot talk to B)
- To prevent IP address conflicts (e.g., same IP address range for both tenants)
- Usually achieved by setting up multiple vSwitches and vRouters
- VLANs are also possible (not easy to manage and setup)



4 - Tenant-defined network services

Tenant may require a complex topology for its services.

- Requires:
  - E.g., tenant deploys a service with 3 VMs (web server, application server, database), but only the web frontend (VM1) must be directly reachable from Internet, for increased security

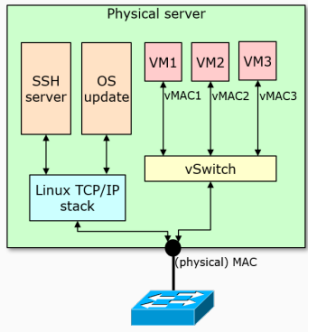


- Implementation
  - Cloud controllers (e.g., OpenStack) provide some pre-defined services (e.g., router, bridge, NAT), often using Linux kernel functionalities or LXC/Docker
    - Tenant can simply use them as "black box", whatever implementation is used
  - Additional services can be set up by the tenant itself (e.g., VM with the proper software, such as for the L7 firewall)(For the hypervisor, those are just VMs)

5 - Co-existence of virtual services and host applications

Server has its own TCP/IP stack (and apps, e.g., for SSH access) that must coexist with VMs

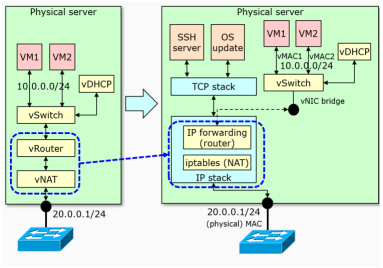
- Some observations:
  - In case of "bridged" VMs, NIC MAC filtering (active by default) must be **disabled** to allow the vSwitch to receive inbound Ethernet traffic for VMs (with their own vMAC)
  - There should be a way to distinguish inbound traffic toward VMs (e.g., through MAC addresses or IP addresses) from the one directed to Linux port
    - In case of VMs using private IP addresses it may be more difficult as we should use a L4 session table



3.5.1 Leveraging the kernel network stack for ancillary services

Nowadays, the Linux kernel is often used also to provide virtual network services

- This justifies the large amount of work recently ongoing in the Linux kernel networking community

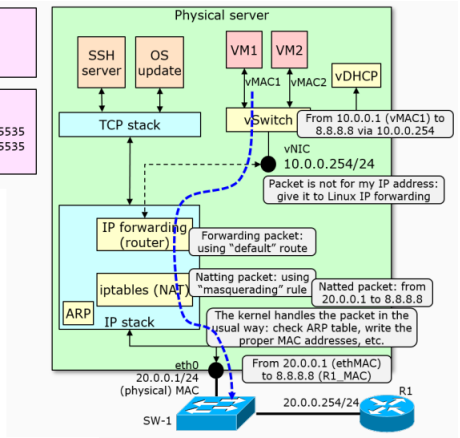


Example:

- Packet sent from VM1 to Internet (Technically, to the default gateway)
- The vNIC receives the packet, it detects that it is not the final recipient, so the packet is transferred to the Linux IP forwarding module
  - vNIC is device driver (runs in kernel)
- Kernel forwards it to the next hop based on the host routing table and ARP table (usual)

```
$ sudo ip route
default via 20.0.0.254 dev eth0 proto dhcp
20.0.0.0/24 dev eth0 proto kernel scope link src 20.0.0.1
10.0.0.0/24 dev vNIC0 proto kernel scope link src 10.0.0.254
```

```
$ sudo iptables -t nat -L
target      prot  source         destination
MASQUERADE  tcp   10.0.0.0/24   !10.0.0.0/24   masq ports: 1024-65535
MASQUERADE  udp   10.0.0.0/24   !10.0.0.0/24   masq ports: 1024-65535
MASQUERADE  all   10.0.0.0/24   !10.0.0.0/24
```



3.6 DC - Wide Services

3.6.1 Introduction

- When moving from a single server to a datacenter, in principle we must solve the same problems presented before
  1. IP addresses assignment
  2. Providing feature-rich network connectivity
  3. Support for multi-tenancy
  4. Support for tenant-defined network services
  5. Support (and integration) with the host TCP/IP stack
- The challenge is that everything must be provided "at scale", datacenters can include thousand of servers

3.6.2 Tenant vs cloud manager view

- Tenant view
  - Tenants want to deploy their "virtual services" and must ignore the physical infrastructure (how many servers we have, how are they connected)
- Cloud manager view
  - Physical (physical network, servers and hypervisors) infrastructure must be able to provide the requested "logical" view
  - Additional problems (e.g., VM migration used to optimize computing tasks, energy consumption, etc.) may further complicate the picture

3.6.3 Providing L2 connectivity to tenant services across DC

- Let's pretend the customer wants to have a vanilla L2 network, spanning among all its services across the entire datacenter
  - Frequent in OpenStack, where we want to provide highly customizable tenant networks
- In this case, the tenant wants to have control over IP addresses (it gets an L2 infrastructure, so L3 is under its own responsibility)
  - Consequently, IP addresses for VMs/containers are necessarily private
  - Decoupled addressing in the real (network) and virtual (VMs/containers) space
- Solution: tenant's traffic is **tunneled** between servers

Example

Using TUNNELS to extend L2 tenant network across DC

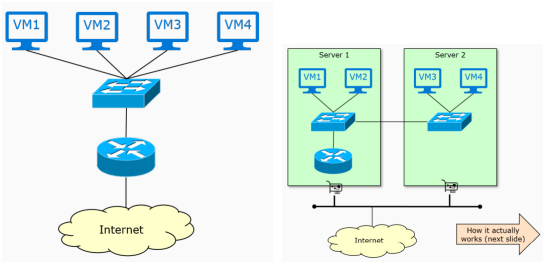
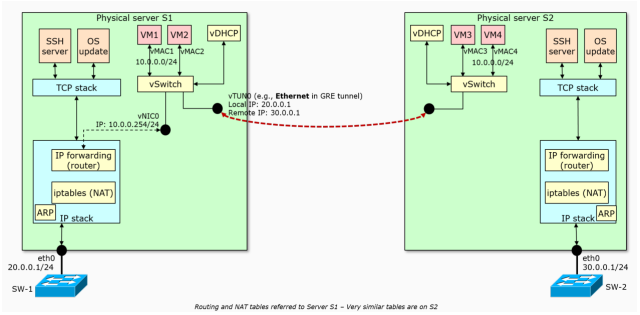
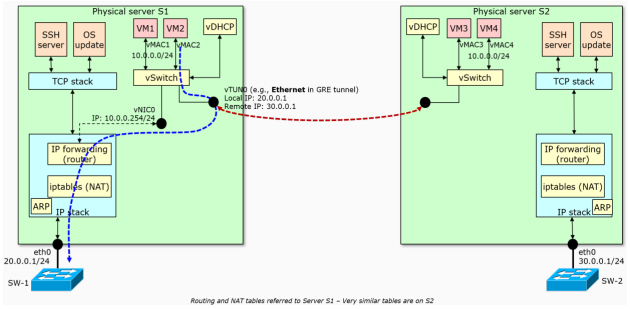


Figure 3.8: Requested service: tenant view & Requested service: cloud manager view

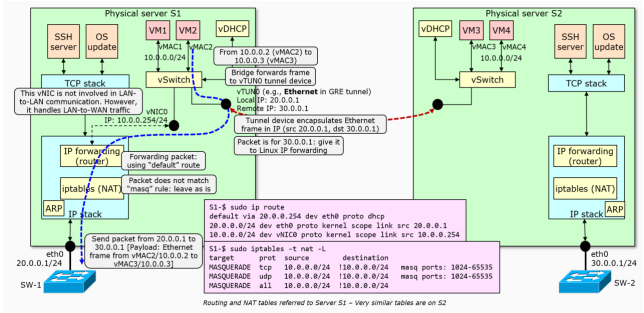
Example 1 Using TUNNELS to extend L2 tenant network across DC



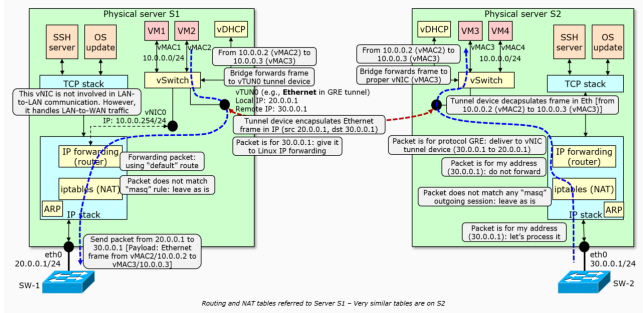
Example 2 Using TUNNELS to extend L2 tenant network across DC



Example 3 Using TUNNELS to extend L2 tenant network across DC

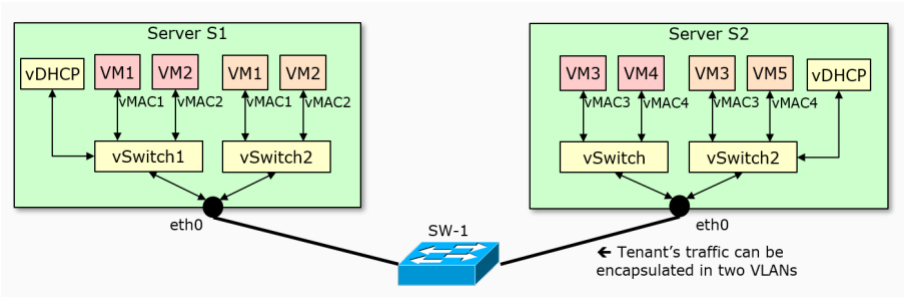


Example 4 Using TUNNELS to extend L2 tenant network across DC



3.6.4 Would VLAN be appropriate to extend tenant's L2 networks?

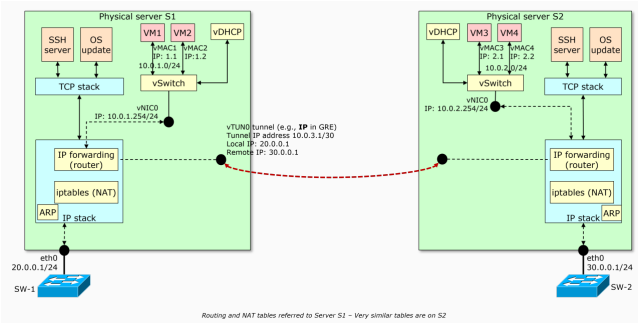
- There are several reasons why this would not work properly
  - It require the cooperation of the (physical) network IT manager
  - VLANs are limited (4096); QinQ or similar should be used, complicating this solution
  - VLAN cannot be propagated with L3 physical networks (happens in large DCs)
    - Even worse, the physical network could a WAN link (e.g., geo-distributed data-centers)
  - What about the traffic *toward Internet*? (need a "gateway node", as in OpenStack)
  - What about the traffic *to configure the server?* (need another NIC, as in OpenStack)



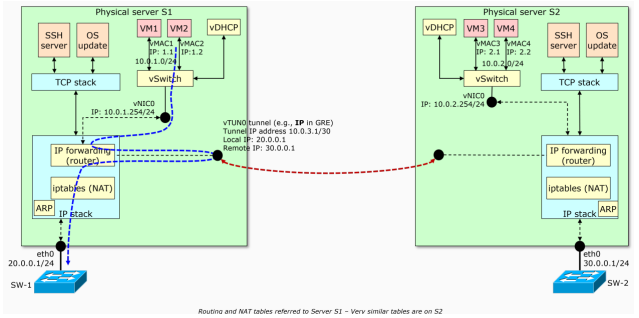
3.6.5 Providing L3 connectivity to tenant services across DC

- Let's pretend the customer just asks for L3 connectivity and does not care about having an L2 network nor which IP addresses are used
  - This represents the "Kubernetes" case, with loosely customizable tenant networks
- In case the orchestrator can deliver sophisticated network services to provide connectivity, which are not under the control of the tenant
  - The tenant cannot create its own network services either
- In this case, two working modes are possible: **tunneling** and **direct routing**

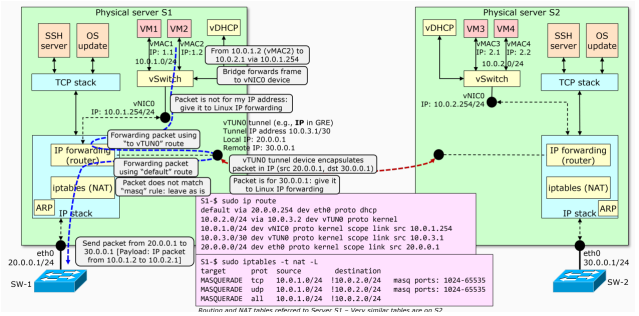
Example 1 Using TUNNELS to extend L3 tenant network across DC



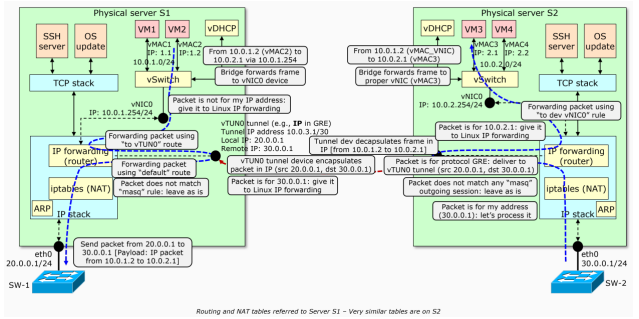
Example 2 Using TUNNELS to extend L3 tenant network across DC



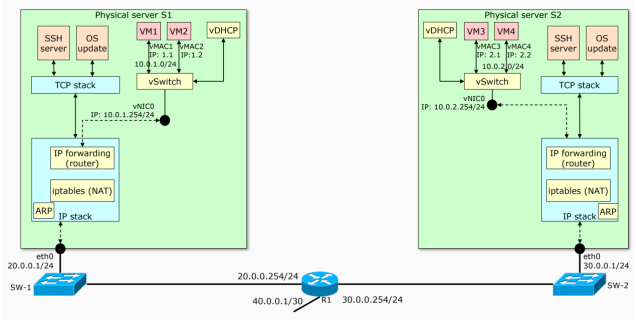
Example 3 Using TUNNELS to extend L3 tenant network across DC



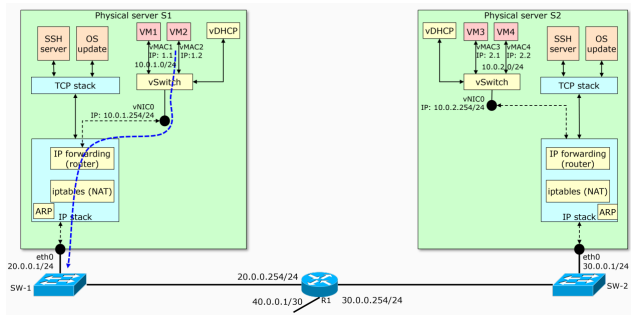
Example 4 Using TUNNELS to extend L3 tenant network across DC



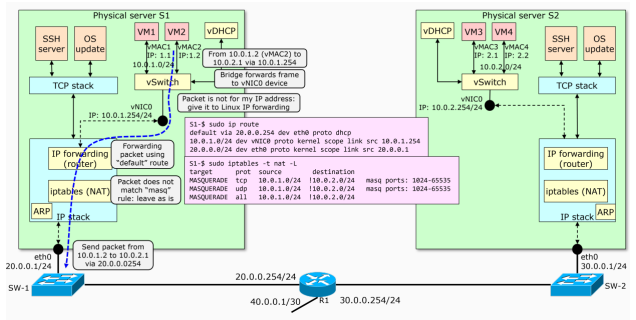
Example 1 Using DIRECT routing to extend L3 tenant network across DC



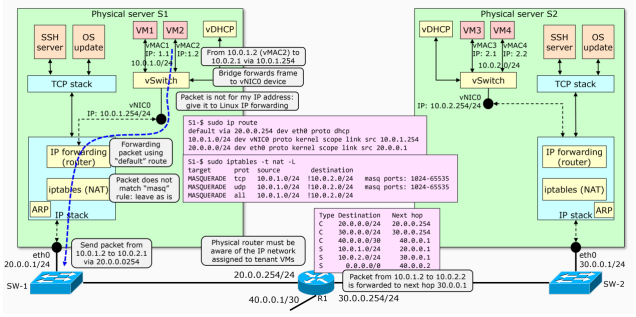
Example 2 Using DIRECT routing to extend L3 tenant network across DC



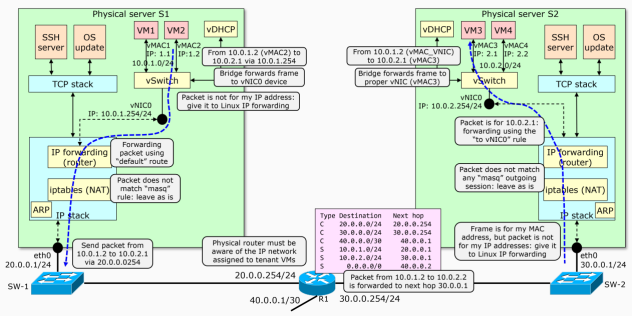
Example 3 Using DIRECT routing to extend L3 tenant network across DC



Example 4 Using DIRECT routing to extend L3 tenant network across DC



Example 5 Using DIRECT routing to extend L3 tenant network across DC



Link to explanation of the examples Google Doc:
`https://docs.google.com/document/d/1uvVr5-xNPwHPmMNgXzzjoUgLxQgocf5kJgZfrhodlq4/edit?usp=sharing`

Some comments

- The frame/packet flow is "easy" to understand if you remember who you are, and the information available in that component (i.e., according to the specific context of the packet). For example:
  - When the packet is in the VM, you know only the IP routing table / ARP table of the VM
  - When the frame is in the bridge, you know only the filtering database
  - When the packet is the Linux forwarding module (networking stack), you know only nthe IP routing table / ARP table of the host
- The traffic may be handled in different way according to the destination
  - E.g., if the destination is another (local) VM, the packet may have to be tunneled
  - E.g., if the destination is the Internet, the packet may have to be NATted
- An orchestrator may be in charge of updating all the parameters required to scale with the number of servers (e.g., adding the required routes in the servers, creating the proper tunnels, etc.)

Tunnels and full mesh

Point-to-point tunnels do not scale (a full mesh is required among servers): This what is needed with tunneling technologies such as GRE (or Secure GRE, nobody uses this technology), IPSec.

- This is the reason often tunnels are created using the VxLAN technology, which provides two additional advantages
  - A transparent full-mesh of tunnel among the machines that you created, with a single endpoint for all the hosts belonging to the same VxLAN domain
  - VxLAN frames are encapsulated in UDP, hence the L4 port enables distributing the traffic across multiple parallel links (Each leaf switch has multiple connections towards the spine)

Overlay vs Direct Routing

Overlay

- Logical Network on top of a network
- Also called tunnel(methodology), it enables the deployment of tenant services without any interaction with the infrastructure provider
  - Useful in case of public datacenters
  - May trigger some performance problems due to the reduced MTU on the tunnel
- Preferred with OpenStack
- Also known as "Overlay model"

Direct Routing

- Requires the collaboration of the infrastructure provider
  - Most cloud providers expose a software interface where each tenant (actually, the orchestrator such as Kubernetes) configures the required routes
  - No MTU problems
- Preferred with Kubernetes

Additional considerations for overlay model

- Some orchestrators prefer to create the virtual network topology without touching the physical network
  - Network is just a "traditional" network, either L2 or L3
  - Network assigns IP addresses to the servers (in fact, to hypervisors) to transport traffic coming from virtual networks (which is, in fact, tunneled)
- **Complete decoupling of physical and virtual networks**
  - Virtual network traffic is independent from the physical topology
  - IP addressing in each virtual network is independent from the physical infrastructure
  - Traffic is moved from a software switch to another by tunneling
  - Cloud orchestrator do not need to touch (e.g., configure) the physical network; hence can be controlled by different parties (e.g., IT dept. the former, network admin the latter)
- Complete decoupling of VM orchestration (e.g., Vmware) from physical network (e.g., Cisco)

3.6.6 Summary

- Understanding how (virtual) networking works may be more difficult than originally thought
- **Challenges**
  - Need to consider multiple models: e.g., Overlay vs Direct routing
  - Need to mix virtual and real networking
  - Need to be aware of how Linux networking works (Not always obvious for a network engineer used to practice with hardware boxes)
- The good: Usually networking works seamlessly
- The bad: When it does not work properly, debugging may be a nightmare

4 Cloud Storage

4.1 Introduction

Storage and data models Storage model describes the layout of a data structure in a physical storage, while a data model captures the main logical aspects of a data structure in a database

Properties of storage models Read/write coherence and before-or-after atomicity are two highly desirable properties of any storage model and in particular of cell storage

4.1.1 Types of storage

Block storage

Block storage manages data as blocks within sectors and tracks, commonly used for databases. DBMSs(Data Base Management Systems), that is a software that controls the access to the database, use block storage to read and write structured data. In cloud, data stores are usually distributed creating distributed databases where users store information on a number of nodes

File storage

File storage manages data in structured files. It exposes a file system that controls how data is stored and retrieved. File systems are used on local data storage devices or provide file access via a network protocol (Examples of distributed file systems: NFS, Google File System).

Object storage

Object storage manages data as objects. Each object typically includes the data itself and a globally unique identifier. Example of distributed object-storage for cloud: OpenStack Swift.

4.2 Block storage

4.2.1 OpenStack Cinder

Implements services and libraries to provide on-demand, self-service access to block storage resources, manages block storage. It provides API to interact with vendors' storage backends, exposes vendor's storage hardware to the cloud, provides persistent storage and enables end users to manage their storage without knowing where that storage is coming from.

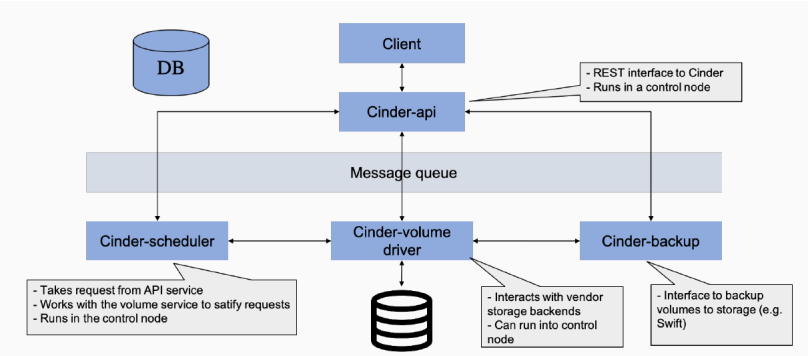


Figure 4.1: Cinder Architecture

Cinder supports: Volume create/delete, volume attach/detach, snapshot create/delete, create volume from snapshot, copy image to volume, copy volume to image, clone and extend volume.

4.3 Distributed file systems

Logical and physical organization of a file File is a linear array of cells stored on a persistent storage device. File pointer is a cell used as a starting point for a read or write operation. A file can be organized:

- Logical: reflects the data model, the view of the data from the perspective of the application
- Physical: reflects the storage model and describes the manner the file is stored on a given storage media

File system File system is collection of directories, each directory provides information about a set of files (file system controls how data is stored and retrieved). It can be:

- Traditional – Unix File System
- Distributed file systems – Network File System

4.3.1 Traditional: UFS

Unix File System (UFS) is layered design provides flexibility(separate the concerns for the physical file structure from the logical one). Hierarchical design supports scalability (supports multiple levels of directories and collections of directories and files). Metadata supports a systematic design philosophy, it includes: file owner, access rights, creation time, time of the last modification, file size, the structure of the file and the persistent storage device cells where data is stored. At the end, inodes contain information about individual files and directories.

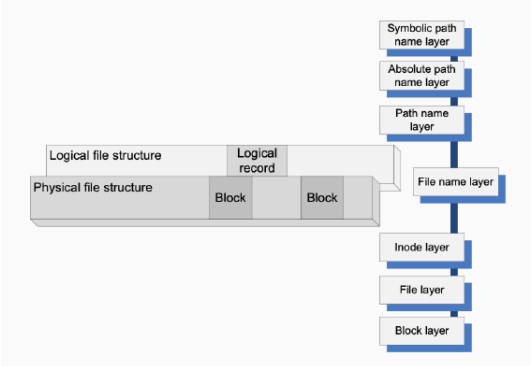
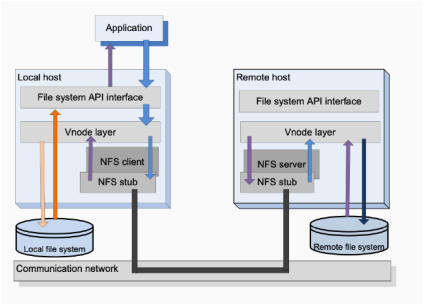


Figure 4.2: UFS layering

- Lower layers: physical organization
- Upper layers: logical organization
- File name layer mediates between machine- and user-oriented views of the FS

4.3.2 Network file systems: NFS

The Network File System (NFS) design objectives are: Provide the same semantics as a local Unix File System (UFS), facilitate easy integration into existing UFS, support clients running on different operating systems and accept a modest performance degradation due to remote access.
NFS is based on the client-server paradigm, so client runs on the local host while the server is at the site of the remote file system.



A remote file is uniquely identified by a file handle (fh), it is a 32-byte internal name.

Vnode layer (correnspondingof inode) implements file operation. The steps of interaction:

1. NSF client packages the relevant information about the target
2. NFS server passes it to the vnode layer on the remote host
3. Remote vnode layer directs it to the remote file system

Design choices for distributed file systems

- Common policy: once file is closed, server will have the newest version on persistent storage
- Policy to write a block: delay in write-backs: a block is first written to cache and writing on the disk is delayed for a time in the order of tens of seconds
- Concurrency: Sequential write-sharing(a file cannot be opened simultaneously for reading and writing by several clients) and concurrent write-sharing(multiple clients can modify the file at the same time).

4.3.3 General Parallel File System (GPFS)

Parallel I/O implies concurrent execution of multiple input/output operations. Concurrency control is a critical issue for parallel file systems. It is developed at IBM and designed for optimal performance of large clusters. A file consists of blocks of equal size, ranging from 16 KB to 1 MB, stripped across several disks.

GPFS configuration Disks are interconnected via SAN, while compute servers are distributed in LANs.

GPFS reliability To recover from system failures, GPFS records all metadata updates in a write-ahead log file. The log files are maintained by each I/O node. Data striping (segmenting logically sequential data, e.g. file, so that consecutive segments are stored on different physical storage devices) allows concurrent access and improves performance but can have unpleasant side-effects.

4.3.4 Google File System (GFS)

GFS uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs. We have to analyse some design considerations:

- Scalability and reliability are critical features of the system
- Most common operation is to append to an existing file
- Sequential read operations are the norm
- Users process the data in bulk and are less concerned with the response time
- Consistency model should be relaxed to simplify the system implementation

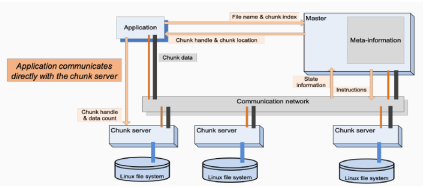
An example is CloudStore, an open source C++ implementation of GFS.

GFS design decision Segment a file in large chunks, implement an atomic file to append to the same file, build cluster around high-bandwidth, eliminate caching at the client site, ensure consistency by channeling critical file operations through a master, a component of the cluster which controls the entire system and support efficient checkpointing, fast recovery mechanisms and a garbage collection mechanism.

GFS chunks GFS files are collections of fixed-size segments called chunks, size is 64 MB, it consists of 64 KB blocks and each block has a 32 bit checksum. Chunks are stored on Linux files systems each chunk is assigned a unique chunk handle.

GFS cluster architecture

- Master maintains state information about all system components
- A chunk server runs under Linux
- Data and the control paths are shown separately
- Arrows show control flows between app, master and chunk servers



Steps of a write request

1. Client contacts the master which assigns a lease to one of the chunk servers
2. Client sends data to all chunk servers holding replicas
3. Client send write request to primary chunk server once it got acks from all chunk servers holding replicas
4. Primary chunk server sends write to all secondaries
5. Each secondary applies mutations in the order of the sequence number and sends ack back to primary
6. After receiving ack from all secondaries, primary acks client

4.4 Lock and consensus

Operating systems use lock managers to organise and serialise access to resources.
To elect a leader or a reliable master to take decisions, consensus must be sometimes reached.

Types of Locks

- Effect
  2 kind of locks:
  - **Advisor locks:** Is based on the assumption that all processes play by the rules, they don't have effect on processes and access the shared object directly
  - **Mandatory locks:** Have block access to the locked object to all processes that don't hold the locks
- Time
  2 kind of locks:
  - **Fine-grained locks** have locks that can be held for only a very *short* time
  - **Coarse-grained locks** that held for a *longer* time

Systematic approach to locking Two approaches possible:

- Delegate to the clients the implementation of the consensus algorithm and provide a library of functions needed for this task (Depend on NO other servers)
- Create a locking service which implements a version of the asynchronous Paxos algorithm and provide a library to be linked with an application client to support service calls (easier to maintain existing structure and reduces number of server needed)

4.4.1 The Chubby lock service

Chubby is developed by Google for use within a loosely-coupled distributed system. It has large number of machines connected by high-speed network, provides coarse-grained locking and reliable (low-volume) storage.

Then Chubby provides an interface much like a distributed file system, whole file read and writes operation (no seek), has an advisory locks and a notification of various events (file modification) It uses asynchronous consensus: PAXOS with lease timers to ensure liveness.

Design choices Two key design decision:

- Google chooses lock service
- Serve small files to permit the election of client application masters

System Structure There are two main components that communicate via RPC:
A **replica server** and a **library** linked against client applications.
Chubby cell consists of a small set of servers (typically 5) known as replicas, which use PAXOS to elect a **master**(R/W roule) and replicate logs. If a master fails, other replicas elect a new one.

1. Clients find the master by sending master location reqs to the replicas listed in the DNS
2. Clients use RPCs to communicate with the master via a chubby library
3. Client redirects all requests to it either until it ceases to respond or until it indicates it is no longer the master

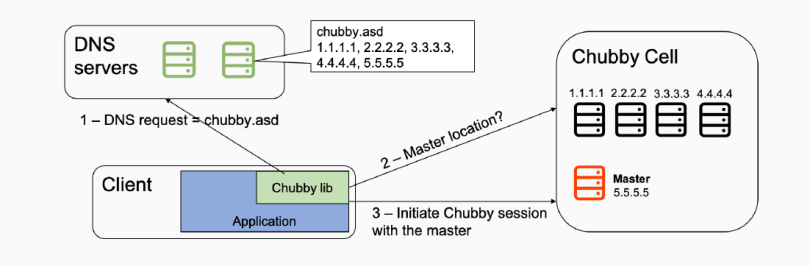


Figure 4.3: Chubby design

Chubby cell consisting of 5 replicas:

- One of them elected as a master
- All maintain copies of a simple database

Clients use RPCs to communicate with the master via a chubby library (Send read/write only to the master).

Master propagates write to replica and replies after the write reaches a majority, then replies directly to reads, as it has most up to date state.

If applications wants to get the lock over a shared resource (e.g. and operate that resource):

- Resources can be registered in the chubby file system as nodes
- Potential clients try to create a lock on chubby
- The first one that gets the lock becomes the one that can use that resource

Files and directories Chubby exports a file system interface simpler than Unix:

- Tree of files and directories with name components separated by /
- Each directory contains a list of child files and directories
- Each file contains a sequence of un-interpreted bytes
- No symbolic or hard links, no file move
- No directory modified times, no last-access times
- No path-dependent permission semantics:

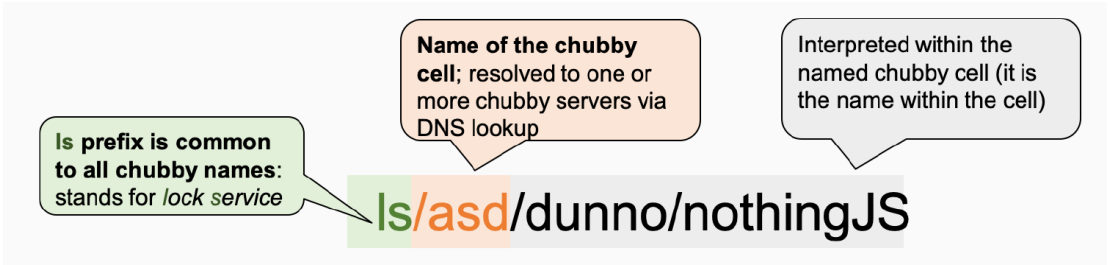


Figure 4.4: Chubby files and directories

A **Chubby Node** is a file or directory, any node can act as advisory reader/writer lock and it can be either permanent of temporary. **Metadata** is 64-bit file content checksum, instead **Handlers** include check digits, sequence number and more info.

Locks, sequencers and delay Each Chubby file and directory can act as reader/writer lock (advisory ones). Acquiring a lock in either mode requires write permission. It can be: Exclusive mode (writer, for one client) and Shared mode (reader, for any number of client) In case of contention, there are two solutions:

- Sequencer: string holding information about state of the lock
- Lock-delay: time to wait before claiming the lock if the lock becomes free

API List of possible API that can be used:

- Open/close node name
- Read/write full contents
- Set ACL
- Delete node
- Acquire or release lock
- Set, get or check sequencer

Chubby can be used for:

- Elect a primary from redundant replicas(GFS & Bigtable)
- standard repository for files that require high-availability(ACLs)
- well-known and available location to store a small amount of meta-data
- name service

The bigtable can be used to:

- Elect the master
- Allow the master to discover the servers its controls
- To permit clients to find the master

4.5 Distributed databases

4.5.1 Google Big Table

It's a **distributed storage system** for managing *structured data*. It is a flexible and it has an high performance solution. It is designed to scale to a very large size(petabytes of data) and it isn't a relational database, it is a sparse, distributed, persistent multi-dimensional sorted map (key/-value store).

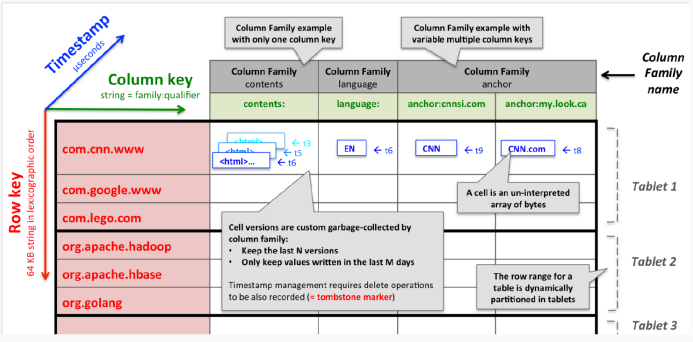
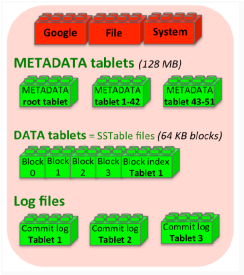


Figure 4.5: Data Model

4.5.2 Building blocks

1 - Google File System

Bigtable uses the fault-tolerant and scalable distributed Google File System file system to store:



- Metadata: METADATA tablets
- Data: SSTables collection by tablet
- Log: tablet logs

2 - Google SSTable (Sorted String Table) It is used to store table data in GFS and the file format is persistent, ordered immutable map from keys to values. It contains a sequence of 64 KB blocks and block index stored at the end of the file.

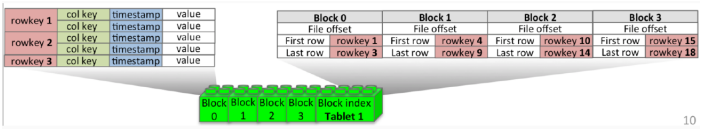
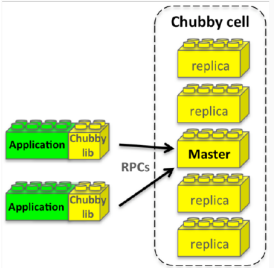


Figure 4.6: Building Box

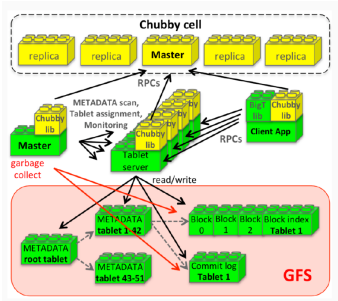
3 - Google Chubby Chubby server with 5 active replicas and one master to serve requests, it provides a reliable namespace that contains directories and small files(<256 KB).
Bigtable uses Chubby for a variety of tasks:



- Ensure there is at most one active master at any time
- Store the bootstrap location of Bigtable data
- Discover tablet servers and finalize tablet server deaths
- Store Bigtable schema information (column family information for each table)
- Store ACLs
- If Chubby is unavailable, then Bigtable is unavailable

GBT - Components

The major components of GBT are:



1. One **master server** that assign tablets to tablet servers, detect the addition and expiration of tablet servers, balance tablet server load, do a garbage collecting of files in GFS and handling schema changes.
2. Many **Tablet servers**, each manages a set of tablets, handles R/W request and split large size tablets
3. **Clients** Communicate directly with tablet servers for R/W

Tablet location

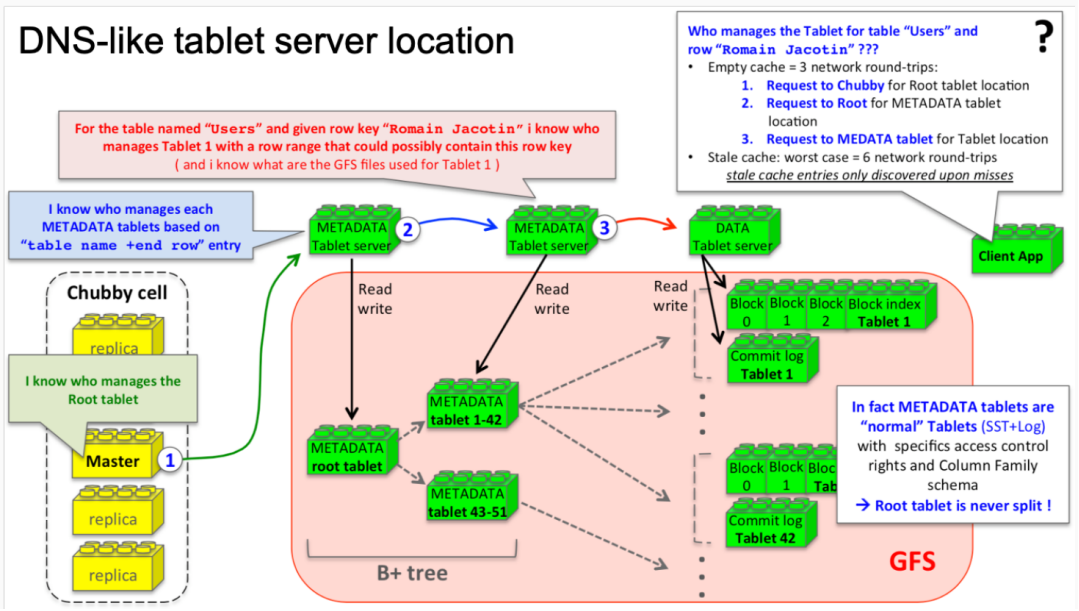


Figure 4.7: Tablet location

GBT - Implementation API

Tablet assignment and server discovery Each tablet is assigned to one tablet server at time. Master keeps tracks of set of live tablet servers, current assignment/unassigned of tablet to tablet servers and when a tablet is unassigned, master assigns the tablet to an available tablet server by sending a tablet load request. When a tablet server starts, it creates and acquires an exclusive lock on a uniquely-named file in a specific Chubby directory ("servers" directory). A tablet server stops serving its tablets if it loses its exclusive Chubby lock.

Tablet server monitoring Master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets. Master periodically asks each tablet server for the status of its lock to detect when a tablet server is no longer serving its tablets. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last attempts, the master attempts to acquire the lock for the Chubby file.

Master isolated and startup To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can changes them:

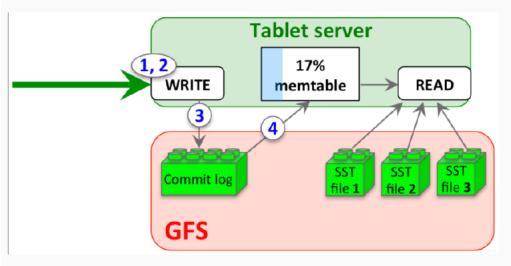
1. Master grabs a unique master lock in Chubby to prevent concurrent master instantiations
2. Master scans the servers directory in Chubby to find the live tablet servers
3. Master communicate with every live tablet servers to discover what tablets are already assigned to each server
4. Master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet is not discovered in step 3
5. Master scans the METADATA table to learn the set of tablets (and detect unassigned tablets)

Tablet merging/splitting The set of existing tablets only changes when a tablet is created or deleted. Master initiates Tablets merging, while tablet server initiate tablet splitting.

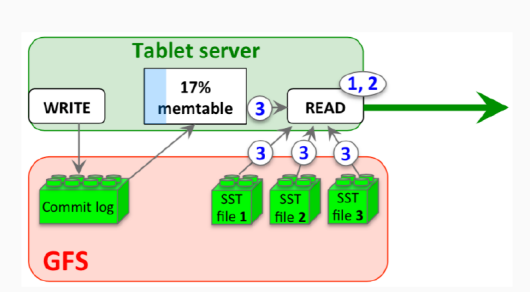
Tablet serving

1. Write operation
2. Read operation
3. Tablet recovery

1 - Write operation
Firstly, in the write operation Server checks request is well-formed, checks sender is authorized to write, valid mutation is written to commit log that stores redo records and after mutation is committed, its contents are inserted into memtable.



2 - Read operation
Secondly, in the read operation Server checks that the request it is well-formed, checks that the sender is authorized to read, valid that read operation is executed on a merged view of the sequence of SSTables and the memtable.



3 - Tablet Recovery For the Tablet Recovery: Tablet server reads its metadata from the METADATA table, then reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have a committed since the redo points.

Compactions There are three type of compactions:

1. Minor compactions
2. Merging compactions
3. Major compactions

1 - Minor Compaction When memtable size reaches a threshold, memtable is frozen, a new memtable is created, and the frozen memtable is converted to a new SSTable and written to GFS. So the goal is reduce the memory usage of the tablet server and the amount of data that has to be read from the commit log during a recovery.

2 - Merging Compaction The problem is that every minor compaction creates a new SSTable. The solution in this case is a periodic merging of a few SSTables and the memtable.

3 - Major Compaction The Major compaction is a merging compaction that rewrites all SSTables into exactly one SSTable that contains no deletion information or deleted data. Bigtable cycles throught all of it tablets and regularly applies major compaction to them.

5 Resource Allocation

5.1 Resource Allocation - Why?

Several benefits:

- Save money
- Improve productivity
- Saving time
- Effective use of resources

5.2 Resource Allocation - Problem

Given a type of resource whose total amount is equal to N, we want to allocate it to n activities so that a certain **objective function** is minimised (or maximised)

Objective function: cost or loss (profit or reward) of the resulting allocation

5.2.1 Problem 1

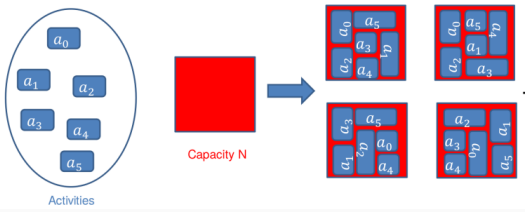


Figure 5.1: 1st Problem - Each configuration: specific objective value

Formally:

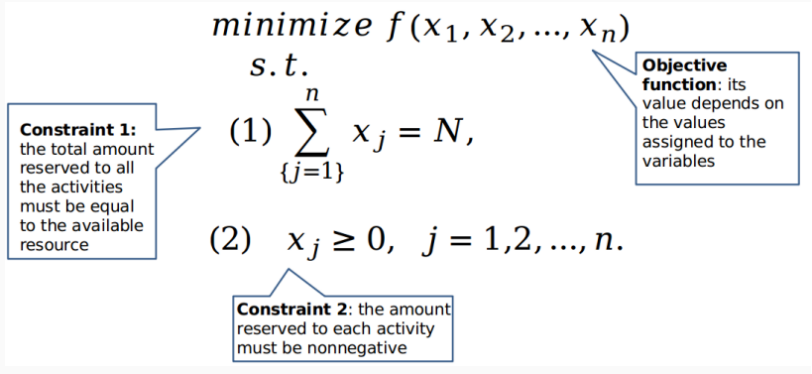


Figure 5.2: 1st Problem - Formally

Example

Distribution of search effort:

- Detect the position of an object whose possible positions are among those numbered from 1 to n
- The probability of the object being in position j is pj
- $(1 - e^{-\alpha x_j})p_j$: conditional probability of detecting the object in position j
- x_j : amount of search effort
- **Constraint**: N is the total amount of effort
- **Objective**: maximize the overall probability of detecting the object

Resource Distribution Problem:

- n locations to which resources such as newspapers are distributed from the central factory
- Demand at each location j
- $q_j(x_j)$: the expected cost at location j when x_j is allocated to j
- N: total amount of resources allocated to all locations

$$\begin{aligned} &\textit{minimize} \sum_{\{j=1\}}^{n} q_j(x_j) \\ &\textit{subject to } \textstyle\sum_{\{j=1\}}^{n} x_j = N, \\ &x_j \geq 0. \end{aligned}$$

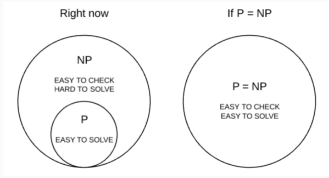
Online Caching Problem

- Requests from a catalog of N files $\mathbb{N} = 1, \ldots, N$
- $w_i \in \mathbb{R}+$: cost to serve a request for file i by a remote server
- Single-cache system with capacity K
- Storage: arbitrary fractions of files
- Slotted time $t \in \mathbb{T} = 1, \ldots, T$
- Cache state $x_t = [x_{t,i}]i\in\mathbb{N}$ drawn from

$$\mathcal{X} = \left\{ \mathbf{x} \in [0,1]^N : \sum_{i=1}^{N} x_i = K \right\},$$

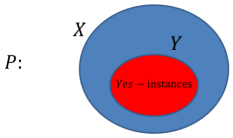
where $x_{i,t}$: fraction of file i stored at time t

5.2.2 P VS NP



5.2.3 Decision Problem

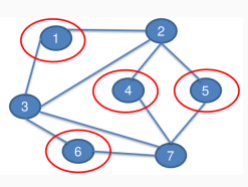
Definition (Decision Problem) A decision problem is a pair $P = (X, Y)$, where X is a language decidable in polynomial time and $Y \subseteq X$.
Elements of X: **instances** of P;
Elements of Y : **yes-instances**



Example

Problem (Independent Set) Given a graph G, a k-independent set is a subset of its vertices of cardinality k, where no couple of vertices is connected by an edge in G. (4-independent set)

- Instances: [G, k]
- Yes-instances: [G, k] s.t. G contains an independent set of k vertices

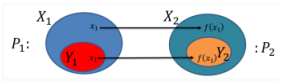


5.2.4 Polynomial Reduction

Definition (Polynomial Transformation) Let $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$ be decision problems. P_1

polynomially transforms to P_2 ($P_1 \leq pP_2$) if there is a function f : $X_1 \to X_2$, computable in polynomial time, s.t. $f(x_1) \in Y_2, \forall x_1 \in Y_1 \wedge f(x_1) \in X_2/Y_2, \forall x_1 \in X_1/Y_1$.

Proposition If P1 $\leq$p P2 and there is a polynomial time algorithm for P2, then there is a polynomial time algorithm for P1



5.2.5 NP-completeness

Definition A problem P is NP-complete if:

- $P \in NP$
- $\forall P^1 \in NP, P^1 \leq pP$ (NP-hardness)

Theorem SATISFIABILITY is NP-complete (Cook [1971])

5.2.6 NP-hard Problems

Knapsack:

- Instance: A number $n \in \mathbb{N}$ and nonnegative integers c_i , w_i and W for $i = 1, \ldots, n$.
- Task: Find a subset $S \subseteq 1, ..., n$ such that $\sum_{i \in S} w_i \leq W and \sum_{i \in S} c_i$ is maximum.
- Maximize the total value of collected items without exceeding the knapsack's capacity

Bin-Packing:

- Instance: A list of nonnegative numbers $a_1, ..., a_n \leq 1$.
- Task: Find a $k \in \mathbb{N}$ and an assignment
  $f : 1, ..., n \to 1, ..., k$ with
  $\sum_{i:f(i)=j} a_i \leq 1, \forall j \in 1, ..., k$ s.t. k is minimum.
- Packing all the items using the minimum number of bins

Subgraph Isomorphism

- Instance:Two graphs $G = (V, E)$ and $H = (V^1, E^1)$.
- Task: Find a subgraph $G_0 = (V_0, E_0) : V_0 \subseteq V, E_0 \subseteq E$ s.t. $G_0 \simeq H$.

Theory: We cannot find poly-time algorithms for NP-hard problems (unless P = NP)

Desired features:

- Optimality
- Poly-time
- Arbitrary instances of the problem

Trade-off:

- Quality of the solution
- Efficiency

5.3 Approximation

Approximation algorithms

- Approximation guarantees
- Approximation proof
- General case

Efficient Heuristics

- No approximation guarantees
- Experimental proof of approximation
- Specific case (most of the time)

Note: hard to find good approximation algorithms for several NP-hard problems

5.4 Resource allocation in the Cloud

Splitting and assign physical server to VMs. The goal is try to minimize the costs (less active server because they consume a lot of energy).

Typical approaches:

- Static Allocation
- Dynamic Allocation

5.4.1 Static Allocation

Workloads are predicted but the problem is NP-hard. We can use two approaches: mathematical solution but it isn't scalable and a approximate solution.

5.4.2 Dynamic Allocation

It is a most efficient solution. Controller can respond to over/under load on a server in runtime, possible through the service live migration (Moving VMs from one physical machine to another without disrupting services -¿ high availability).

We have two type of controls:

- Reactive control: A migration is triggered if the utilization of a server exceeds or falls below a certain threshold
- Proactive control: only trigger migration if the forecast suggest we are exceeding the capacity in the near future

5.4.3 Static vs Dynamic

Suggestions: good combination of both mechanisms Static allocation has long term period allocation, while Dynamic has exceptional workload peaks.

Kubernetes, open-source system for automatic deployment, scaling, and management of containerized applications, uses a heuristic solution to allocate pods in servers.

5.5 Resource allocation in the Fog

Extended cloud computing(higher than Fog).
Fundamental role in the trade-off between resource utilization and application requirements satisfaction

5.5.1 Problems

- Cloud: far (high latency)
- High costs
- Privacy issues
- Network congestions

5.5.2 Fog Computing

- Extension of cloud computing
- Fog nodes: access points, routers, switches, base stations, servers
- Computation closer to things
- Geographical distribution
- Reduction of data traffic towards the cloud
- Low latency
- Support for mobility
- Reduced costs
- Privacy improved

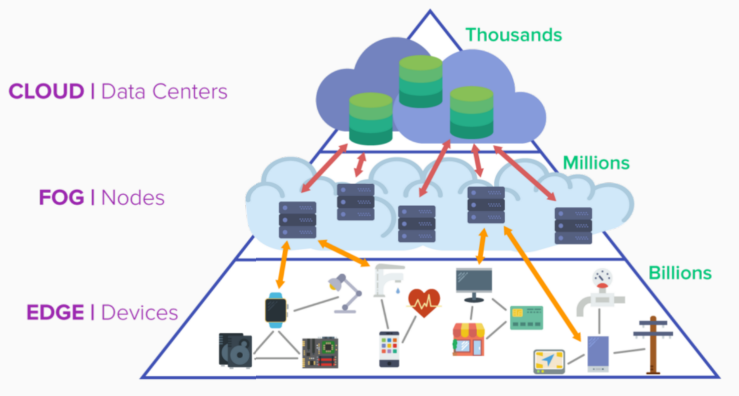


Figure 5.3: Fog Computing Schema

5.5.3 Static Allocation

- Applications and Network graphs
- Application deployment reduces to Virtual Network Embedding (VNE) problem (variant of Subgraph Isomorphism): NP-hard
- Approximation algorithms
- Heuristics:
  - Node ranking for each application module
  - Routing between deployed modules

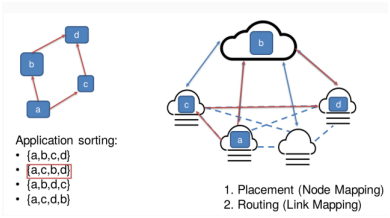


Figure 5.4: Static Allocation

5.5.4 Dynamic Allocation

- Different motivations w.r.t. the Cloud
- Live migration is applied to:
  - Deal with the geographical distribution
  - Save money on the cloud
  - Better utilize fog resources
  - Guarantee a good level of QoS

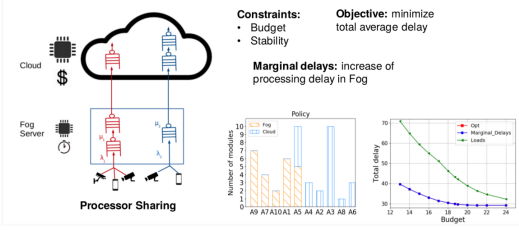


Figure 5.5: Dynamic Allocation

5.5.5 Static vs Dynamic in Fog

- Complete knowledge of the application requirements and network availability
- Optimization problem
- NP-hard problems
- Approximated solutions
- Partial knowledge
- Variability of applications performances
- Live migration
- Monitoring systems
- Application scaling

5.5.6 Cost-Effective Workload Allocation Strategy

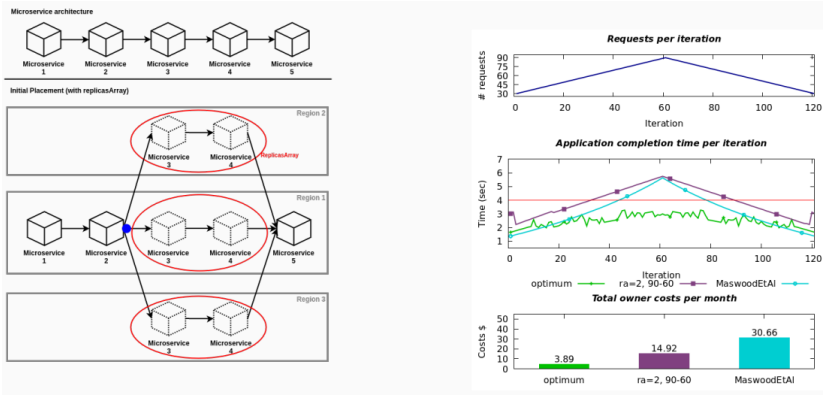


Figure 5.6

- Orchestration problem with replicas
- NP-hard problem
- Trade-off: Cost vs. Completion Time

6 Laboratory

6.1 Vagrant

Vagrant is an open source virtual machine manager; it works with many hypervisor software, called providers, including VirtualBox, VMware and KVM. Vagrant, along with Docker is one of the leading software for portable deployment of application development environments.

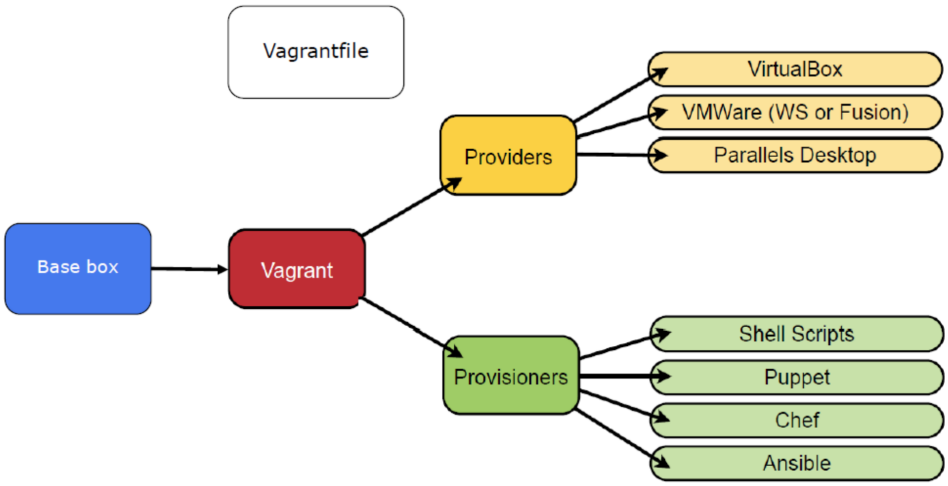


Figure 6.1: Vagrant Schema

6.1.1 Vagrant Commands

- **vagrant help** - Get info on all commands
- **vagrant COMMAND –h** - Get help for COMMAND
- **vagrant init** [**box**] - Create a Vagrantfile
- **vagrant up** - Create and provision the VM
- **vagrant provision** - Only provision the VM
- **vagrant ssh** - Enter the VM via SSH
- **vagrant status** - Get VM status
- **vagrant global-status** - Get all VM status
- **vagrant plugin list** - List all plugins
- **vagrant box list** - List all boxes

6.1.2 SSH Socks Proxy Tunnel

A client machine, while opening the ssh session, instructs the SSH server to open tunnels that can run in both directions.

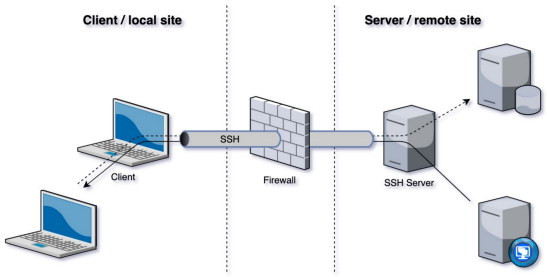


Figure 6.2: Proxy Tunnel

6.1.3 Configuration Management Systems

- In DevOps, permits to maintain OS configuration files
  - Configuration management (CM) is a systems engineering process for establishing and maintaining consistency of a product
- Used with IaC – Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine
  - readable definition files
- Track changes in VCS, like git

6.2 Ansible

Ansible Automation Platform is a framework for building and operating automation across an organization. The platform includes all the tools to implement enterprise-wide automation.

6.2.1 Ansible Architecture

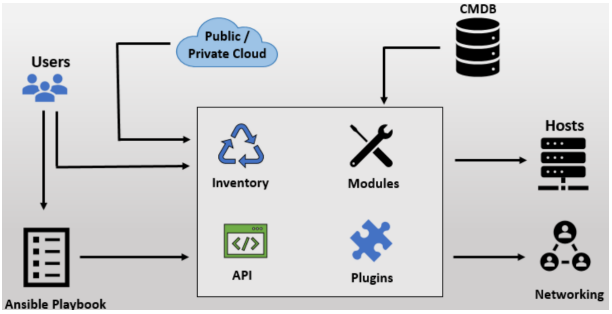


Figure 6.3: Ansible Architecture

6.2.2 Ansible Features

- Proprietary / GNU GPL
- Huge docs and community
- Idempotent
- Extensible
- Mostly used for CM
- Can do also IaC
- YAML syntax
- Written in Python
- agent-less
- Based on SSH

6.2.3 Ansible Playbook

$ ansible-playbook playbook.yml
$ –f 10

Ansible Playbooks offer a repeatable, re-usable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications. If you need to execute more than one task with Ansible, write a **playbook** and put it under source control. Then you can use it to push out new configuration or confirm the configuration of remote systems. Playbooks can:

- Declare configurations
- Orchestrate steps of any manual ordered process, on multiple sets of machines (defined order)
- Launch tasks synchronously or asynchronously

6.2.4 Ansible Inventory

$ ansible-playbook –i inventory
$ playbook.yml
$ –f 10

Ansible works against multiple managed nodes or “hosts” in your infrastructure at the same time, using a list or group of lists known as inventory. Once your inventory is defined, you use patterns to select the hosts or groups you want Ansible to run against.
The default location for inventory is a file called */etc/ansible/hosts*. You can specify a different inventory file at the command line using the -i <path> option. You can also use multiple inventory files at the same time as described using multiple inventory sources, and/or pull inventory from dynamic or cloud sources or different formats (YAML, ini, etc)

6.3 Containers vs Hypervisors

Containers are an application-centric way to deliver high-performing, scalable applications on the infrastructure of your choice.

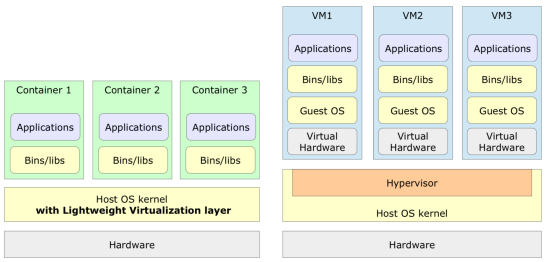


Figure 6.4: Container VS Hypervisors (infrastructure)

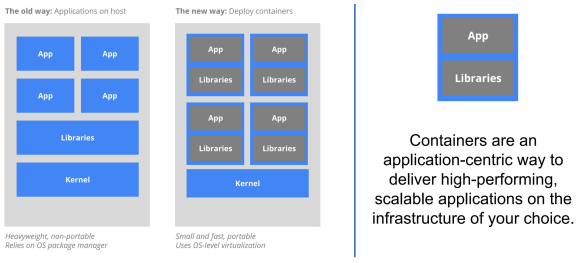


Figure 6.5: Container VS Hypervisors (software)

6.3.1 Containers Benefits

- Agile application creation and deployment: Increased ease and efficiency of container image creation and deployment compared to VM.
- Continuous development, integration, and deployment: Provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
- Dev and Ops separation of concerns: Create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- Observability Not only surfaces OS-level information and metrics, but also application health and other signals.
- Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-prem, Google Kubernetes Engine, and anywhere else.
- Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- Loosely coupled, distributed, elastic, liberated micro-services: Applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- Resource utilization: High efficiency and density.

6.3.2 Containers disadvantages

- Less isolation/security
  - Others weaknesses: Images distribution, Image scanning
- A solution
  - Combine Virtual Machines and Containers
- Kata containers
  - Kata Containers is an open source community working to build a secure container runtime with lightweight virtual machines that feel and perform like containers, but provide stronger workload isolation using hardware virtualization technology as a second layer of defense.

6.4 Docker

- OS level virtualization (lightweight)
- Relies on Linux kernel features: cgroups and namespaces
- Layered filesystem (similar as git commit)
  - Images as packaged containers derived incrementally from a preexisting one
- Enable:
  - DevOps
  - Microservice architecture
  - Portability

6.4.1 Images VS Containers

To use a computer science metaphor, if an image is a class, then a container is an instance of a class, in other words a runtime object.

Docker Images

- A read-only template with instructions for creating a Docker container.
  - Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application.
- You might create your own images or you might only use those created by others and published in a registry.
- To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image.
- It is the object that makes your application portable.

Docker Containers

- A runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
- By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

6.4.2 Docker Main Components

- **Docker daemon** - The main process that manages containers
- **Docker Host** - The host (physical or virtual) where Docker daemon runs
- **Docker client** - For communicating with the Docker daemon
- **Docker registry** - Image repository

6.4.3 Docker Host Overview

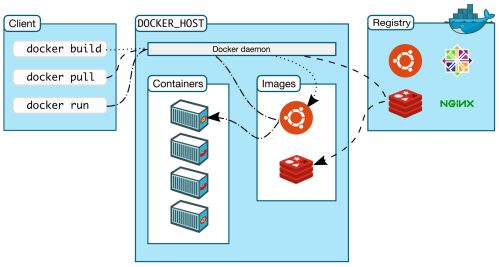


Figure 6.6: Docker Host Overview

6.4.4 Docker related Tools

- **Docker compose** - For deploying multi-container apps
- **Docker machine** - For setting up remote Docker Hosts
- **Docker swarm** - Container orchestrator

6.4.5 Container Lifecycle

- Create - **docker create** <image>
- Start - **docker start** <container id>
- Stop - **docker kill , docker stop** <container id>
- Restart - **docker restart** <container id>
- Remove - **docker rm** <container id>

6.4.6 Some Basic Commands

$ docker <command> –help	
$ docker run [-it -d] <image>	(docker container run)
$ docker ps [-a]	(docker container ls)
$ docker images	(docker image ls)
$ docker rm <container id>	(docker container rm)
$ docker rmi <image name>	(docker image rm)

6.4.7 Debugging and Logging

- docker inspect (info about a container)
  - Tip: **docker inspect <container_id> — jq -C .[ ] — less –RN**
- docker stats (statistics on ram, cpu etc)
- docker events (Docker Host related events)
- docker logs (get logs of apps inside a container)

6.4.8 Run in Interactive or Detached Mode

Interactive	**Detached**
$ *docker run -it <image> <args>*	$ *docker run -d -p <port host>:<port guest> <image> <args>*

6.4.9 Build a Docker image with a Dockerfile

DockerFile

FROM <original image>
ADD <filename> <destination path>
RUN <command>
...
EXPOSE 80

Run the new image

$ docker build -t <image namespace>/<image name>
$ docker images
$ docker run -ti <image namespace>/<image name>

6.4.10 Docker Netrworking

Docker offers many network functionalities:

- Bridge Network (used with single host, default and user defined).
  docker network create mynet
- Host Network (No isolation, *–net=host*)
- None (No network *–net=none*)
- Overlay Network (among different hosts)

It offers also DNS service

- Across containers on same network
- Default using container name
- *–alias to customise*

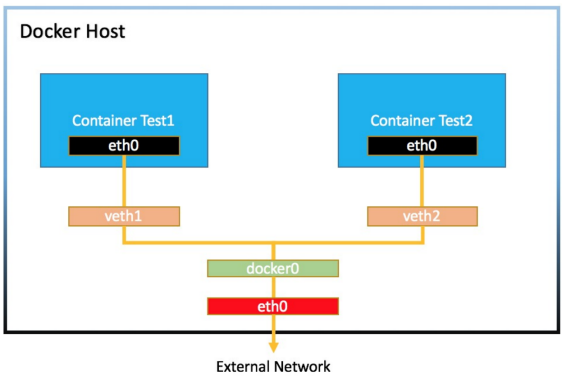


Figure 6.7: Docker default bridged network (docker0) schema

If i did *ip link* i'll see all the interfaces, for example "docker0"

Interconnecting two (or more) containers

1. **Deprecated**:
   $ docker run –link <container_name>:<alias>
   Note: Use docker network connect instead

2. Use **Bridge Network**
   Connect two or more containers to the same network (default docker0 or user defined)

3. Use **Overlay Network Connect** two or more containers across different hosts

6.4.11 Docker Volumes

Containers has an Ephemeral disk. Docker offers few storage features:
Mount a data volume from a Docker Host on a container or **create** a Docker volume

Bind Mount and Volumes

Mounting a host directory into a container.
$ docker run ***-v <host path>:<container path>*** *<image>*
Creating a volume and sharing it
$ docker volume create my_vol
$ docker run -v my_vol:/data <image>

6.5 Cloud Native

Definition: Cloud Native is a collection of patterns in order to build software working in a cloud that scale on elastic infrastructure. It's about 2 main concepts:

1. The twelve factors methodology (DEV)
2. Pets Vs Cattle (OPS)

1 - The twelve factors methodology

Set of rules written by people working at the Heroku platform. It's methodology for writing apps, that uses backing services. **First 3 of The twelve factors are:**

1. **Codebase** - One codebase tracked in revision control, many deploys
2. **Dependencies** - Explicity decalre and isolate dependecies (a way is a Docker Image)
3. **Config** - Store config in the environment(not inside repo or codebase)

2 - Pets VS Cattle

Bill used an analogy that resonated deeply with me. He talked about scale-up vs. scale-out in the context of comparing a pet and a herd of cattle. Put pets vs cattle in the context of cloud, and second emphasize the disposability of cattle and the uniqueness of pets. This was much more important than whether you scale-up or scale-out. Those are side effects of how you view a server. In the old way of doing things, we treat our servers like pets, for example Bob the mail server. If Bob goes down, it's all hands on deck. The CEO can't get his email and it's the end of the world. In the new way, servers are numbered, like cattle in a herd. For example, www001 to www100. When one server goes down, it's taken out back, shot, and replaced on the line.

If you view a server (whether metal, virtualized, or containerized) as inherently something that can be *destroyed and replaced* at any time, then it's a member of the herd(**cattle**).

If you view a server (or a pair of servers attempting to appear as a single unit) as *indispensable*, then it's a **pet**.

Pets Servers or server pairs that are treated as indispensable or unique systems that can never be down. Typically they are manually built, managed, and "hand fed". Examples include mainframes, solitary servers, HA loadbalancers/firewalls (active/active or active/passive), database systems designed as master/slave (active/passive), and so on.

Cattle Arrays of more than two servers, that are built using automated tools, and are designed for failure, where no one, two, or even three servers are irreplaceable. Typically, during failure events no human intervention is required as the array exhibits attributes of "routing around failures" by restarting failed servers or replicating data through strategies like triple replication or erasure coding. Examples include web server arrays, multi-master datastores such as Cassandra clusters, multiple racks of gear put together in clusters, and just about anything that is load-balanced and multi-master.

6.6 OpenStack

Open source software for creating private and public clouds.

- OpenStack software controls large pools of compute, storage, and network-ing resources throughout a datacenter, managed through a dashboard or via the OpenStack API. OpenStack works with popular enterprise and open source technologies making it ideal for heterogeneous infrastructure.
- OpenStack Community
  - Wiki, Specs, Projects, RC meetings, gerrit, OpenStack Foundation
- Four "open"s
  - Open Source, Open Design, Open Development, Open Community

6.6.1 OpenStack Identity Management (Keystone)

Keystone is an OpenStack service that provides:

- API client authentication
- Service discovery
- Distributed multi-tenant authorization

by implementing OpenStack's Identity API.

6.6.2 OpenStack Dashboard (Horizon) + CLI

Horizon is the canonical implementation of OpenStack's **Dashboard**, which provides a web based user interface to OpenStack services including Nova, Swift, Keystone, etc.
OpenStack offers also a **CLI**.
Both relies on top of the OpenStack API interfaces.
The OpenStack Dashboard is a *web-based interface* that allows you to **manage** OpenStack resources and services. The Dashboard allows you to **interact** with the OpenStack Compute cloud controller using the Open- Stack APIs.
Dashboard is available with different functionalities for users (tenants) and for admins.

6.6.3 Compute service (Nova) - Security Group

The OpenStack Compute service allows you to control an Infrastructure-as-aService (IaaS) cloud computing platform.
It gives you control over instances and networks, and allows you to manage access to the cloud through users and projects.
Compute doesn't include virtualization software. Instead, it defines drivers that interact with underlying virtualization mechanisms that run on your host operating system, and exposes functionality over a web-based API.
Group different resources to assign to your VM under a name.
Common resources: RAM, DISK, num of CPU (and usually a cost per Hour, Month or Year).
Think about them as a Firewall Rules managed by OpenStack for Group of Hosts.
Default policy is deny. Everything must be allowed explicitly.
An host can be part of multiple Security Groups.

6.6.4 Network Service(Neutron)

The Networking service, code-named neutron, provides an API that lets you define network connectivity and addressing in the cloud.
The Networking service enables operators to leverage different networking technologies to power their cloud networking.
The Networking service also provides an API to configure and manage a variety of network services ranging from L3 forwarding and NAT to load balancing, edge firewalls, and IPsec VPN.

6.6.5 Image Service(Glance)

The Image service (glance) enables users to discover, register, and retrieve virtual machine images. It offers a REST API that enables you to query virtual machine image metadata and retrieve an actual image. You can store virtual machine images made available through the Image service in a variety of locations, from simple file systems to objectstorage systems like OpenStack Object Storage.

6.6.6 Block Storage service (Cinder)

The OpenStack Block Storage service (cinder) works through the interaction of a series of daemon processes named cinder-* that reside persistently on the host machine or machines. Can be used as main instance disk or as an external disk (like a USB key)

6.6.7 Object Storage Service (Swift)

OpenStack Object Storage (swift) is used for redundant, scalable data storage using clusters of standardized servers to store petabytes of accessible data.
It is a long-term storage system for large amounts of static data which can be retrieved and updated via REST-API.
Object Storage uses a distributed architecture with no central point of control, providing greater scalability, redundancy, and permanence.

- Unstructured data such as music, images, and videos
- Backup and log files
- Large sets of historical data
- Archived files

6.7 Container Orchestration

In Development and Quality Assurance (QA) environments, we can get away with running containers on a single host to develop and test applications. However, when we go to Production, we do not have the same liberty, as we need to ensure that our applications:

- Are fault-tolerant
- Can scale, and do this on-demand
- Use resources optimally
- Can discover other applications automatically, and communicate with each other
- Are accessible from the external world
- Can update/rollback without any downtime

Container Orchestrators are the tools which group hosts together to form a cluster, and help us fulfill the requirements mentioned before. (Everything at Google runs in a container)

Why use Container Orchestrators? It is all about SCALING

- Bring multiple host togheter and make then part of a cluster
- Schedule containers to run on different hosts
- Help containers running on one host reach out to containers running on other hosts in the cluster
- Bind containers and storage
- Bind containers of similr type to higher-level construct, like services, so we don't have to deal with individual containers
- Keep resource usage in-check, and optimize it when necessary
- Allow secure access to applications running inside containers

6.8 K8s

What is Kubernetes, aka k8s ?

- From k8s website: "Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications."
- Highly inspired by the Google Borg system
- Started by Google and (from v1.0 release in July 2015) donated to the CNCF
- k8s community defined it: "Platform to build other platforms" → CRD (Custom Resource Definition)

6.8.1 K8S - Architecture

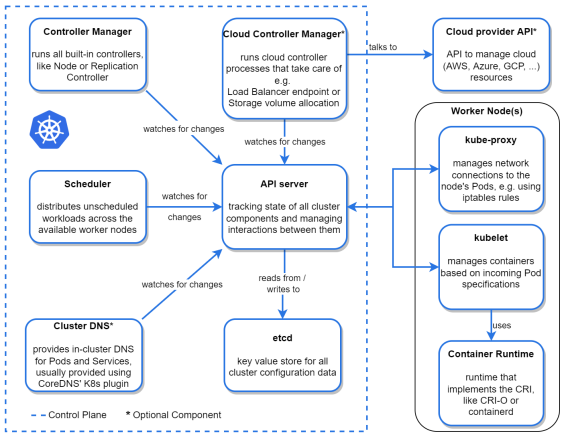
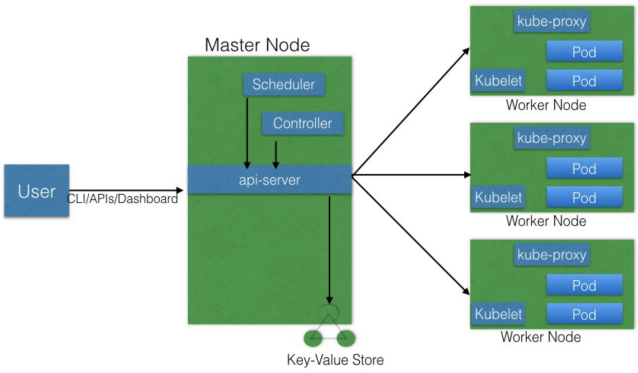


Figure 6.8: K8S Architecture



6.8.2 K8S - Master Node

- The Master Node is responsible for managing the Kubernetes cluster, and it is the entry point for all administrative tasks. We can communicate to the Master Node via the CLI, the GUI (Dashboard), or via APIs
- If we have more than one Master Node, they would be in a HA (High Availability) mode, and only one of them will be the leader, performing final operations and decisions
- To manage the cluster state, Kubernetes uses etcd, and all Master Nodes connect to it. etcd is a distributed key-value store

Master Node Components

API Server: A user/operator sends REST commands to the API Server, which then validates and processes the requests. After executing the requests, the resulting state of the cluster is stored in the distributed key-value store.

Scheduler: Schedules the work to different Worker Nodes. The Scheduler has the resource usage information for each Worker Node. Before scheduling the work, it also takes into account the quality of the service requirements, data locality, affinity, anti-affinity, etc. eg: disk==ssd

Controller Manager: Manages different non-terminating control loops, which regulate the state of the Kubernetes cluster. Each control loop knows about the desired state of the objects it manages, and watches their current state through the API Server. If the current state of the object does not meet the desired state, then it takes corrective steps to make sure that the current state is the same as the desired state.

etcd: Distributed key-value store used to store the cluster state. It can be part of the Kubernetes Master or can be configured externally (Master Nodes would connect to it)

6.8.3 K8S - Worker Node

- A Worker Node is a machine (VM, physical server, etc.) which runs the applications by means of Pods and is controlled by the Master Node.
- Pods are scheduled on the Worker Nodes, which have the necessary tools to run and connect them.
- A Pod is the scheduling unit in Kubernetes. It is a logical collection of one or more containers which are always scheduled together.
- To access the applications from outside, we connect to Worker Nodes and not to the Master

Worker Node Components

Container Runtime: To run containers, we need a Container Runtime on the Worker Node. By default, Kubernetes is configured to run containers with Docker. It can also run containers using the rkt Container Runtime.

kubelet: An agent which runs on each Worker Node and communicates with the Master Node. It receives the Pod definition via various means (primarily, through the API Server), and runs the containers associated with the Pod. It also makes sure the containers which are part of the Pods are healthy at all times.

kube-proxy: Instead of connecting directly to Pods to access the applications, we use a logical construct called a Service as a connection endpoint. A Service groups related Pods, which it load balances when accessed. kube-proxy is the network proxy which runs on each Worker Node and listens to the API Server for each Service endpoint creation/deletion. For each Service endpoint, kube-proxy sets up the routes so that it can reach to it.

State Management

- etcd is a distributed key-value store based on the *Raft Consensus Algorithm.*
- Allow a collection of machines to work as a coherent group that can survive failures
- At any given time one node is the master, the rest are followers
- Any node can be the Master

6.8.4 K8S - Object Model

- Kubernetes has a very rich (and now extensible) object model by means of which it represents different persistent entities in the cluster. Those entities describes:
  - What containerized applications we are running and on which node
  - Application resource consumption
  - Different policies attached to applications, (restart/upgrade policies, fault tolerance)
- With each object, we declare our intent or desired state using the spec field. The Kubernetes system manages the status field for objects, in which it records the actual state of the object.
- At any given point in time, the Kubernetes Control Plane tries to match the object's actual state to the object's desired state.
- Examples of Kubernetes objects are Pods, Deployments, ReplicaSets
- Most of the time, we provide an object's definition in a .yaml file, which is converted by kubectl in a JSON payload and sent to the API Server.
- Within each object:
  - We declare our intent or desired state using the spec field
  - Kubernetes records the actual state in the status

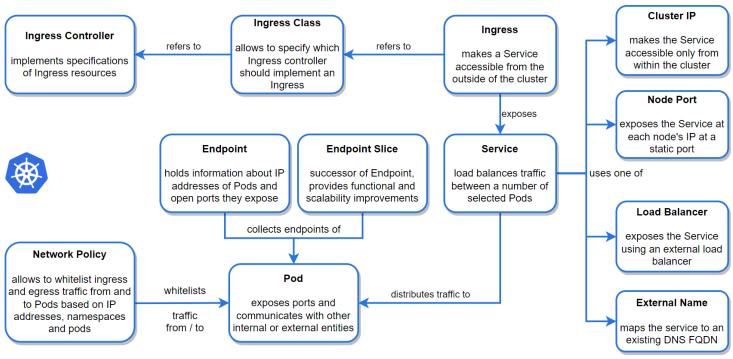


Figure 6.9: K8S Networking Objects

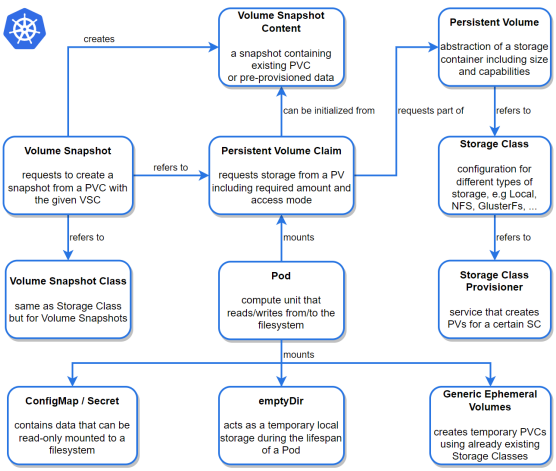
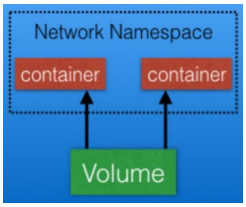


Figure 6.10: K8S Storage Objects

6.8.5 K8S Object - POD

The smallest and simplest Kubernetes object.
It is the unit of deployment in Kubernetes, which represents a single instance of the application (microservice).

- Is a logical collection of one or more containers, which:
  - Are scheduled together on the same host
  - Share the same network namespace
  - Mount the same external storage (Volumes).

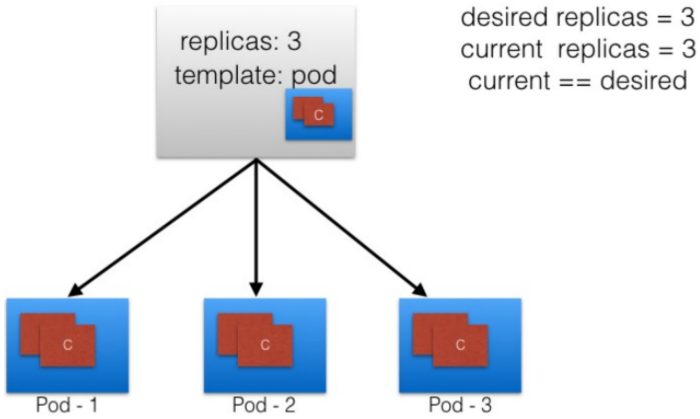


- Pods are ephemeral in nature, and they do not have the capability to self-heal by themselves.
- We use controllers to manage them. Examples of controllers are:
  - Deployments
  - ReplicaSets
  - ReplicationControllers

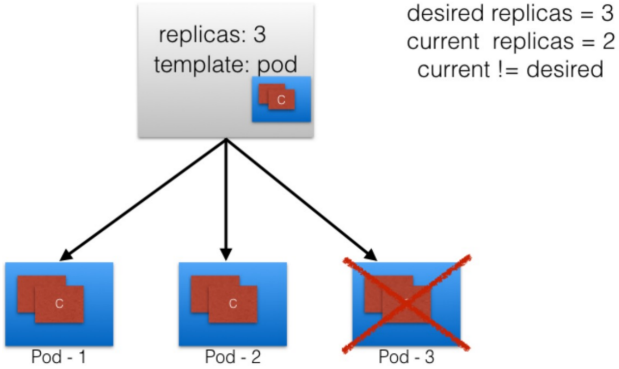
6.8.6 K8S Object - ReplicaSets

- A ReplicaSet Controller is part of the Master Node's Controller Manager. It makes sure the specified number of replicas for a Pod is running at any given point in time looking at the ReplicaSet resources.
  - If there are more Pods than the desired count, the ReplicationController would kill the extra Pods
  - If there are less Pods, then the ReplicationController would create more Pods to match the desired count.
- Generally, we don't deploy a Pod independently, as it would not be able to re-start itself, if something goes wrong.
  We always use controllers like ReplicaSet to create and manage Pods.
- A ReplicaSet (rs) is the next-generation ReplicationController. ReplicaSets support both equality- and set-based Selectors, whereas ReplicationControllers only support equality-based Selectors. Currently, this is the only difference.

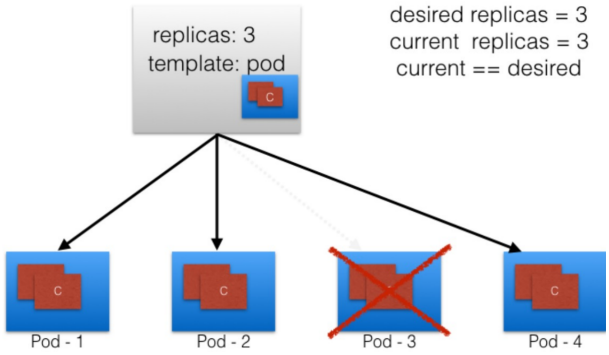
1/3 - Lifecycle of a ReplicaSet:
We set the replica count to 3 Pods in the ReplicaSet spec section under the attribute replicas.



2/3 - Lifecycle of a ReplicaSet:
If one Pod dies, our current state is not matching the desired state anymore.



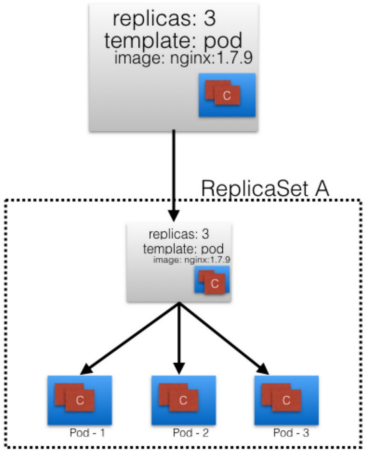
3/3 - Lifecycle of a ReplicaSet:
The ReplicaSet detects that the current state is no longer matching the desired state. So, it will create one more Pod, thus ensuring that the current state matches the desired state. We can use ReplicaSet independently but most of the time we use Deployments to manage the Pods: creation, deletion and updates.



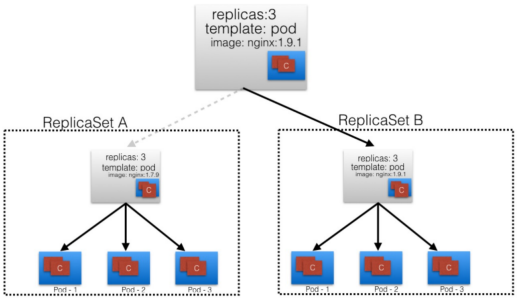
6.8.7 K8S Object - Deployments

- Deployment objects provide declarative updates to Pods and ReplicaSets. The DeploymentController is part of the Master Node's Controller Manager, and it makes sure that the current state always matches the desired state.
- On top of ReplicaSets, Deployments provide features like Deployment recording, with which, if something goes wrong, we can rollback to a previously known state.

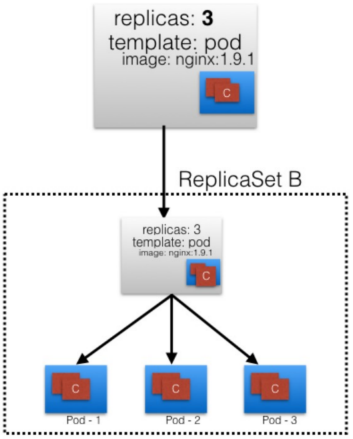
1/3 - Lifecycle of a Deployment:
One Deployment creates a ReplicaSet A.
ReplicaSet A creates 3 Pods:
Each Pod uses nginx:1.7.9 as container image



2/3 - Lifecycle of a Deployment:
The Pod template section of the Deployment is changed, the nginx container image is renamed from nginx:1.7.9 to nginx:1.9.1
As Deployment is modified a new ReplicaSet B is created



3/3 - Lifecycle of a Deployment:
Deployment now point to ReplicaSet B.
By means of ReplicaSets, Deployments provide features like Deployment recording, if something goes wrong, we can rollback to a previously known state.

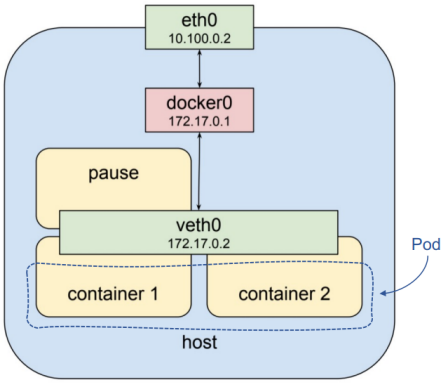


Container Network Interface (CNI)

- In Kubernetes, each Pod gets a unique IP address.
- For container networking, there are two primary specifications:
  - Container Network Model (CNM), proposed by Docker
  - Container Network Interface (CNI), proposed by CoreOS (used by k8s)
- The Container Runtime offloads the IP assignment to CNI, which connects to the underlying configured plugin, like Bridge or MACvlan, to get the IP address. Once the IP address is given by the respective plugin, CNI forwards it back to the requested Container Runtime.

Communication - Container-to-Container inside a Pod

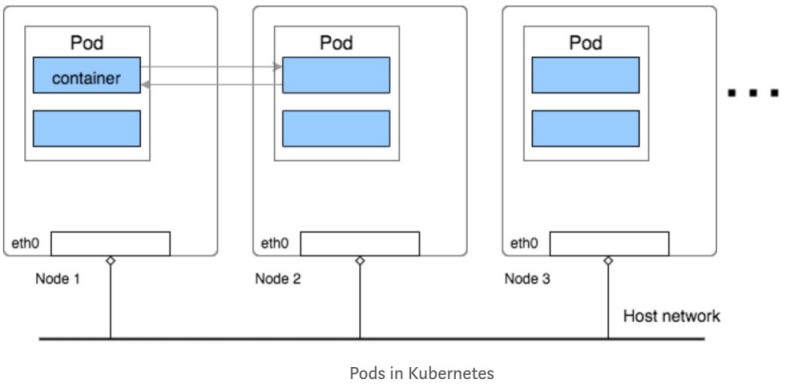
- With the help of the underlying Host OS, all of the Container Runtimes generally create an isolated network entity for each container that they starts.
- On Linux, that entity is referred to as a Network Namespace. These Network Namespaces can be shared across containers, or with the Host Operating System.
- *Inside a Pod, containers share the Network Namespaces*, so that they can reach to each other via **localhost**.
- **pause** is a special container that provides a virtual network interface for the other containers to communicate.



Communication - Pod-to-Pod Across Nodes

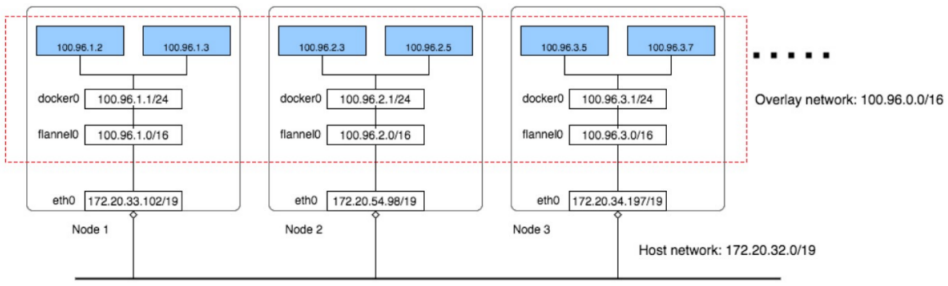
- In a clustered environment, the Pods can be scheduled on any node.
- We need to make sure that the Pods can communicate across the nodes, and all the nodes should be able to reach any Pod.
- Kubernetes also puts a condition that there shouldn't be any Network Address Translation (NAT) while doing the Pod-toPod communication across Hosts. We can achieve this via:
  - Routable Pods and nodes, using the underlying physical infrastructure, like Google Container Engine
  - Using Software Defined Networking, like Flannel, Weave, Calico, etc

6.8.8 K8S - Networking



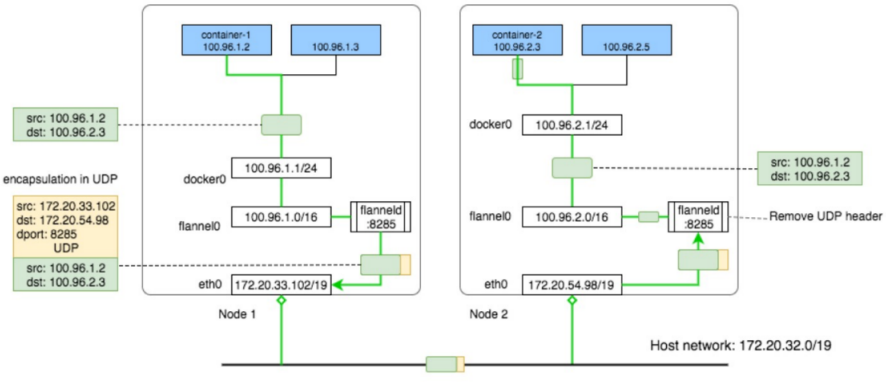
Pods in Kubernetes

Network Overlay



Flannel overlay network

Network Plugin - Flannel



Cross host communication

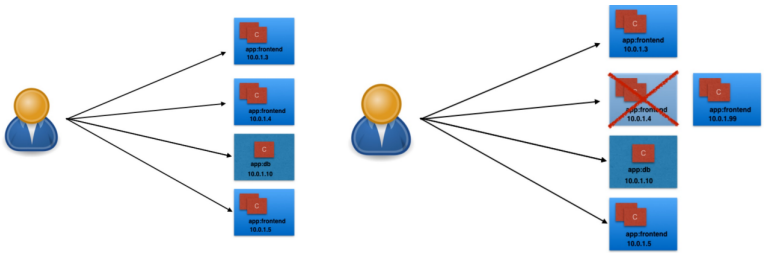
Flannel uses UDP encapsulation in order to encapsulate generic packets into UDP packets. Many implementations: IPSec, VXLan, others

External World-to-Pod Communication

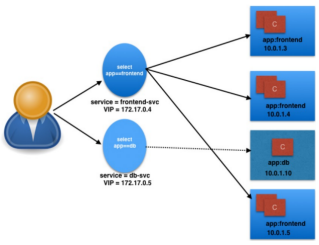
By exposing our services to the external world with kubeproxy, we can access our applications from outside the cluster. A Service is an high-level abstraction, which logically groups Pods and add policies to access them.
This grouping is achieved via Labels and Selectors

6.8.9 K8S Object - Service

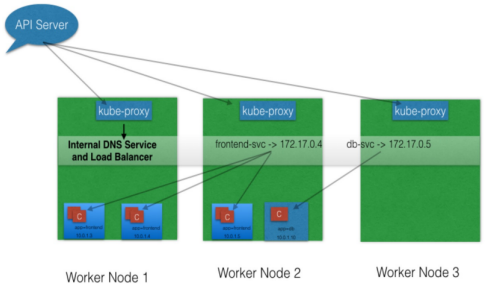
- Scenario in which a user/client is connected to a Pod using its IP address.
- If the Pod to which the user/client is connected **dies**, a new Pod is **created** by the controller.
- The new Pod will have a new IP address, which will not be known automatically to the user/client of the earlier Pod.



- Kubernetes provides a higher-level abstraction called Service, which logically groups Pods and a policy to access them.
- This grouping is achieved via Labels and Selectors. (e.g. Using Selectors **(app==frontend and app==db)**, we can group them into two logical groups: one with 3 Pods, and one with just one Pod.)
- We can assign a name to the logical grouping, referred to as a service name. In our example, we have created two Services, frontend-svc and db-svc, and they have the app==frontend and the app==db Selectors, respectively



- All of the Worker Nodes run a daemon called kube-proxy, which watches the API Server on the Master Node for the addition and removal of Services and endpoints.
- For each new Service, on each node, kube-proxy configures the IPTables rules to capture the traffic for its ClusterIP and forwards it to one of the endpoints. When the Service is removed, kube-proxy removes the IPtables rules on all nodes as well.



Service - Types

ClusterIP: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default ServiceType.

NodePort: Exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.

LoadBalancer: Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

ExternalName: Maps the service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

6.8.10 K8S Object - Namespaces

- If we have numerous users whom we would like to organize into teams/projects, we can partition the Kubernetes cluster into ≪sub-clusters≫ using Namespaces
- The names of the resources/objects created inside a Namespace are unique, but not across Namespaces
- Generally, Kubernetes creates two default namespaces:
  - kube-system: contains objects created by k8s system
  - default: contains objects which belong to any other user
- Using Resource Quotas, we can divide the cluster resources within Namespaces

Labels and Selectors

Labels:

- are key-value pairs that can be attached to any Kubernetes objects (e.g. Pods)
  - Labels are used to organize and group a subset of objects, don't provide uniqueness to objects
- Examples: *app=webserver*, *app=database*, *env=dev*,*env=prod*, *env=qa*

Selectors:

- We can select a subset of objects
  - **Equality-Based Selectors:** Filters based on label keys and values
- Operators: =, ==, != (e.g.: *env==dev*)
  - **Set-Based Selectors:** Filters based on a set of values
- Operators: in, noin, exists (e.g.: *env in (dev,qa)*)

6.8.11 K8S Object - ConfigMaps & Secrets

- While deploying an application, we may need to pass runtime parameters like configuration details, passwords, etc.
- In such cases, we can use the **ConfigMap** API resource.
- Similarly, when we want to pass sensitive information, we can use the **Secret** API resource.
- Both ConfigMaps and Secrtes can be created and retrieved in various ways
  - Created from literal values, from files and from directory of files...
  - Used via ENV_VARS, Volumes, etc

6.8.12 K8S Objects - Others & Features

Deployment Advanced Features: Autoscaling, Proportional Scaling, Pausing & Resuming

ConfigMaps and Secrets: Way to decouple configuration details from container image allowing to pass them as key-value pairs to k8s objects or system components. Also passing them as reference, controlling the usage and hiding the content. (3° of 12° Factor rules)

Ingress: Collection of rules that allow inbound connections to reach the cluster Services.

Jobs: Create one or more Pod to perform a given task. It makes sure the task is completed , then terminates the Pods. Cron Job is a Job on a time-based schedule

StatefulSets: For application that require a unique identity, like name, net id, strict ordering, e.g.: mysql or etcd cluster (it was called PetSet < 1.5)

DaemonSets: Special Pod that run on all nodes and is started/deleted automatically when node is added/removed

Quota Management: Limit resource consumption per namespace: Compute, Object Count, Storage

CRD: Create our own API objects and Controller that manages them

RBAC: Authorization mechanism for managing permissions around Kubernetes resources

Kubernetes Federation: Manage multiple Kubernetes clusters from a single control plane

6.8.13 K8S - Volumes

- Container are ephemeral in nature, so if it crashes all data stored in it is deleted. kubelet restarts it but without any of the old data inside
- To manage the storage and to overcome this k8s uses Volumes
- A Volume is essentially a directory backed by a storage medium. The storage medium and its content are determined by the Volume Type.
- A volume:
  - is shared among the containers of the same Pod
  - has the same lifetime as the Pod
  - it outlives the containers of the Pod

Volumes - Types

Volume Type decides the properties of the directory, types are:

emptyDir: An empty Volume is created for the Pod as soon as it is scheduled on the Worker Node. The Volume's life is tightly coupled with the Pod. If the Pod dies, the content of emptyDir is deleted forever.

hostPath: With the hostPath Volume Type, we can share a directory from the host to the Pod. If the Pod dies, the content of the Volume is still available on the host.

gcePersistentDisk: With the gcePersistentDisk Volume Type, we can mount a Google Compute Engine (GCE) persistent disk into a Pod.

awsElasticBlockStore: With the awsElasticBlockStore Volume Type, we can mount an AWS EBS Volume into a Pod.

nfs: With nfs, we can mount an NFS share into a Pod.

iscsi: With iscsi, we can mount an iSCSI share into a Pod.

secret: With the secret Volume Type, we can pass sensitive information, such as passwords, to Pods.

Volume management and API

- Kubernetes resolves the problem of the storage with the Persistent Volume subsystem
- Provides APIs for users and administrators to manage and consume storage:
  - To manage the Volume → PersistentVolume (PV) API resource type
  - To consume the Volume → PersistentVolumeClaim (PVC) API resource type

PersistentVolume and StorageClass

- A Persistent Volume is (usually) a network attached storage in the cluster
- A persistent Volume can be:
  - Statically provisioned by the administrator.
  - Dynamically provisioned based on the StorageClass resource.
- A StorageClass contains pre-defined provisioners and parameters to create a Persistent Volume.

PersistentVolumeClaim

A PersistentVolumeClaim (PVC) is a request for storage by a user

- Users request for Persistent Volume resources based on size, access modes, etc. via a PVC
- Once a suitable Persistent Volume is found, it is bound to a Persistent Volume Claim
- After a successful bind, the PersistentVolumeClaim resource can be used in a Pod.
- Once a user finishes its work, the attached Persistent Volumes can be released. The underlying Persistent Volumes can then be reclaimed and recycled for future usage.

Helm

Resources in k8s are described as YAML manifests

- We can bundle all those manifests after templatizing them into a welldefined format, along with other metadata
- Such a bundle is referred to as Chart
- These Charts can then be served via repositories, such as those that we have for rpm and deb packages

Helm is a package manager (analogous to yum and apt) for Kubernetes, which can help to define, install, update, delete those Charts (applications) in the Kubernetes cluster.

6.8.14 K8S Object - Ingress

- Ingress is another method we can use to access our applications from the external world.
- With Services, routing rules are attached to a given Service.
  - They exist for as long as the Service exists.
- With Ingress we can somehow decouple the routing rules from the application
  - When we update our application we do not worry about its external access rules
- An Ingress is a collection of rules that allow inbound connections to reach the cluster Services.
- To allow the inbound connection to reach the cluster Services, Ingress configures a **Layer 7 HTTP load balancer** for Services and provides the following:
  - TLS (Transport Layer Security)
  - Name-based virtual hosting
  - Path-based routing
  - Custom rules
- With Ingress, users don't connect directly to a service. Users reach the Ingress endpoint (**Ingress Controller**), and, from there, the request is forwarded to the respective service.
- Example of Ingress rules are:
  - Name-Based Virtual Hosting Ingress rule (based on the FQDN)
  - Fan Out Ingress rule (based on path)

Ingress Controller

- All of the magic is done using the Ingress Controller
- Once the Ingress Controller is deployed, we can create an ingress resource using the kubectl create command
- An Ingress Controller is an **application** which:
  - Watches the Master Node's API Server for changes in the Ingress resources
  - Updates the Layer 7 load balancer accordingly
- Kubernetes has different Ingress Controllers, and, if needed, we can also build our own
- GKE L7 Load Balancer and Nginx Ingress Controller are examples of Ingress Controllers

6.8.15 K8S Object - Scheduler

- Control plane process
- Works with controller-manager through the API-server
- Run in master node(s)
- Run in kube-system namespace
- Assigns Pods to Nodes
- Uses constraints and available resources
- Algorithms can be extended/customised

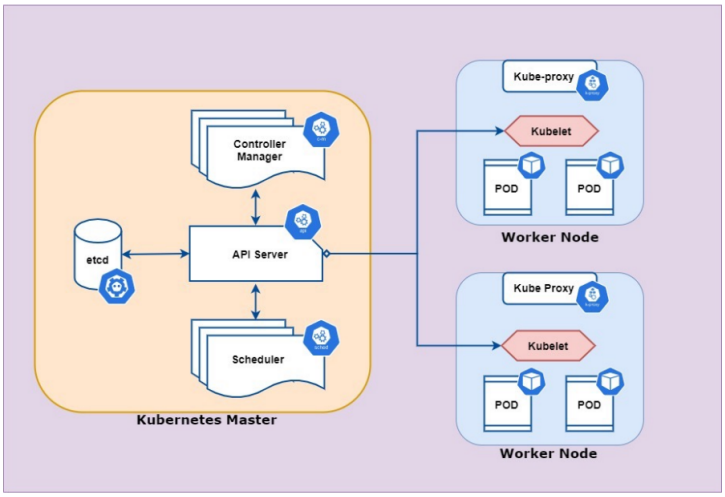


Figure 6.11: K8s Scheduler

Pod Placement

- The kube-scheduler selects a node for the pod in a 2-step operation:
  - Filtering
  - Scoring
- Filtering uses some scheduler constraints
  - NodeSelector
  - Affinity and anti-Affinity

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: lb-example2
    spec:
      nodeSelector:
        region: "EDGE"
      containers:
      - name: worker
        image: python:3.6-alpine
        command:
        - "/bin/sh"
```

- Scoring sorts the remaining nodes to choose the most suitable Pod placement, based on active scoring rules