Introduction to Storage & Cloud Management: OpenStack

Gabriel Scalosub

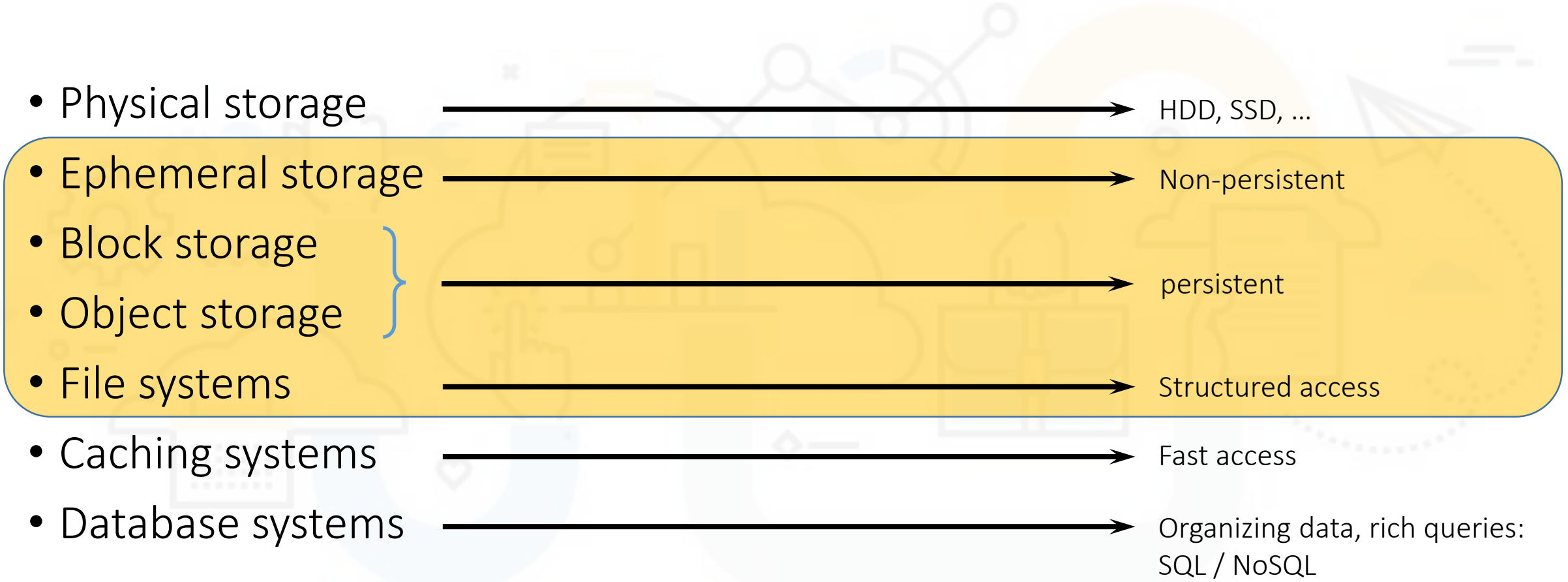
Borrowed extensively from:

Erez Biton, Kobi Ginon, openstack.org, Sander van Vugt, David Mahler, Christian Schwede, Sam Merritt, Roy Cambell and various other papers/resources (see list at the end)

Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

Introduction to Storing Data in the Cloud



Ephemeral Storage

- Ephemeral = short-lasting (i.e., non-persistent)
- Examples
  - General computing: anything in RAM that is not saved to disk (volatile memory)
  - Containers: any local changes/files used/saved in a Docker container
    - Local volume (disk) content erased after container termination
  - Ubuntu Live CD
- VMs
  - Usually storage on the host running the VM
  - Not part of the “standard” cloud storage infrastructure

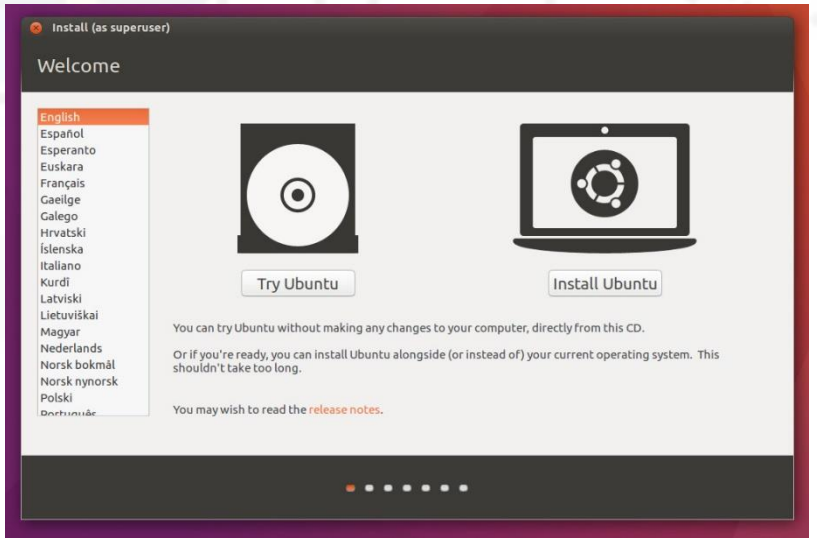


Image: ubuntu.com

Block Storage

- Data / files are split into equal size blocks
  - Block size < a few KB (e.g., 512B)
  - Just address, no metadata
- Main use cases:
  - High I/O rate, many random read/write accesses
    - Databases
    - Email servers
    - Virtual Hard Disk (vHD)
- Volume:
  - Collection of blocks aggregated into a single logical entity
  - Accessed by OS as a mounted drive
    - Directly Attached Storage (DAS) on the host, or over the network

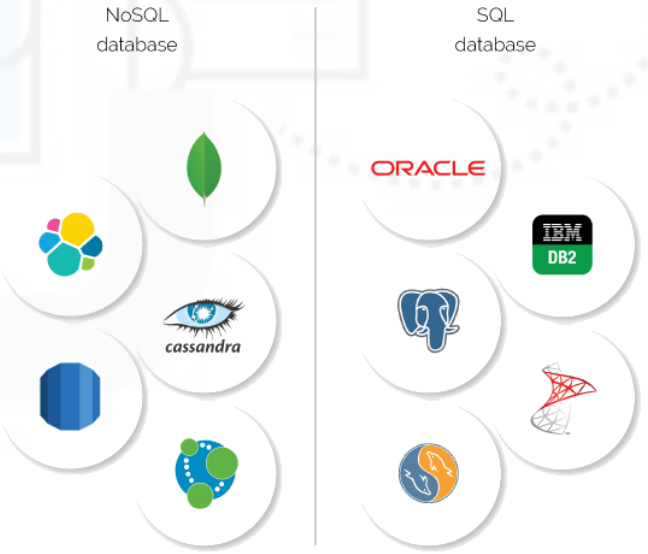


Image: alooma.com

Block Storage

- Block access
  - Storage protocols
    - iSCSI/TCP, Fiber Channel (FC), FCoE, Infiniband
- Storage Area Network (SAN)
  - Distributed storage system over a dedicated network
  - Handles replication across storage nodes, provides high-availability, load balancing
  - Fast & Expensive…
- VMs:
  - Block volumes available as vHDs
  - Quick migration: detach and reattach volume
    - Including boot

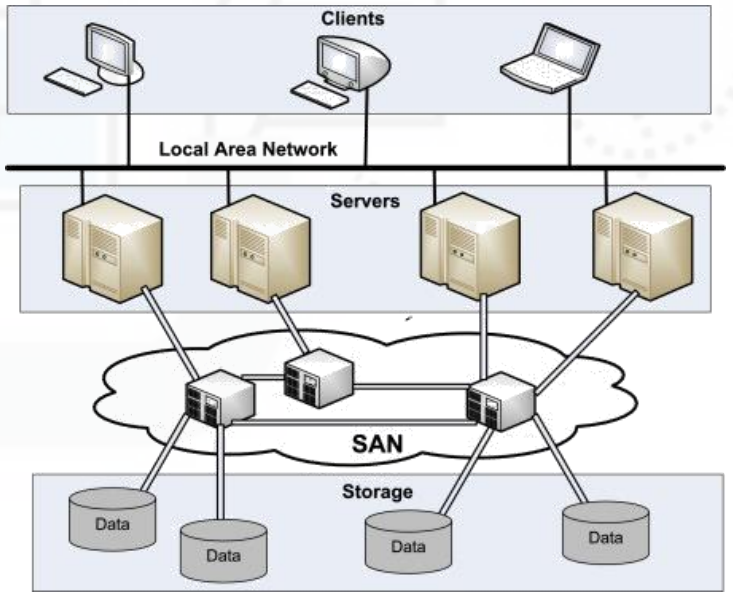


Image: Marinescu (2018)

Block Storage

- RAID (Redundant-Array of Inexpensive Disks)
  - Array of multiple, concurrently-reachable, physical disks
  - RAID X: varying encoding/data distribution, with properties
    - Examples:
      - RAID 0: spread out each data item (files) across disks
        - No fault tolerance (single bad disk destroys all data), fast concurrent access
      - RAID 1: Mirroring all content on more than 1 disk
        - Fault tolerance, somewhat improved latency (reply from fastest disk)
    - Cost vs. performance (access latency) vs. fault tolerance

Image: dell.com

- Examples of block storage in the cloud
  - Amazon EBS, Azure (Managed) Disks, Google Persistent Disk

Object Storage

- Used for storing heterogeneous data

Object data examples			
Media files	Large datasets	Disk images	Container images
Data backups	Log/dump files	Archives	Static web pages

- Object components:
  - Data
  - Metadata
    - Information about the data (description, creation, permissions, etc.)
  - Unique ID
    - For recovery in large distributed systems (e.g., URL)
- Killer application: Big Data
  - No structure/hierarchy is imposed
  - No data type restriction/enforcement

Object Storage

- “Flat“ storage
  - Data objects are stored in ***containers*** (or “buckets”)
    - NOT Docker containers…
  - No metadata management (indexing, search) out-of-the-box
    - E.g., maintaining file system structure, mapping, etc.
    - If required, should be managed by user / third-party application
  - Unbounded extension:
    - Just add more storage containers / nodes

- Object handling: single complete unit
  - Cannot change/update “part” of an object
    - Must read entire object, change/update object, entirely re-written to storage
  - Must have key to retrieve object
  - Supports simultaneous read-access

Object Storage

- Eventually consistent
  - Replication is performed for fault-tolerance
  - It takes time for up-to-date copy to be available everywhere
  - Not designed for frequently changing/updated data
    - E.g., not for DBs
- Intuition: “key-value store”
  - Access data by key
  - Value could be anything
  - E.g., python dictionaries, JSON, …

JS *Object* Notation

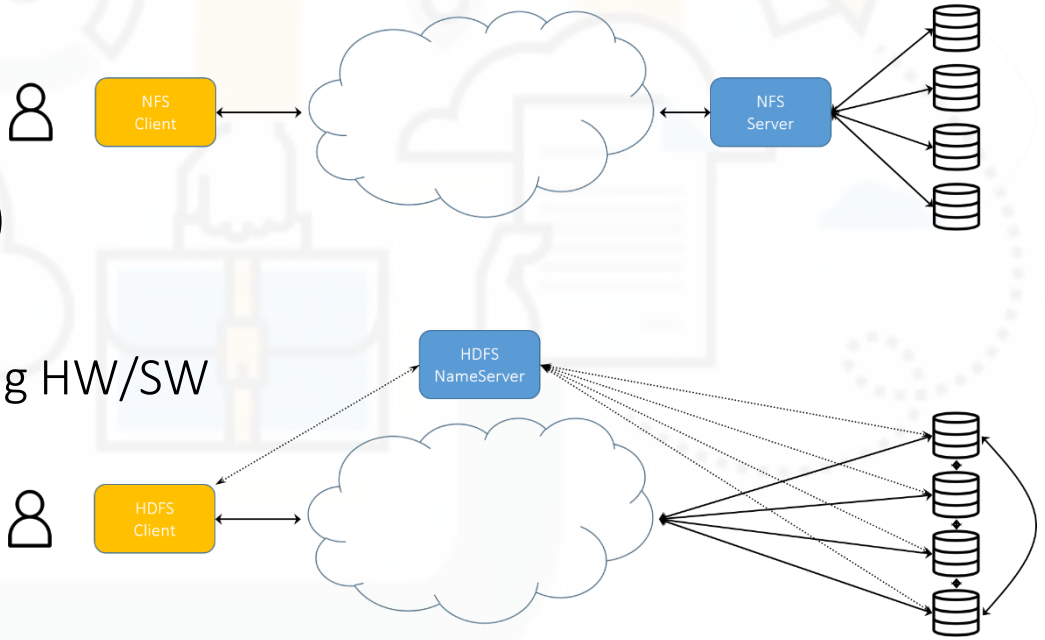
- Examples of object storage in the cloud
  - Amazon S3 (Simple Storage Service), MS Azure Blob, Google Cloud Storage

amazon S3

Microsoft Azure Blob Storage

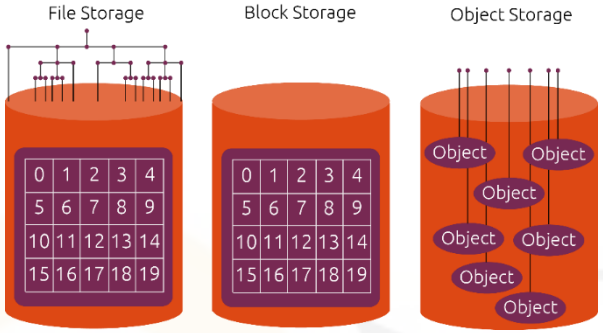
File Systems

- Logical hierarchy on top of blocks
  - Block addresses <-> directories/files
- Network-attached Storage (NAS)
  - FS protocol (on top of some storage protocol)
  - Runs on standard network infrastructure
    - No need for specialized/expensive fiber-supporting HW/SW
  - Accesses block storage using volumes + FS
- Cloud File Systems:
  - Remote file systems: Client-Server
    - E.g., Network File System (NFS), Server Message Block (SMB / Samba)
  - Distributed File Systems: Cluster-based
    - E.g, Hadoop Distributed File System (HDFS), Google FS (GFS)



more on this later in the course

Block vs. Object vs. File



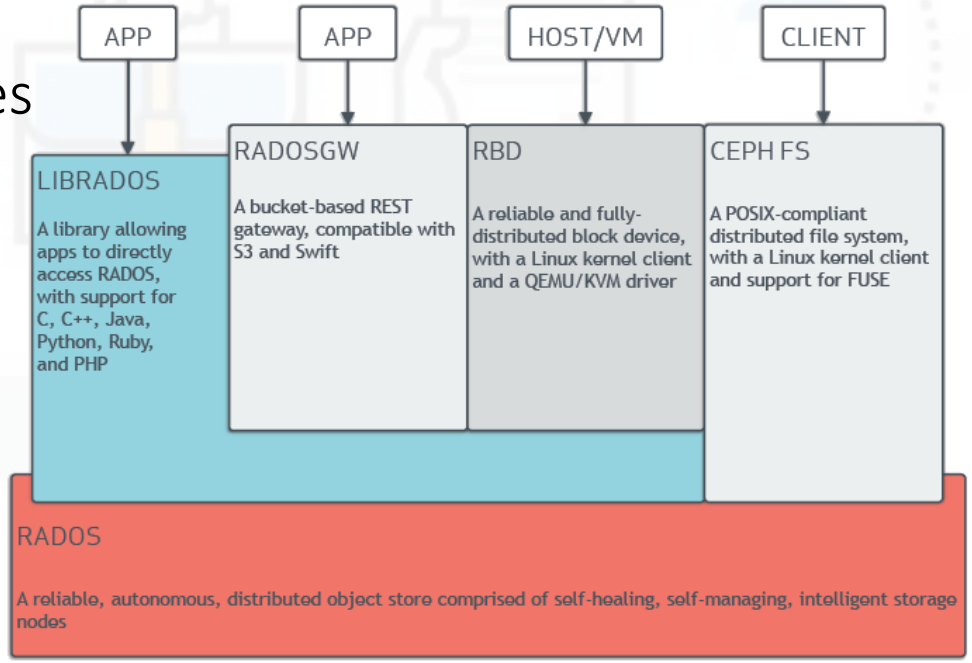
- Roughly…

	Block	File	Object
Interface	Operating System	User	Program (API)
Cost	$$-$$$	$$ - $$$$	$
Performance			
Proximity	Dedicated Network Fibre Channel / 10Gb	LAN / 10Gb	Internet
Use Case	OS, Database	Sharing user data, web content	Images, PDFs, Video
Scale			

Image: thecustomizewindows.com

(A Word on) Ceph

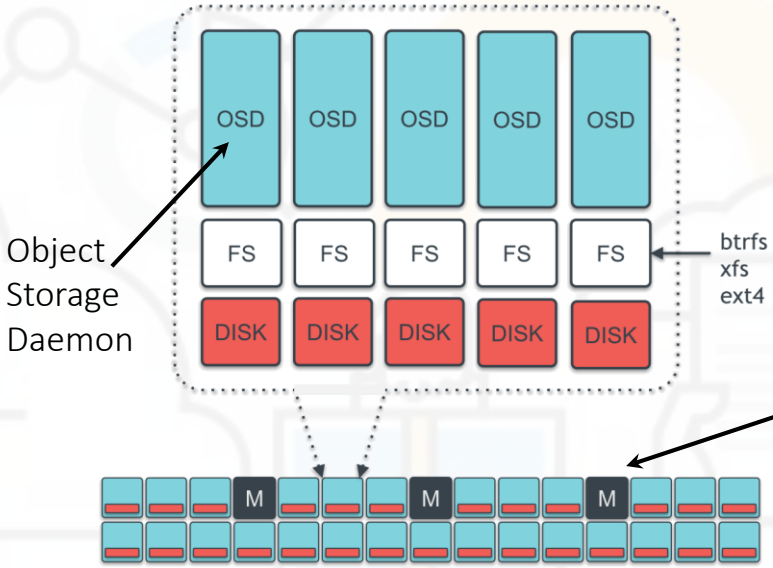
- Probably the most widely used distributed storage system
  - Supports Amazon S3, OpenStack Swift (coming up), etc.
  - Open source
- Architecture
  - Based on RADOS cluster of (object) storage nodes
- Abstractions
  - Object, block, file system
- SDS: Software-defined Storage
  - Self-healing, self-managing, high-availability (coming up)
  - “infinitely” scalable
    - 1000s nodes and more

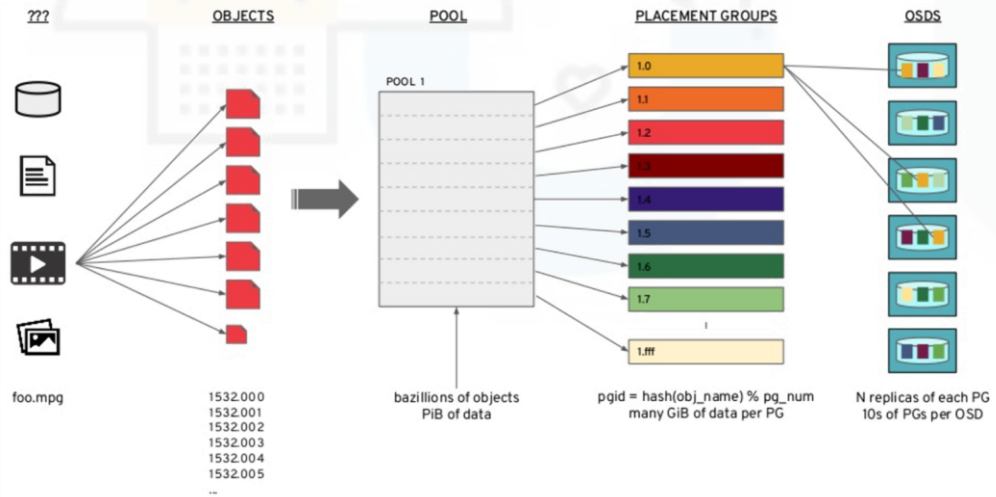


Source: ceph.com

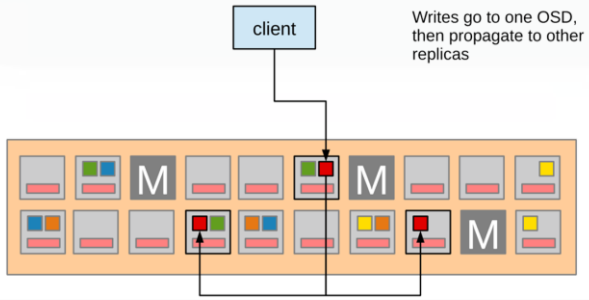
(A Word on) Ceph

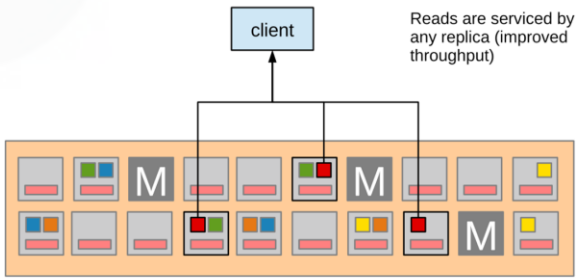
- Architecture (in a bit more depth)
  - Cluster of object storage nodes
  - Efficient read/write
    - Redundancy in the background…
  - Placement and redundancy: CRUSH
    - Replication/erasure coding





Controlled Replication under Scalable Hashing





Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

Orchestration Tasks (Recap*)

- Provisioning and Deployment
- Configuration
- Fault tolerance and healing
- Scaling
- Load Balancing
- Monitoring
- Management

Automation

* Lecture 3

Orchestration: Infrastructure (Recap*)

- Resource provisioning
- Configuration management
  - Terraform, Ansible, puppet, chef, …
  - Describe desired state/action configuration
    - Using .yml, .json, or some specific description language
    - State/action:
      - package availability (e.g., ensure apache is installed and available)
      - service state (e.g., ensure sshd service is running, listening for incoming ssh requests)
      - execute commands (e.g., update via apt-get) …
- Performed on any subset of infrastructure
- Infrastructure-as-Code (IaC): SW-based management


```
case $operatingsystem {
  centos, redhat: { $service_name = 'ntpd' }
  debian, ubuntu: { $service_name = 'ntp' }
}

package { 'ntp':
  ensure => installed,
}

service { 'ntp':
  name      => $service_name,
  ensure    => running,
  enable    => true,
  subscribe => File['ntp.conf'],
}

file { 'ntp.conf':
  path    => '/etc/ntp.conf',
  ensure  => file,
  require => Package['ntp'],
  source  => "puppet:///modules/ntp/ntp.conf",
  # This source file would be located on the Puppet master at
  # /etc/puppetlabs/code/modules/ntp/files/ntp.conf
}
```

Example Puppet manifest. Source: Puppet

* Lecture 3

Orchestration: Containers (Recap*)

- CaaS: Container-as-a-Service
- Tasks
  - Deploy services
    - Container instances that may handle requests for service
    - Satisfy compute, storage, networking requirements
    - Replication requirement
  - Monitor and manage
    - Health, recovery, updates, scaling
  - Load balancing
    - Reverse-proxy, explicit load balancers

* Lecture 3

Management Orchestration: ~~Containers~~ VMs

moving to VMs…

Is it Clear?!

- ~~CaaS: Container-as-a-Service~~ IaaS: *Infrastructure-as-a-Service*
- Tasks
  - Deploy ~~services~~ VMs
    - ~~Container~~ instances that may ~~handle requests for service~~ run multiple applications on their own (arbitrary) OS
    - Satisfy compute, storage, networking requirements
    - Replication requirement
  - Monitor and manage
    - Health, recovery, updates, scaling
  - Load balancing
    - Reverse-proxy, explicit load balancers

we're now virtualizing the HW, not the OS

manage/configure the entire infrastructure

orchestration on the workload level

billing…

not just for applications, also for storage, network, etc.

also running containers…

- Heavy-lifting: Instances that
  - Run multiple, demanding applications on a VM
  - Exhaust most of the OS's resources/abilities
  - Use lots of storage, huge DBs, complex networks
  - Require extra focus on security

Management: VMs

- IaaS: Infrastructure-as-a-Service
- Tasks
  - Deploy VMs
    - Instances that may run multiple applications on their own (arbitrary) OS
    - Manage/configure entire infrastructure
      - Authentication, Hosts, Compute, storage, networking, security, …
    - Orchestrate workloads (replication, scaling, load balancing)
  - Monitor and manage
    - Billing, health, recovery

- Heavy-lifting: Instances that
  - Run multiple, demanding applications on a VM
  - Exhaust most of the OS's resources/abilities
  - Use lots of storage, huge DBs, complex networks
  - Require extra focus on security

Cloud Management

- Large, complex, SW suites, managing the IaaS
- Managed by a Cloud Service Orchestrator
  - Automates workflows
  - Allows customers to configure their own virtual infrastructure
- Commercial management suites (VMware, Microsoft, Cisco, …)
- Open source management suites
  - Open Nebula
  - Apache CloudStack
  - OpenStack

OpenStack: Introduction

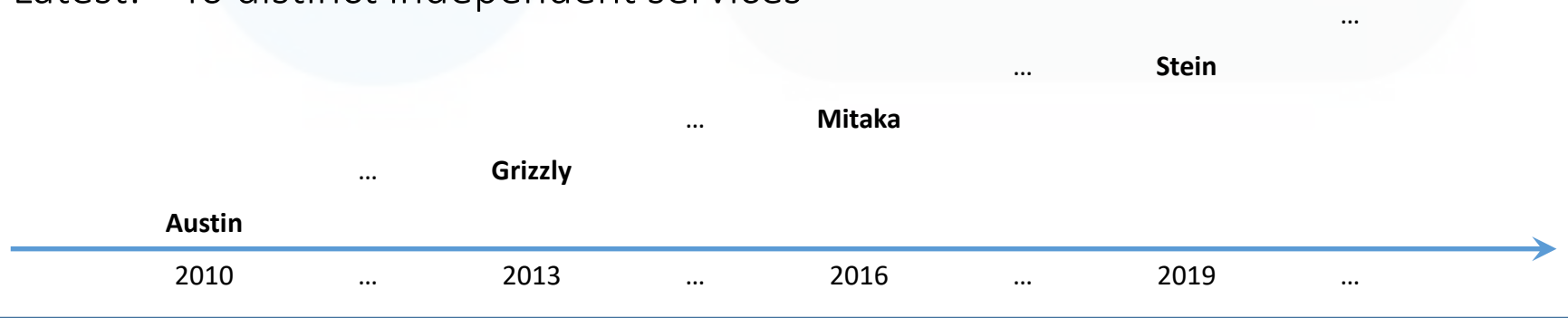
- Open-source project
  - Started by Rackspace and NASA (2010)
  - Written in Python
  - Since 2016 managed by the OpenStack Foundation
- De-facto “standard” for open source cloud management
  - Commercial providers
    - E.g., Amazon AWS, MS Azure, Google Cloud
    - (Probably) use their own proprietary cloud management suite
- Provides a RESTful API
  - AWS-style
    - Actually, a design criteria, for easy inter-operability

Image: openstack.org

OpenStack: Introduction

- Huge community
- Releases
  - 6 months cycle
  - A-B-C-…
    - Austin (2010), …, Ussuri (2020), …
  - Services “become independent”
    - Austin: just 2 services (Nova, Swift)
      - Others were within these two (or didn’t exist…)
    - Latest: ~40 distinct independent services

Image: openstack.org



OpenStack Deployment

- On-Prem
  - Customer owns infrastructure + OpenStack deployment to manage its workload
- Public Cloud
  - Provider manages its cloud using OpenStack
  - Customer merely requests VM deployments (images/HW flavors, etc.)
- Private Cloud (IaaS)
  - Bare metal infrastructure on provider cloud
  - Customer runs its own OpenStack

Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

OpenStack Architecture

- ~~Simple Architecture~~: Compute, Networking, Storage (and Dashboard)

don’t believe
everything you hear….

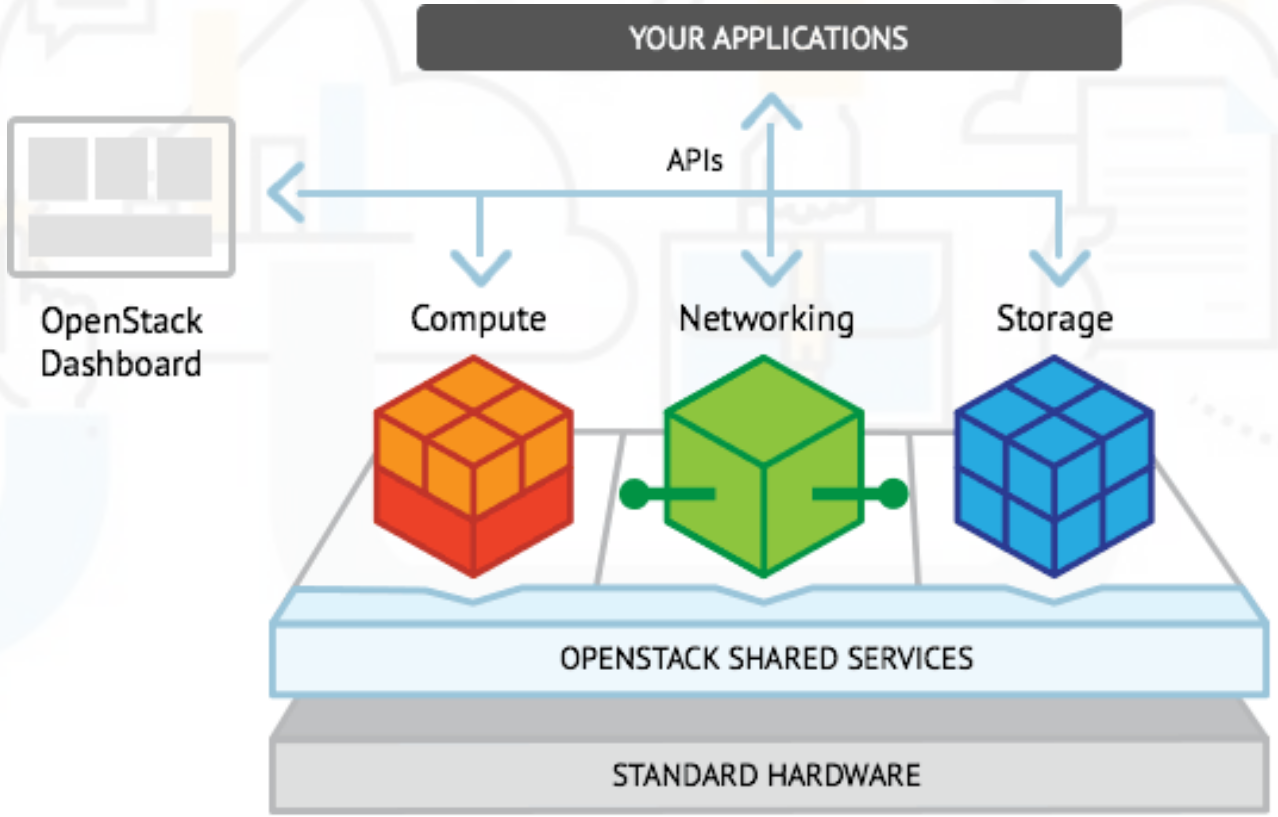


Image: mirantis.com

OpenStack Components

- OpenStack consists of various ***services*** (“anything BUT simple”…)

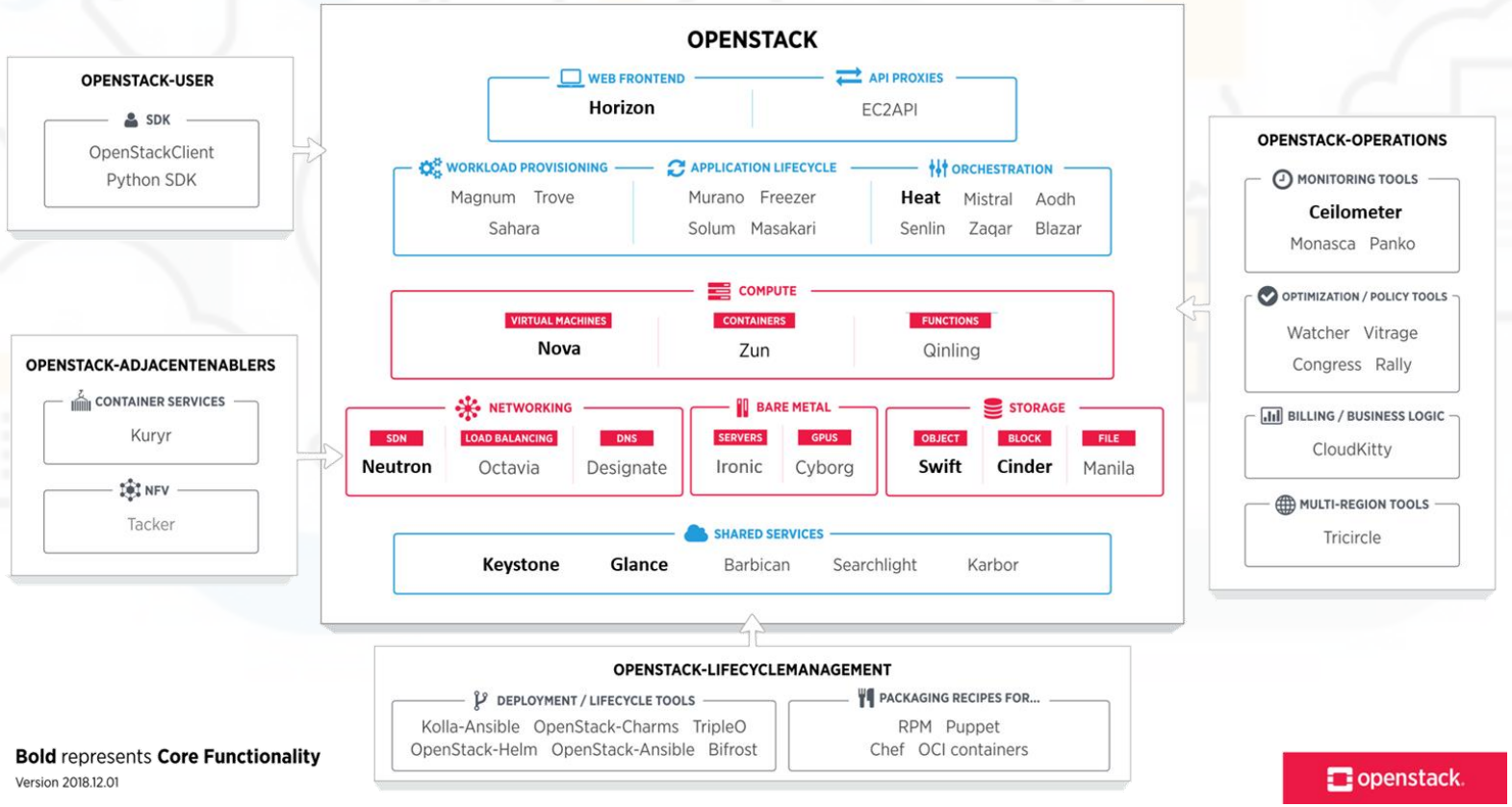
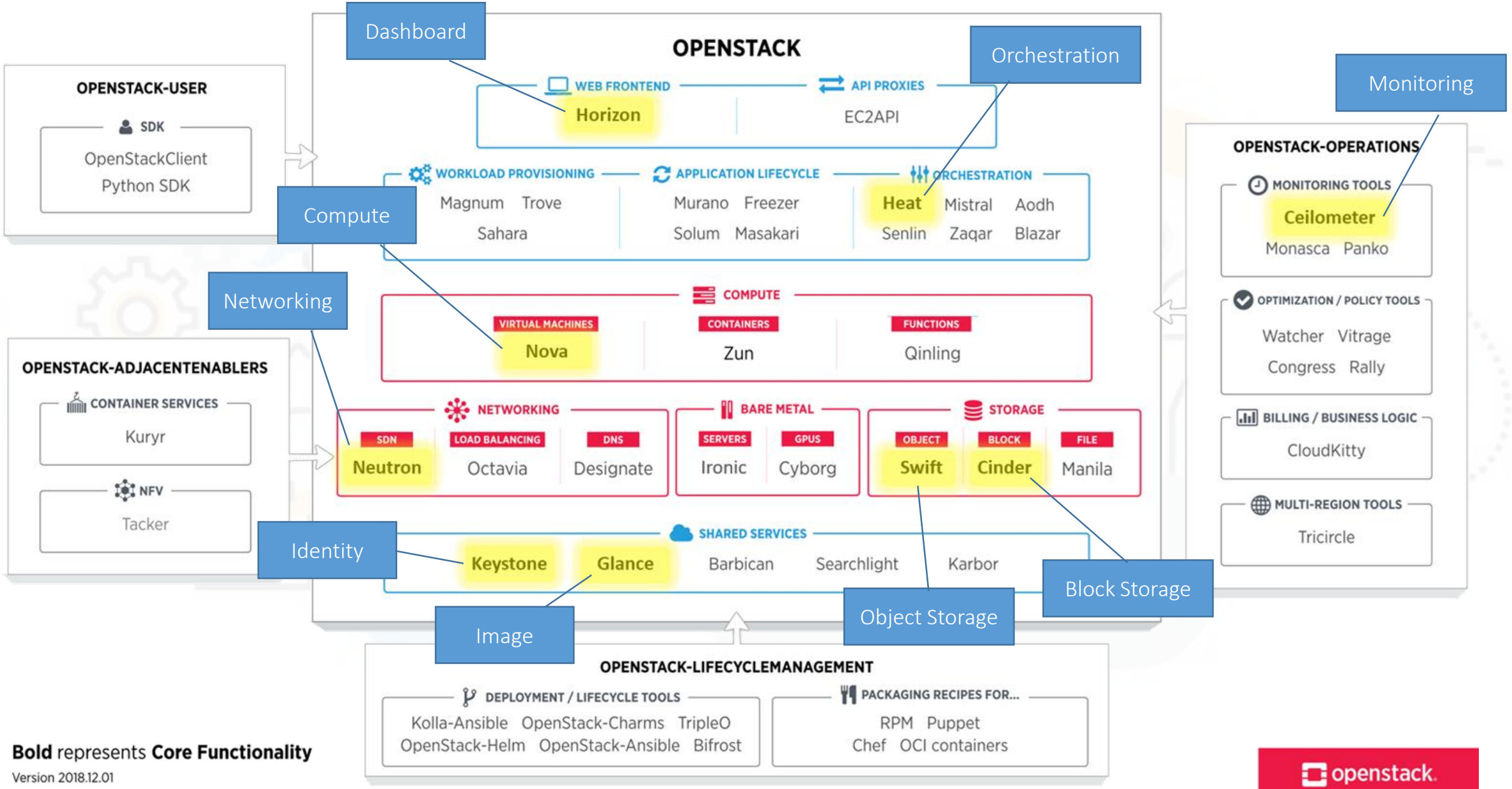


Image: openstack.org



Bold represents **Core Functionality**

Version 2018.12.01

openstack.

OpenStack Terminology

- Instance ≡ VM
  - Generated from image / VM snapshot
- Flavor: Instance requirements
  - #cores, memory, storage
  - “size” of VM to be deployed (think Bin Packing…)
    - Can be arbitrarily defined by user
- Tenant (Keystone) ≡ Project (Horizon)
  - Group of entities: users, images, network(s), volumes, …
  - Encompasses all OpenStack components corresponding to a “customer”
- Role: set of permissions
- User: administrative entity (assigned a specific role)

.conf	vCPU	RAM	Local Disk	Management Interface
.tiny	1	512 MB	1 GB	1 Gbps
.small	1	2 GB	20 GB	1 Gbps
.medium	2	4 GB	40 GB	1 Gbps
.large	4	8 GB	80 GB	1 Gbps
.2xlarge	8	16 GB	160 GB	1 Gbps
.4xlarge	16	32 GB	320 GB	1 Gbps
.8xlarge	32	64 GB	640 GB	1 Gbps

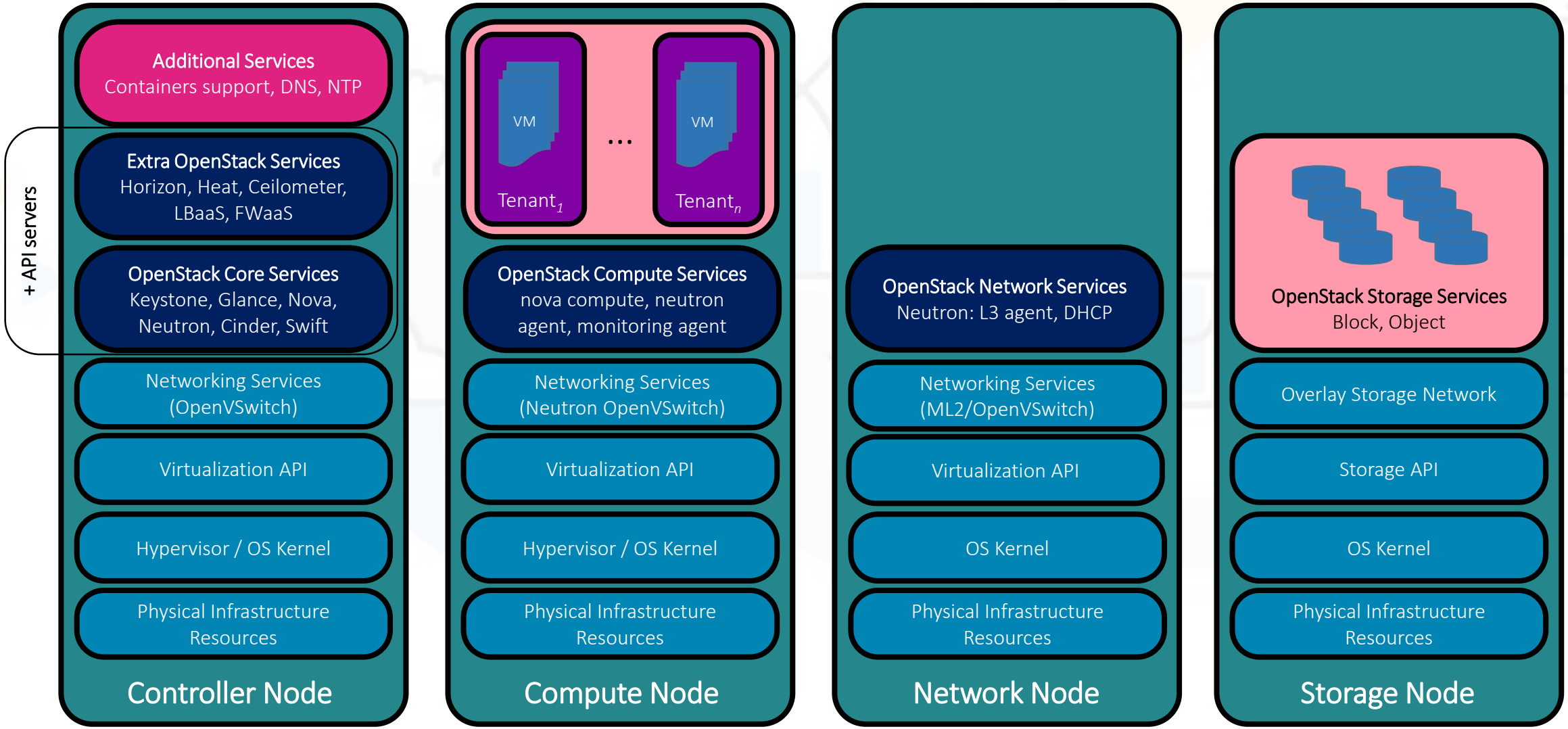
Openstack default flavors (github.com/cntt-n/CNTT/)

OpenStack Terminology

- Compute Node ≡ “hypervisor”
  - Machine running the Nova compute service
    - Can run instances
- Volume (block storage)
  - Persistent disk attached to single instance
    - Can be detached, snapshots can be created
  - Required for data access post instance-termination
    - data produced/changed by instance
- Ephemeral disk
  - Temporary disk used by an instance
    - Erased & unmounted upon instance termination

OpenStack: Types of Nodes

Image: based on github.com/cntt-n/CNTT/



OpenStack Architecture

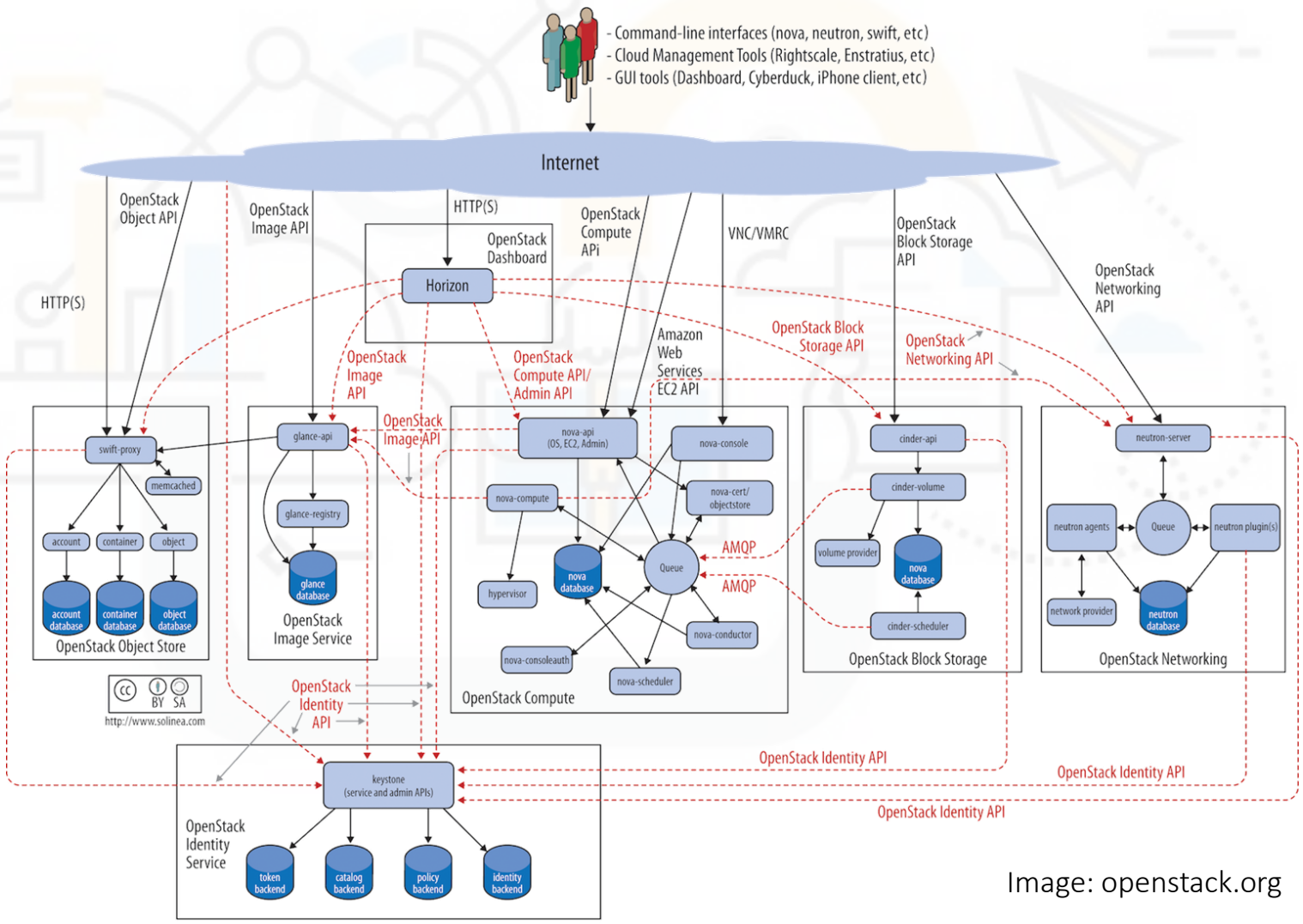
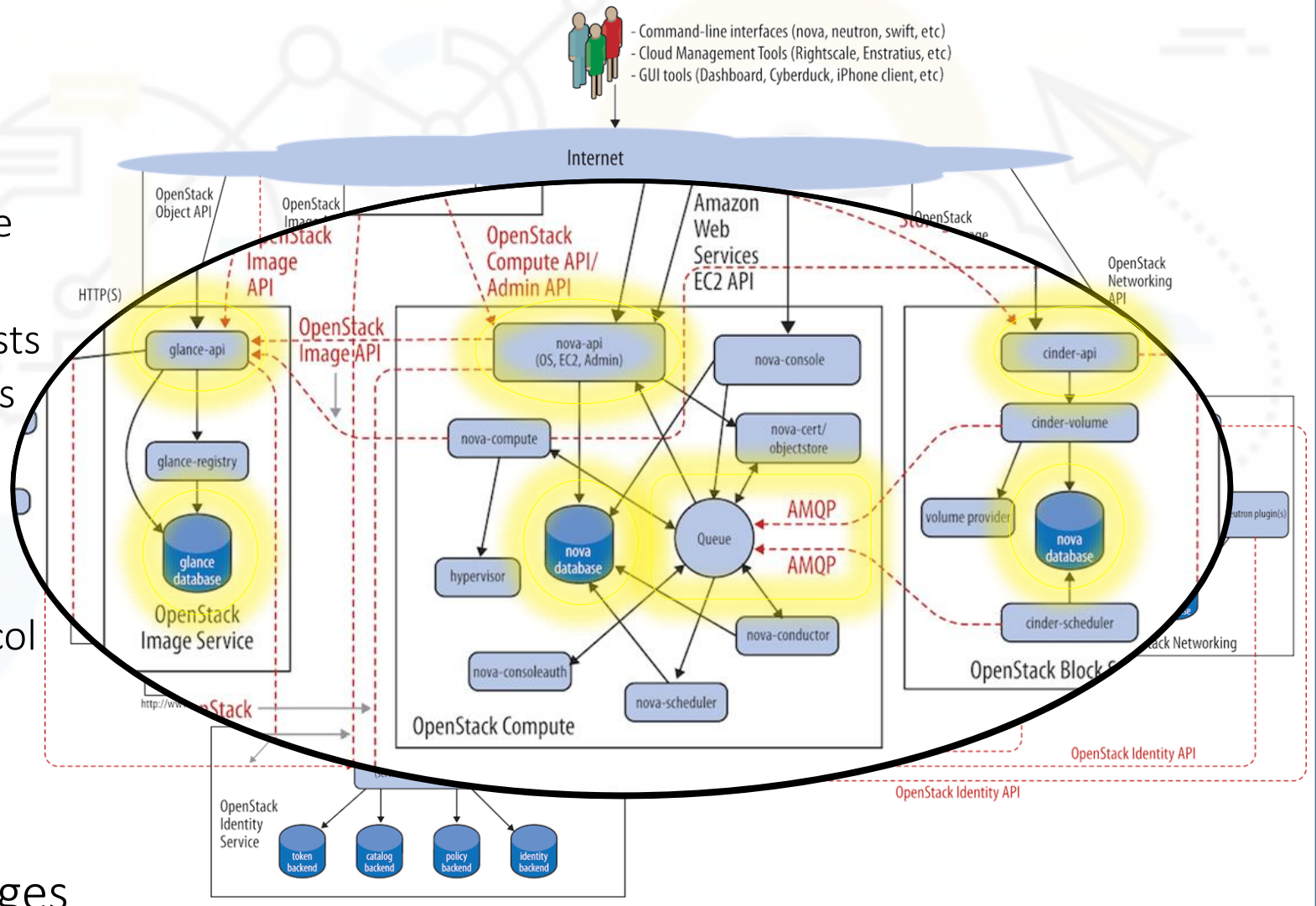


Image: openstack.org

OpenStack Architecture

- Each service has:
  - DB
    - maintains the service state
  - API server
    - REST API: user/SDK requests
    - RPC: inter-service requests
- Components interaction:
  - Exchanging messages
    - AMQP: Advanced Message Queueing Protocol
- Internally:
  - Message queues
    - Usually RabbitMQ
  - Access DBs, volumes/images

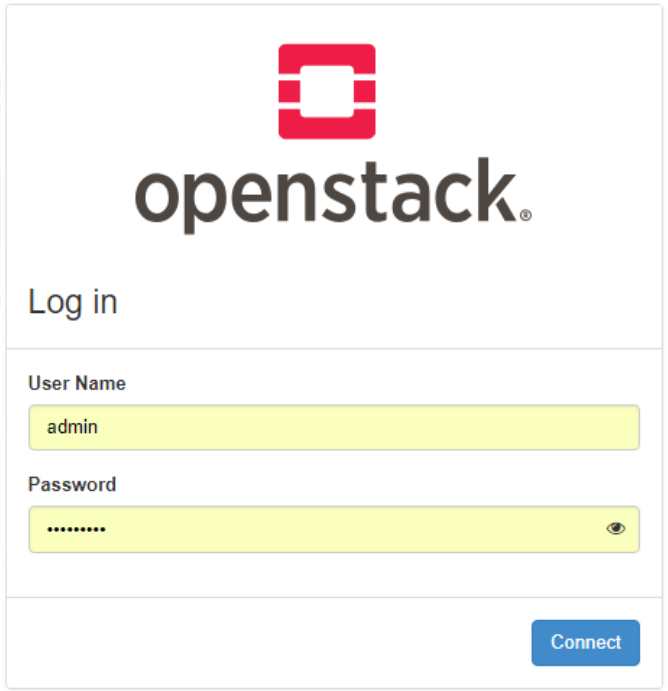


Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

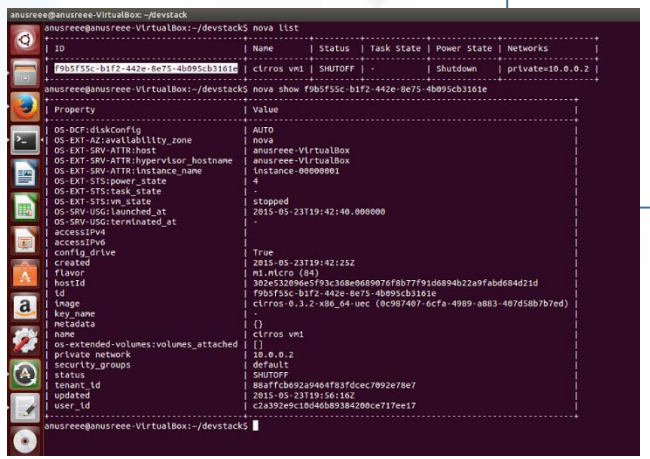
OpenStack Services

- OpenStack Keystone
  - Identity service
    - Authentication
      - Username/password
      - Returns `auth-token`
    - Manage permissions to resources based on roles
      - Resources: Volumes, VMs, networks, IP addresses, …
      - Mapping : Users -> Roles
        - Recall: role ≡ defines a combination of permissions
    - Using `auth-token`:
      - Sent as part of every request to other services made by the user
      - Sent by service to Keystone for authentication of every request



OpenStack Services

- OpenStack Horizon
  - Dashboard
    - GUI for managing OpenStack
  - Examples of management through Horizon:
    - Keystone: add users, tenants, roles, …
    - Neutron: define networks, configure overlays, …
    - Glance: add images, …
    - Nova: deploy instances, …
    - …
- Alternatives:
  - CLI: use REST API
    - On controller node



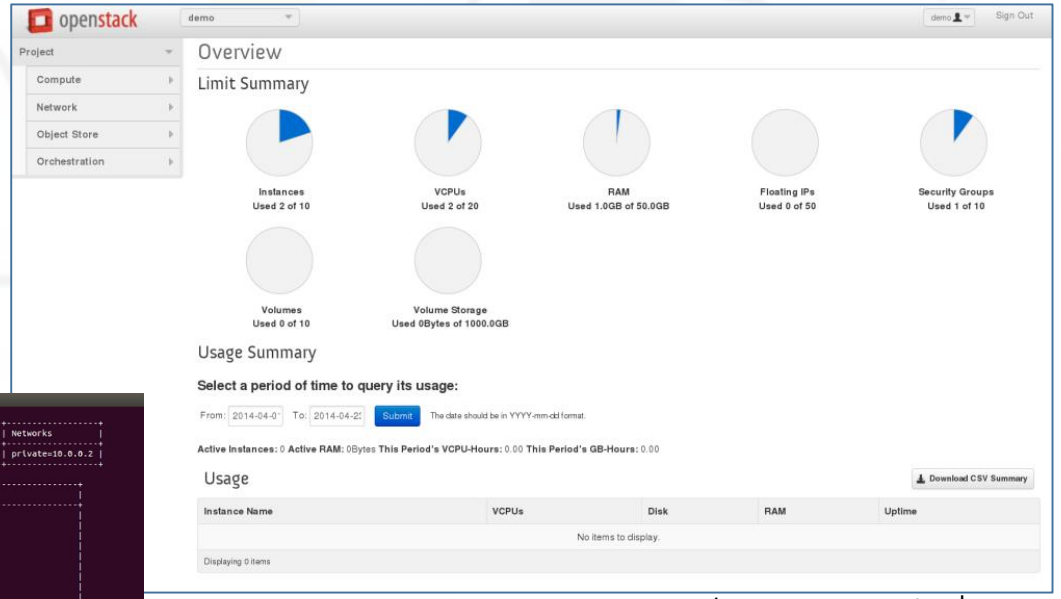


Image: openstack.org

OpenStack Services

- OpenStack Horizon
  - Dashboard
    - GUI for managi
  - Examples of ma
    - Keystone: add
    - Neutron: define
    - Glance: add imag
    - Nova: deploy insta
    - …
- Alternatives:
  - CLI: use REST API
    - On controller node

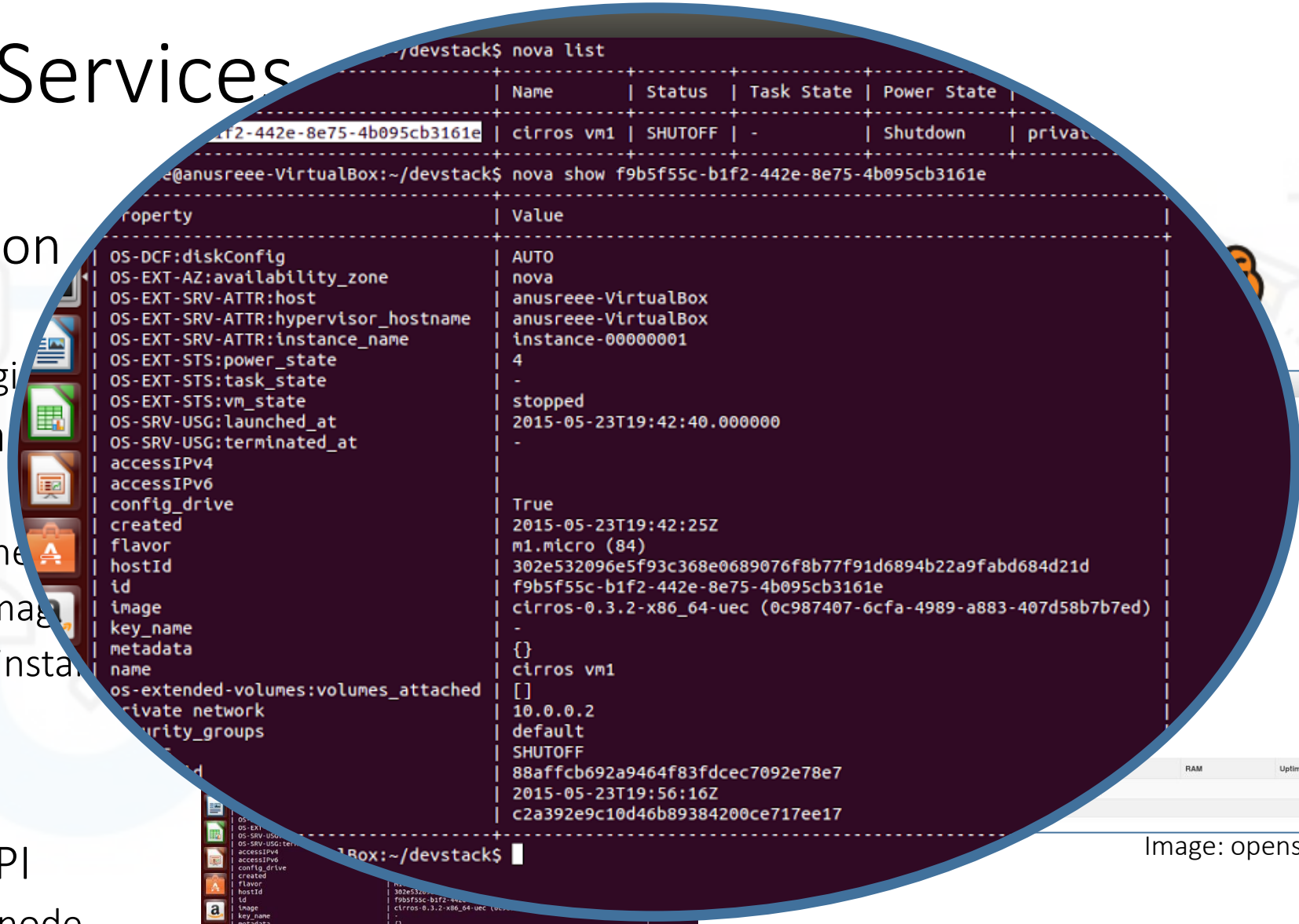


Image: openstack.org

OpenStack Services

- OpenStack Glance
  - Image service
    - Image repository
    - Maintains metadata DB of all available images
    - Actual images actually stored elsewhere
      - E.g., remote filesystems, object stores, DBs
      - Accessed/retrieved by communicating with other components
        - E.g., OpenStack Cinder/Swift, Amazon S3, …

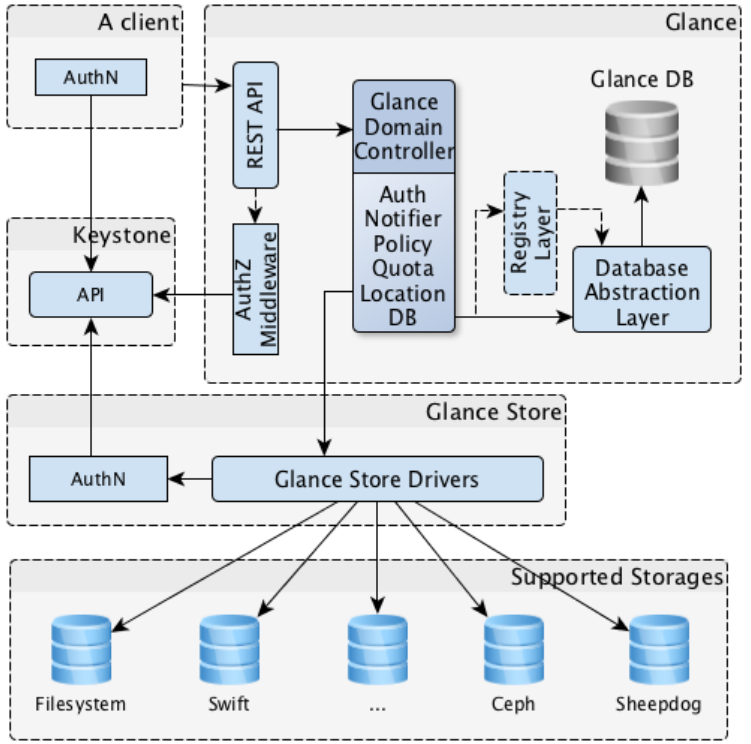
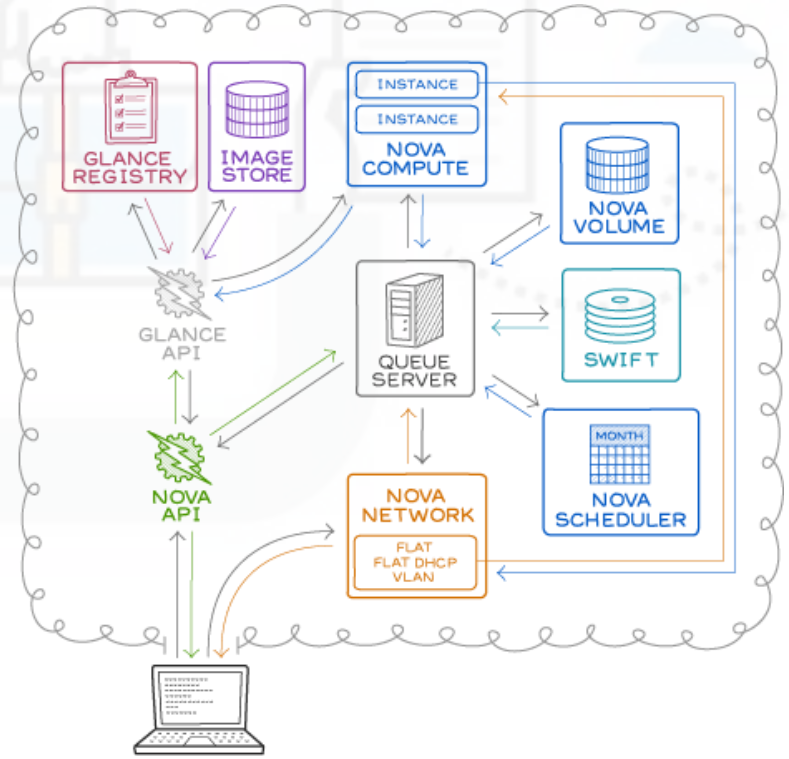


Image: openstack.org

OpenStack Services

- OpenStack Nova
  - Compute service
  - Provisioning VMs
    - `nova-compute`:
      - interface to the hypervisors running on compute nodes
      - Create/terminate VM instances via hypervisors' APIs
      - Download images using Glance
      - Interact with other Nova components
  - Distributed
    - Components can run on multiple servers
  - Interaction between components: Message queue
  - Networking provided by Neutron service
    - Agent running on the compute node
      - Managing OpenVSwitch on compute node



OpenStack Services

- OpenStack Nova
  - VM storage
    - Instances use ephemeral storage
      - On compute node, or
      - Remote compute storage
        - Makes it easier to migrate instances
    - Persistent/remote storage provided by Cinder
      - Block storage mounted as volumes
    - Object storage provided by Swift
  - Nova scheduler
    - Picks the compute node for each instance deployment request
      - Filter scheduler
    - Stay tuned…

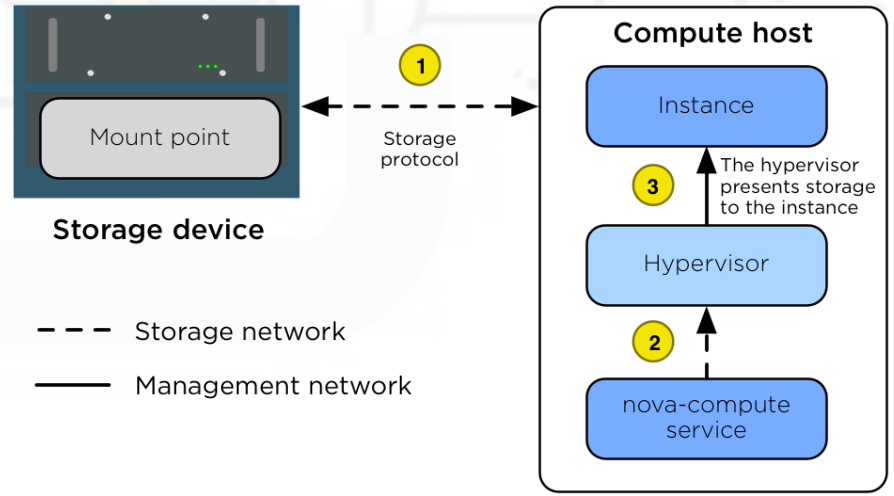


Image: openstack.org

OpenStack Services

- OpenStack Neutron
  - Networking service
    - SDN-controlled
  - Network separation:
    - Router: L3 subnets + NAT
    - VLAN, VXLAN, GRE, …
    - Inter-Tenant separation (and also intra-Tenant)
  - Security groups
    - Implements a firewall
  - Default plugin: ML2 (Modular Layer 2)
    - Manages network states (e.g., local, VLAN VXLAN, flat, etc.) via Type Drivers
    - Mechanism Drivers implement network types using OpenVSwitch, different vendors switch OSs, …

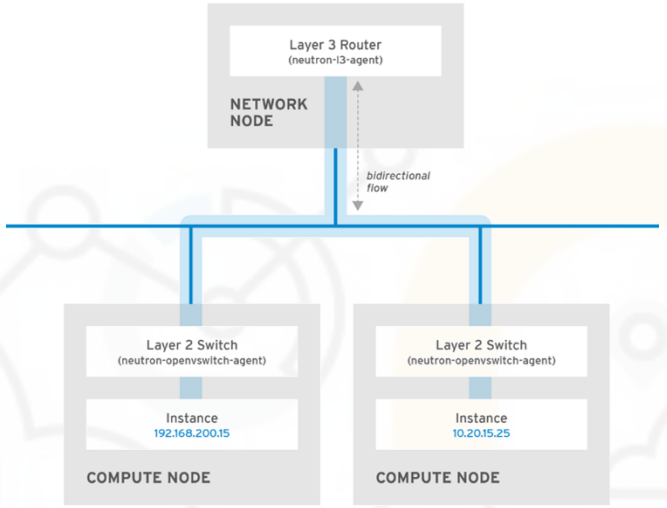


Image: dell.com

Generic Routing Encapsulation (resembles VXLAN)

all connected to single switch

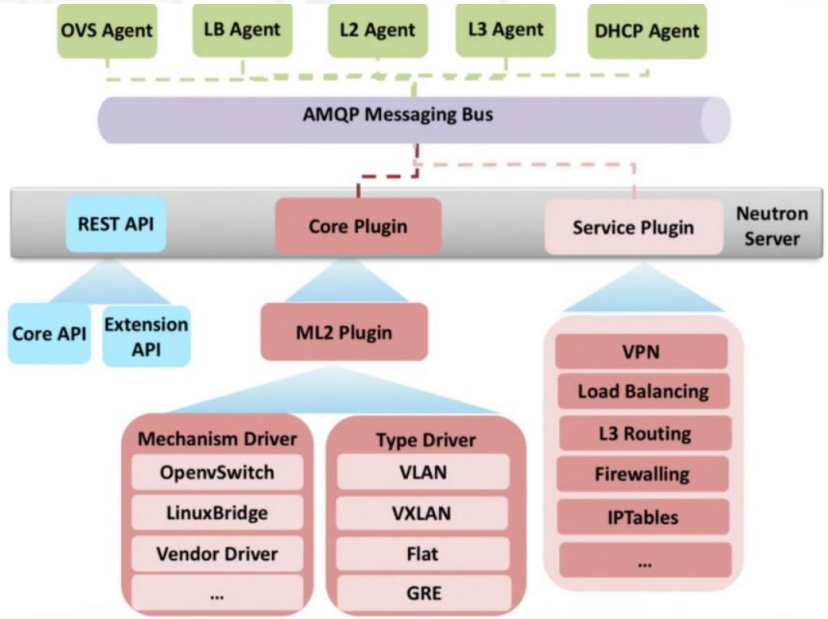


Image: Kedher (2018)

OpenStack Services

- OpenStack Neutron
  - Usually 4 types of (logical) networks (not including storage):
    - Management networks
      - Internal network between OpenStack components
    - Guest/tenant network
      - User-created network: Internal network providing connectivity between tenant instances
    - External network
      - Providing Internet connectivity to instances
    - API network
      - Provides Internet connectivity for OpenStack API calls
  - Provider network (usually physical/underlay)
    - General term denoting networks provisioned by OpenStack admin
      - Tenants “unaware”: can be used by multiple tenants

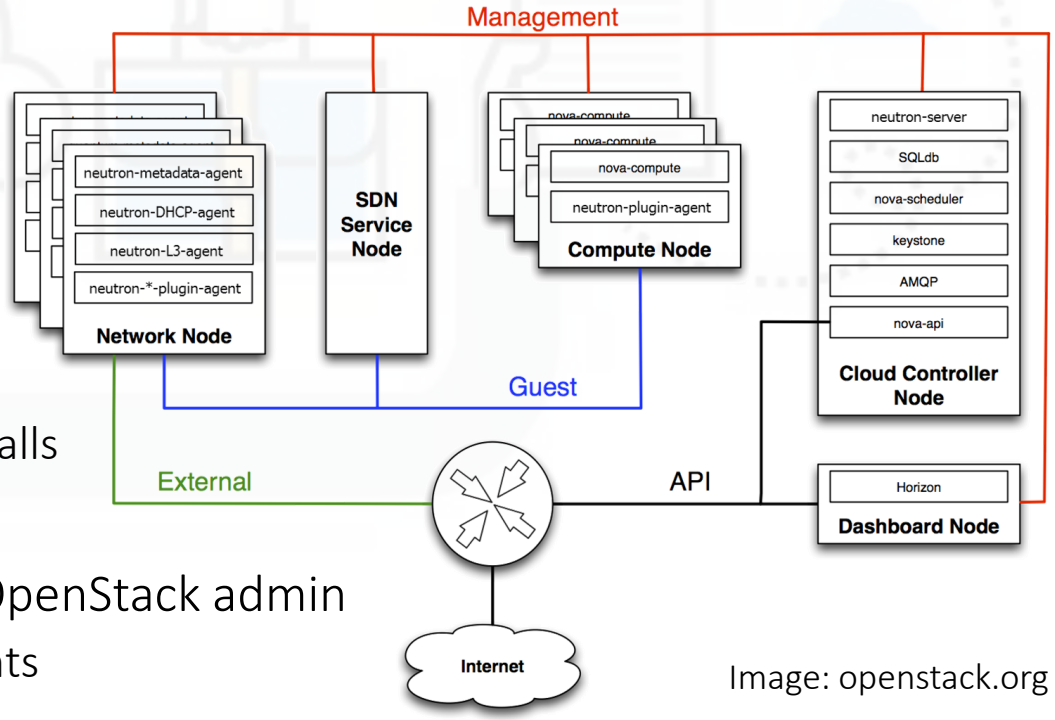


Image: openstack.org

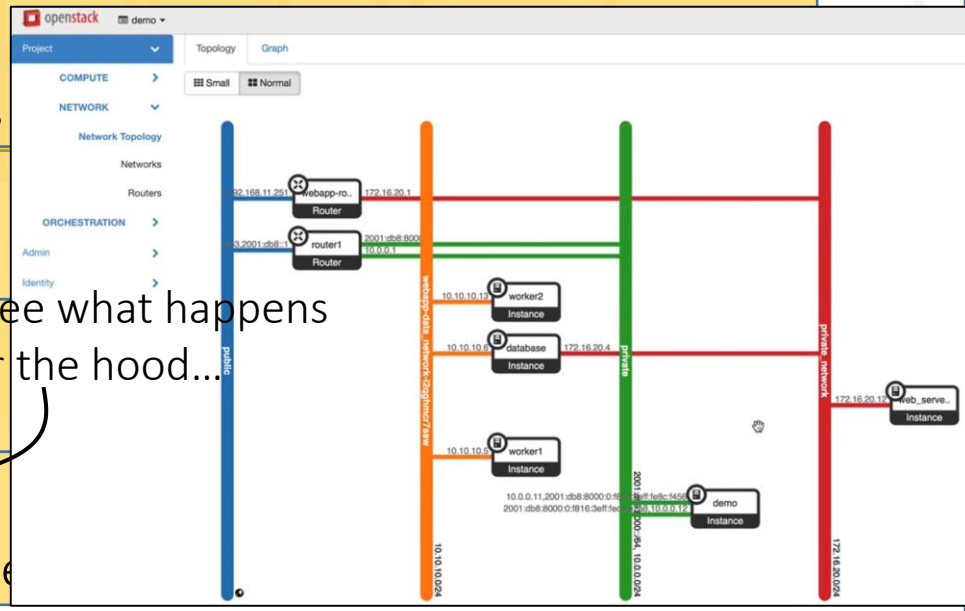
Interlude: Let’s See How It Works

Instance Pre-Deployment: User Perspective

- Minimal steps for deploying an instance within a project on Horizon
  - Configure project (customer) network
    - Internal network of project, router connecting to external network
    - Assign floating IP addresses
      - From public netwok IP address pool (each such IP would give external access to one instance)
    - Define a security group in the cloud
      - Essentially, defines firewall rules for accessing instances
  - Create an SSH key pair
    - Allows secure access to instance
  - Create a Glance image
    - Instances are not installed, they are ***deployed***!
  - Create Instance
    - Define Flavor: resources available for image (#vCPU, me

provided by admin

let's see what happens under the hood...



Instance Creation: Under the Hood

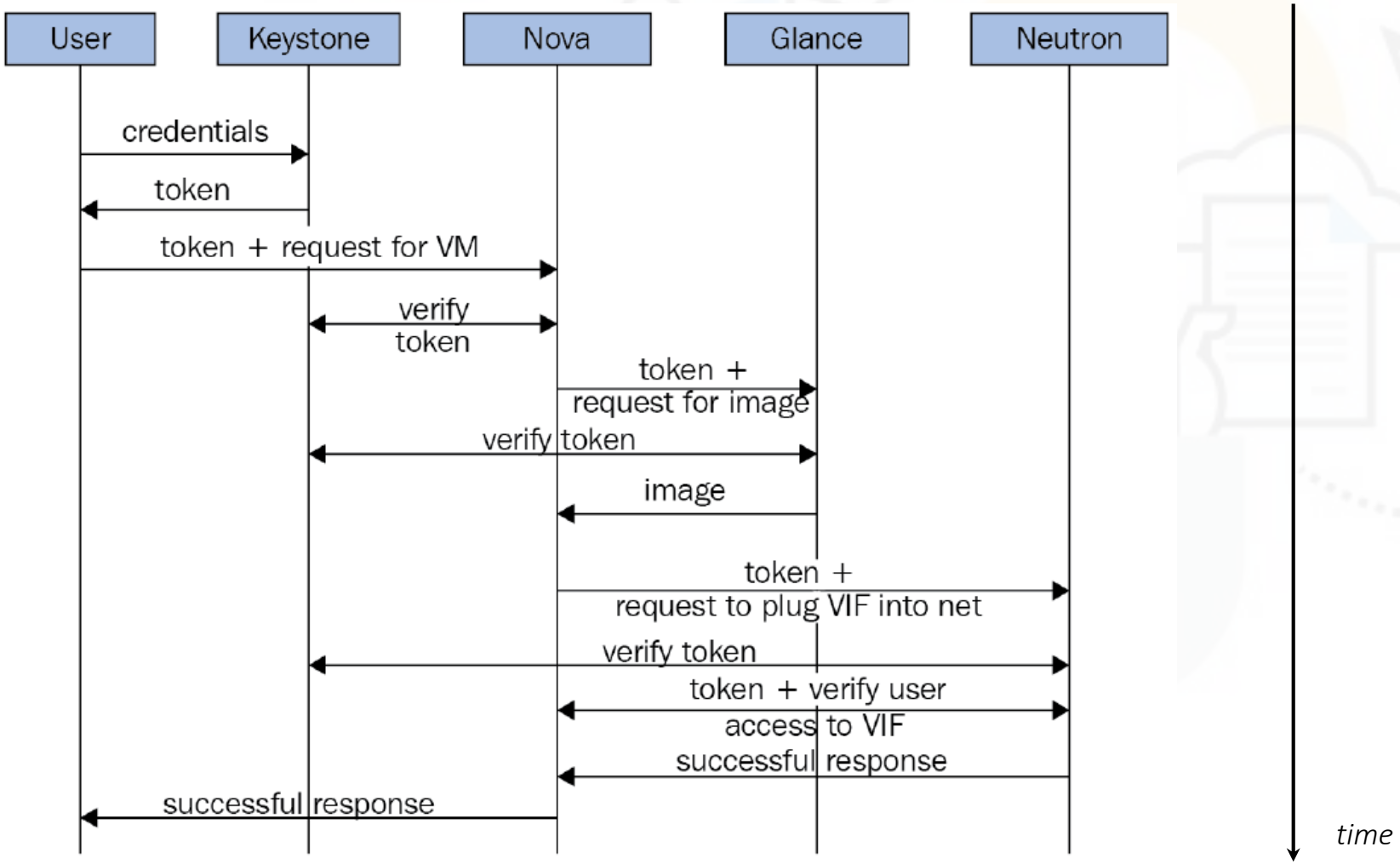


Image: Selvaraj (2017)

OpenStack Services

- OpenStack Cinder
  - Block storage service
    - Software-defined
  - Volume mounting for instance
    - Single instance!
    - Volume accessed via block storage protocol (e.g., iSCSI)
      - Over storage network
    - Cinder scheduler: chooses which volume to assign to each instance
  - Supports creating snapshots/backups from Volumes
    - Current instance volume -> snapshot / backup (object)
    - Snapshot / backup (object) -> new instance volume

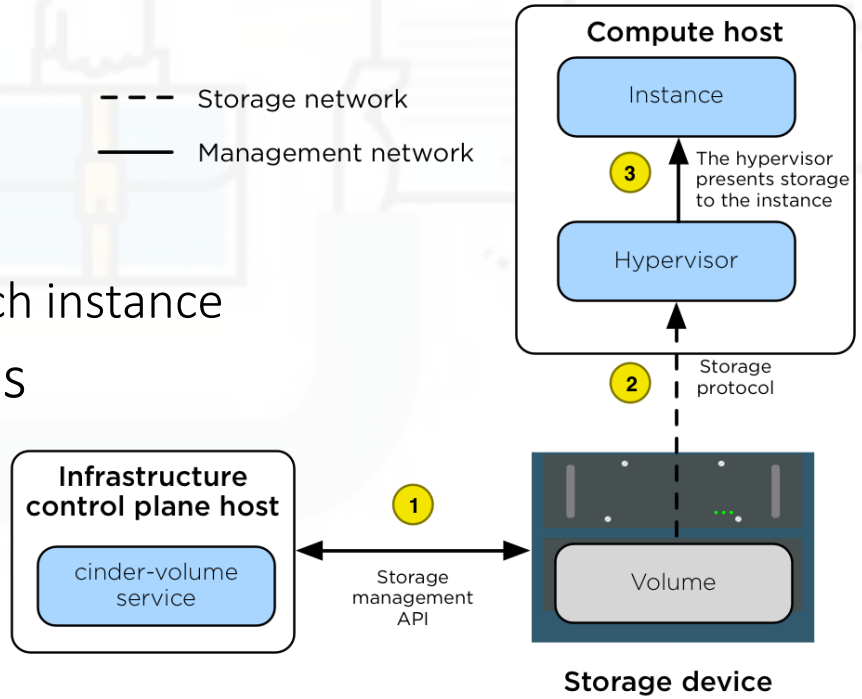


Image: openstack.org

OpenStack Services

- OpenStack Swift
  - Object storage service
  - Cluster-storage
    - Load balanced
      - Could have 100s of LBs in large clusters
    - Fault tolerance & high availability
      - Default replication of data: x3
      - Ensure replicas are well separated (disaffinity)
    - Background maintenance (independent of requests)
      - Restore corrupted objects, ensuring replication
  - Accessing an object
    - REST API (also for listing objects in account/container)
    - `http://myswift.com/v1/{account}/{container}/{object}`

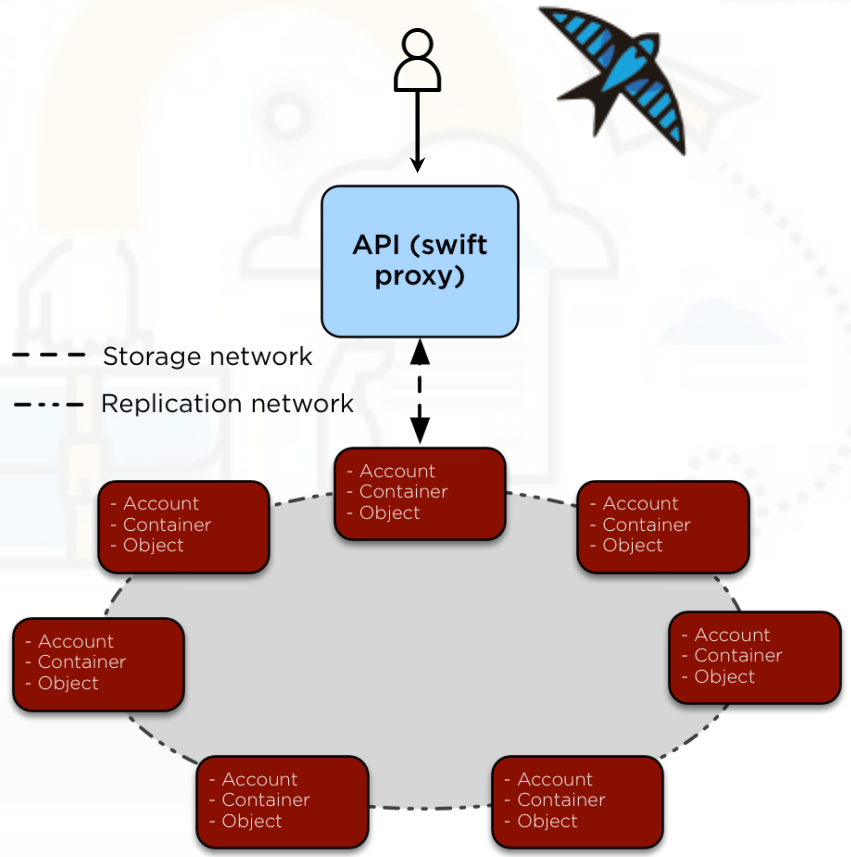


Image: openstack.org

Swift API version

user

user-generated container name

Object name

OpenStack Services

- OpenStack Swift
  - Swift storage policies
    - What type of storage should be used
      - Cold (e.g., tape) or hot (e.g., SSD)?
  - Swift Partitions and Rings
    - Determining which object is stored where
    - Consistent hashing:
      - hash(object_name) -> partition
      - partition -> physical device(s)
        - According to replication requirement
      - Managed by the cloud provider
    - Stay tuned…

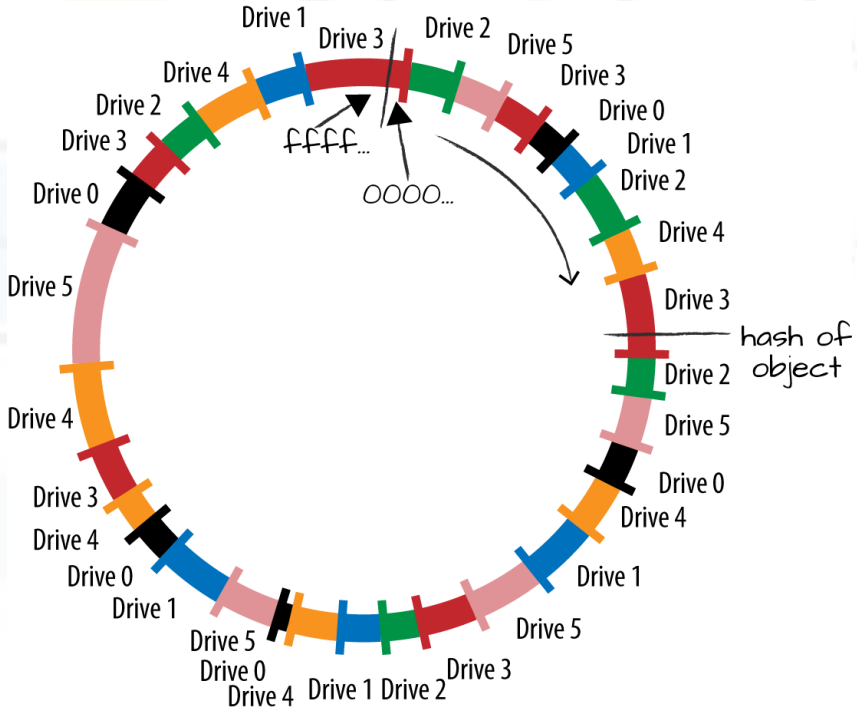


Image: avcourt.github.io/tiny-cluster/

OpenStack Services

- OpenStack Ceilometer
  - Telemetry service
    - Collects events and performance statistics from OpenStack services
      - Active (polling, Nova compute messages, …)
      - Passive (event listener on management network)
  - Example use cases:
    - ***Billing purposes***
    - Monitoring system health + notification
      - Failure, usage thresholds
    - Capacity planning
  - API-server
    - Handling queries

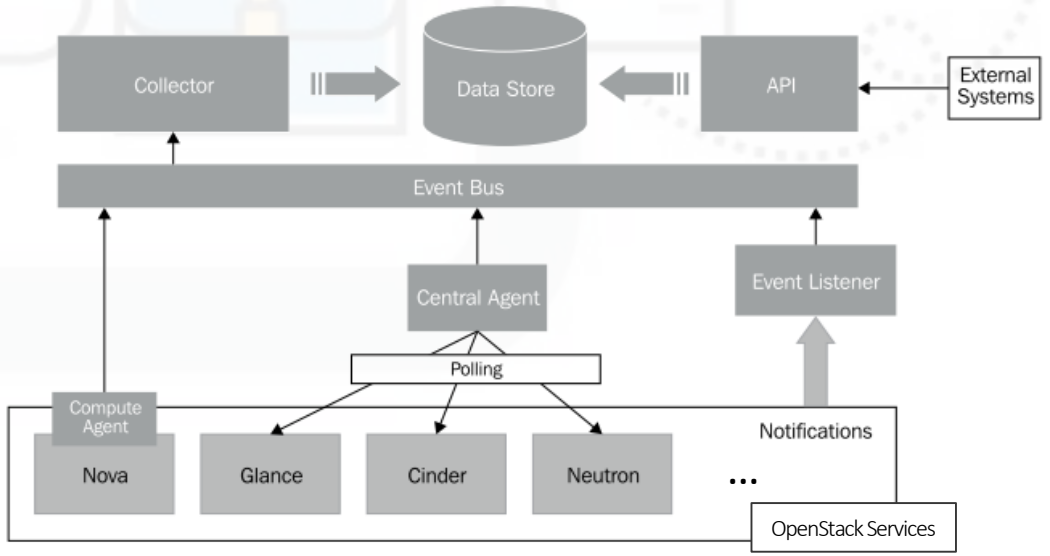


Image: Shrivastwa and Sarat (2015)

OpenStack Services

- OpenStack Heat
  - Orchestration service
  - Launching and managing complex applications
  - Automating the creation/deployment of OpenStack resources
    - Networks (subnets, routers, floating IPs, …)
    - Images, storage (volumes)
    - Instances
    - Security
    - etc.
  - Heat stack
    - Collection of resources orchestrated by Heat
    - Deployment for a specific tenant / admin

OpenStack Services

- OpenStack Heat
  - Heat Orchestration Template (HOT)
    - .yaml file specifying the required resources for a stack
    - Can have parameters to be filled out by values when stack is being executed
      - From environment .yaml file, and/or
      - Parameters provided in CLI
    - Integrates with Puppet/Chef
  - Can specify how to config/run EVERYTHING!
    - Networks, storage, user-password, …, docker containers running in instances, …
    - Default values, constraints on values (incl. regex)
    - Provides replication, scaling, etc.
      - Integrates with Ceilometer (telemetry)
    - Much much more…

```yaml
heat_template_version: 2015-04-30

description: Simple template to deploy a single compute instance

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: my_key
      image: ubuntu-trusty-x86_64
      flavor: m1.small
```

Image: openstack.org

Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

OpenStack Monitoring

- OpenStack Monasca
  - Monitoring-as-a-Service
    - High-speed metrics
    - Log
    - Alarm and notification engine
  - Integrates with OpenStack
    - But also with Kubernetes, Docker, …
  - Grafana integration
- Various alternatives. E.g.,
  - Prometheus
  - ELK
    - ElasticSearch, LogStash, Kibana

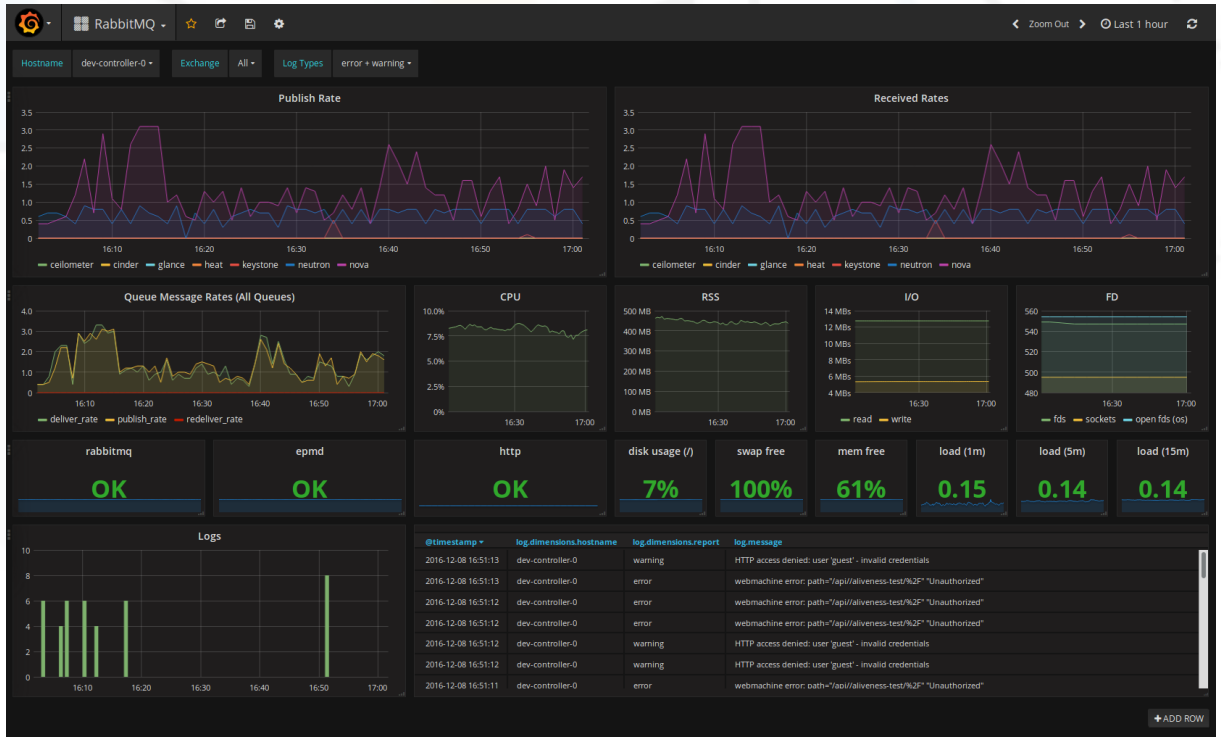


Image: stackhpc.com

OpenStack Packaging

- OpenStack deployment is not(!) trivial
  - Service-by-service, using Puppet / Chef / Ansible / …
  - Networking is a major hurdle
- "OpenStack in a Box"
  - DevStack
    - git clone, and build
    - Single machine / multi-node
    - Works with most Linux distros
  - RDO with packstack: RPM Distribution of OpenStack
    - Somewhat easier packaging than DevStack
    - yum install –y
    - Targeted mostly at CentOS
      - Also Red Hat Enterprise Linux (RHEL) and Fedora
- Production-level packaging (e.g., TripleO)

Infrastructure orchestration scripting tools

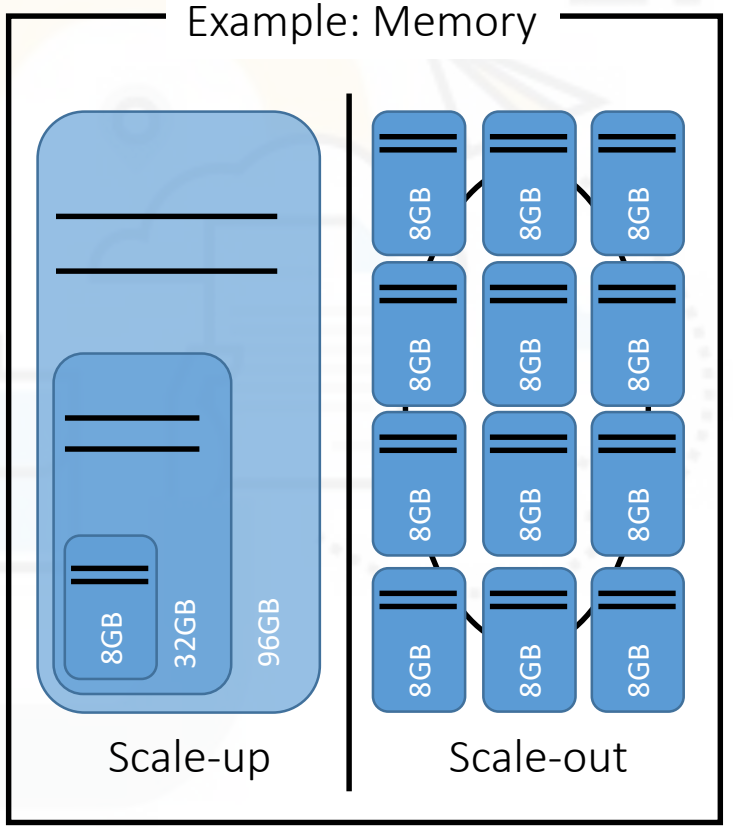
OpenStack-on-OpenStack:
Automate the deployment of OpenStack using OpenStack's services (e.g., Nova, Neutron, Heat, etc.)

Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

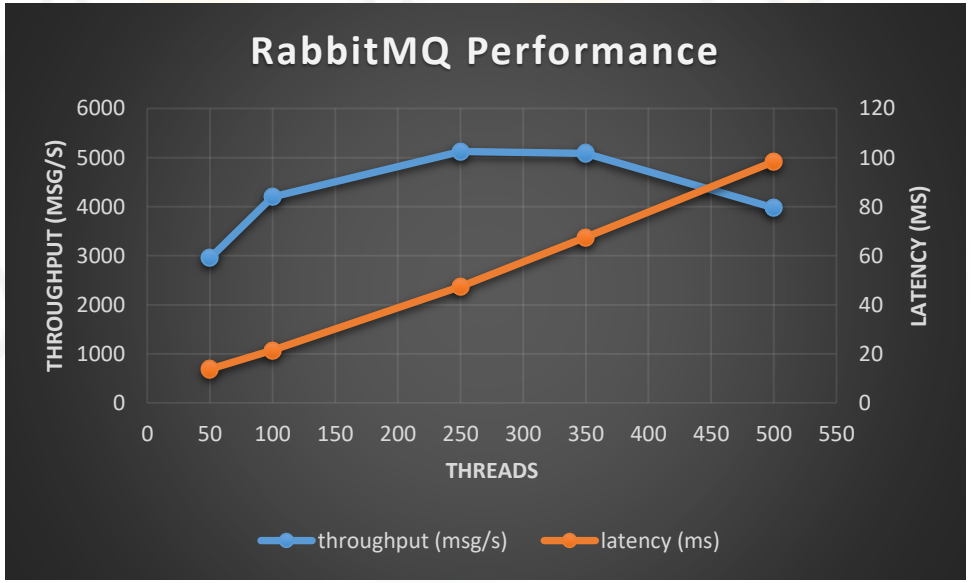
Scalability

- Scalability (sometimes a.k.a. Elasticity)
  - The ability to scale a service as workload intensity grows
    - Networking, compute, storage, …
    - Preferably in an automated manner
      - E.g, using orchestration scripts / manifests
      - Based on monitoring resource utilization
  - Scale-up (vertical scaling)
    - Use more powerful infrastructure
      - E.g., faster/stronger servers, larger/faster disks, fiber-optics instead of copper for networks, etc.
  - Scale-out (horizontal scaling) ← like we've seen in Kubernetes
    - Use more COTS infrastructure
      - More commodity servers, more disks, arrange small switches into Clos networks
    - Requires protocols for managing the more "complex" infrastructure



OpenStack: (Some) Scalability Bottlenecks

- Focus on OpenStack architecture-based bottlenecks
  - Keystone
    - EVERYTHING goes through keystone…
  - Service DBs
    - Maintain service state
    - DB access becomes a bottleneck
  - MQs
    - Throughput scales poorly
  - …
- Other bottlenecks (physical infrastructure)
  - Physical network
  - Physical node capabilities
    - CPUs (not vCPUs…), disk sizes, etc.



Source data:
https://docs.openstack.org/developer/performance-docs/

Outline

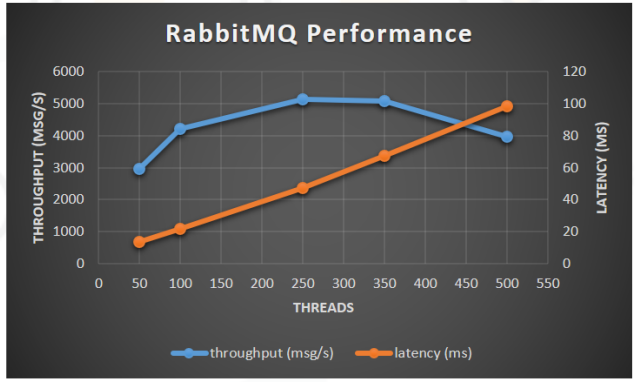
- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

Nova Cells

- Motivation

OpenStack: (Some) Scalability Bottlenecks

- Focus on OpenStack architecture-based bottlenecks
  - Keystone
    - EVERYTHING goes through keystone…
  - Service DBs ← get too large
    - Maintain service state
    - DB access becomes a bottleneck
  - MQs ← handle too much traffic
    - Throughput scales poorly
- Other bottlenecks (physical infrastructure)
  - Physical network
  - Physical node capabilities
    - CPUs (not vCPUs…), disk sizes, etc.



Source data:
https://docs.openstack.org/developer/performance-docs/

Nova Cells

- Logically partitioning compute resources
  - “Divide-and-conquer” (or rather “Divide-and-prosper”)
  - Facilitates easier scaling (for very large deployment) + fault tolerance
- Nova managed by ***super conductor***
  - Handles Nova API calls
  - Doesn’t manage any compute nodes directly
  - Interacts with Nova scheduler
    - How? Coming up next…
- Each cell:
  - Managed by cell-conductor
    - Interacts with super conductor (through cell MQ)
    - Directly manages its compute nodes
      - Separate MQ and DB
        - DB: what’s actually running in cell’s compute nodes

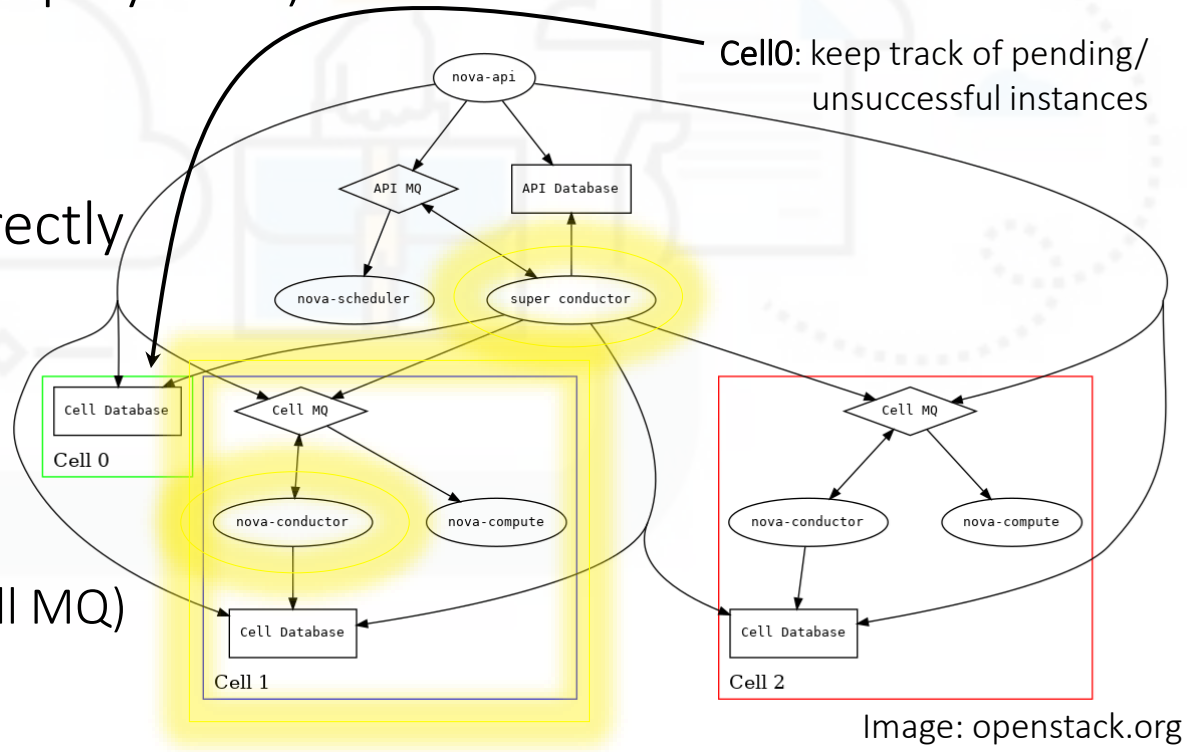
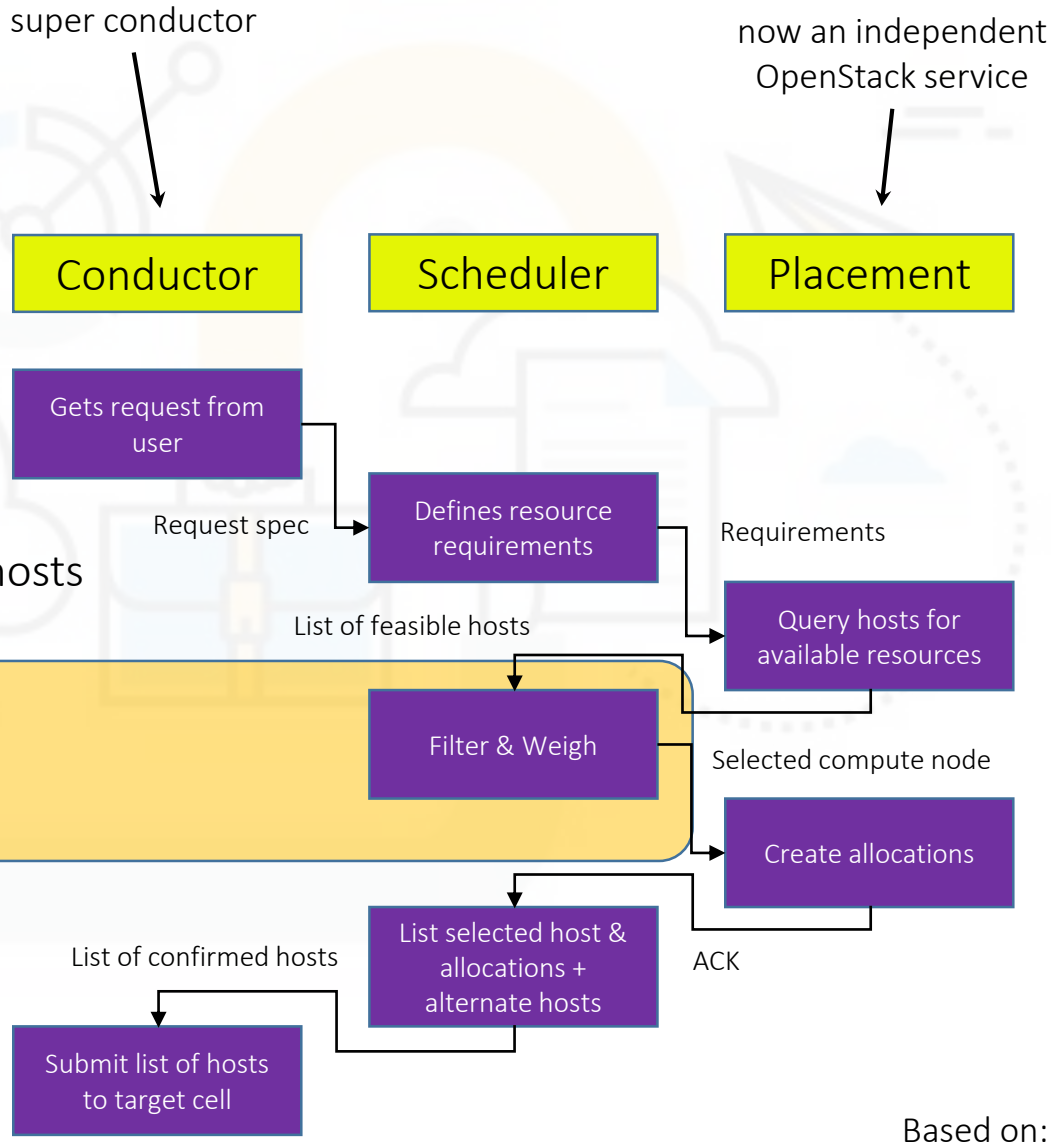


Image: openstack.org

OpenStack Scheduling

- The process behind instance deployment
  - Nova conductor
    - “front-end” for Nova scheduler
    - Some main tasks:
      - Instructs cell-conductor about actual deployment
      - Manages the Nova DB, and communicates with `nova-compute` agents on hosts
        - In simple (no-cells) deployments
  - Nova scheduler
    - Intermediate service
    - Main task: Filter & Weigh
      - Pick and prioritize acceptable/feasible hosts
  - Placement
    - Interacting with hosts
      - Query for state / available resources
      - Defines allocations of resources for selected hosts



Based on:
https://docs.openstack.org/nova/latest/reference/scheduling.html

OpenStack Scheduling

- Filter Scheduler
  - Filtering
    - Eliminate inappropriate/infeasible hosts
      - Resources
        - E.g., memory, #instances running, …
      - Policy
        - E.g., anti-affinity, tenant-isolation, …
  - Weighing
    - Apply a weight function on valid hosts characteristics to prioritize them
      - $v(r)$: amount of available resource on host
      - $w(r)$: weight (positive/negative)
    - Multi-dimensional bin packing….
      - Common heuristic: reduce to single dimension
    - Default: WorstFit based on Memory: $w_{\text{memory}} \cdot v(\text{memory}) \quad (w_{\text{memory}} > 0)$
    - Top priority: highest weight!
      - Non-increasing order

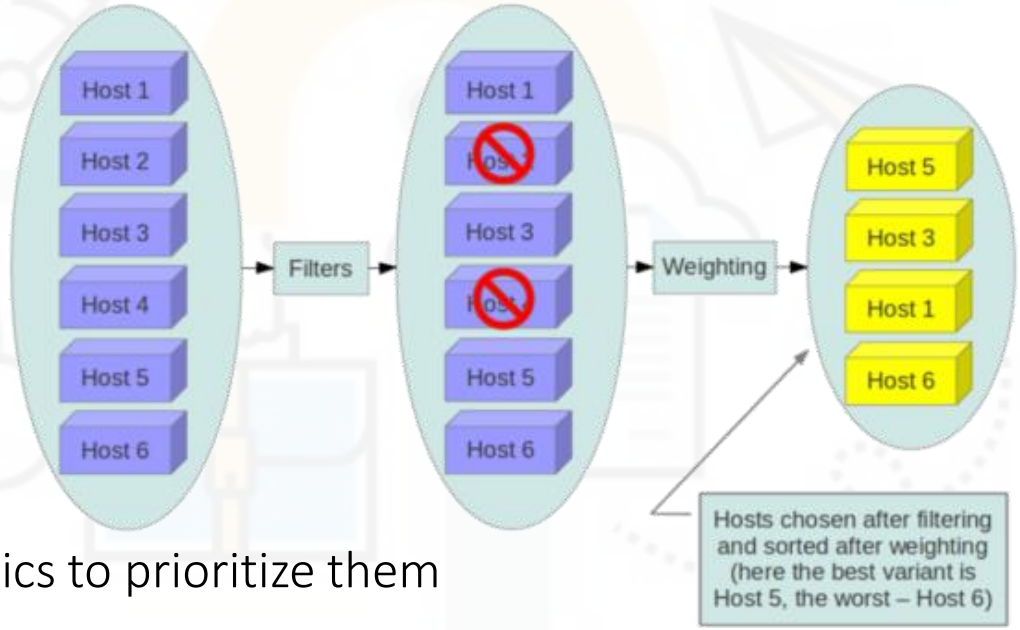


Image: openstack.org

$$\sum_{\text{resource } r} w_r \cdot \left\| \frac{v(r)}{\|v(r)\|} \right\|$$

normalized values

BestFit if $w < 0$

Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

Load Balancing Storage

- Problem statement:
  - Assume we want to store a LOT of objects in an n-node distributed storage system
  - How should we determine where to store each item??
- Requirements:
  - Finding item location fast
  - Might need to increase/decrease storage capacity
    - Avoid copying items (as much as possible…)
      - E.g., if 1 node fails/joins, re-map $\sim\frac{1}{n}$ fraction of objects
        - I.e., load-balanced…

Load Balancing Storage

- Problem statement:
  - Assume we want to store a LOT of objects in an n-node distributed storage system
  - How should we determine where to store each item??
- Naïve approach: `hash(ObjectName)` mod n
  - Finds item location fast!!
- What happens if we add another node?

`hash(ObjectName)`		mod 4
011100	28	0
101101	45	1
011011	27	3
111010	58	2
100010	34	2
010101	21	1
110010	50	2

Load Balancing Storage

- Problem statement:
  - Assume we want to store a LOT of objects in an n-node distributed storage system
  - How should we determine where to store each item??
- Naïve approach: `hash(ObjectName)` mod n
  - Finds item location fast!!
- What happens if we add another node?

`hash(ObjectName)` mod n

↓

`hash(ObjectName)` mod $n + 1$

- How many objects should be moved?

not so fast…

`hash(ObjectName)`		mod 4	mod 5
011100	28	0	3
101101	45	1	0
011011	27	3	2
111010	58	2	3
100010	34	2	4
010101	21	1	1
110010	50	2	0

Load Balancing Storage

- Problem statement:
  - Assume we want to store a LOT of objects in an n-node distributed storage system
  - How should we determine where to store each item??
- Naïve approach: `hash(ObjectName)` mod n
  - Finds item location fast!!
- What happens if we add another node?

`hash(ObjectName)` mod n

↓

`hash(ObjectName)` mod $n + 1$

- How many objects should be moved?
  - Almost all…
  - The same holds if we remove a node

`hash(ObjectName)`		mod 4	mod 5
011100	28	0	3
101101	45	1	0
011011	27	3	2
111010	58	2	3
100010	34	2	4
010101	21	1	1
110010	50	2	0

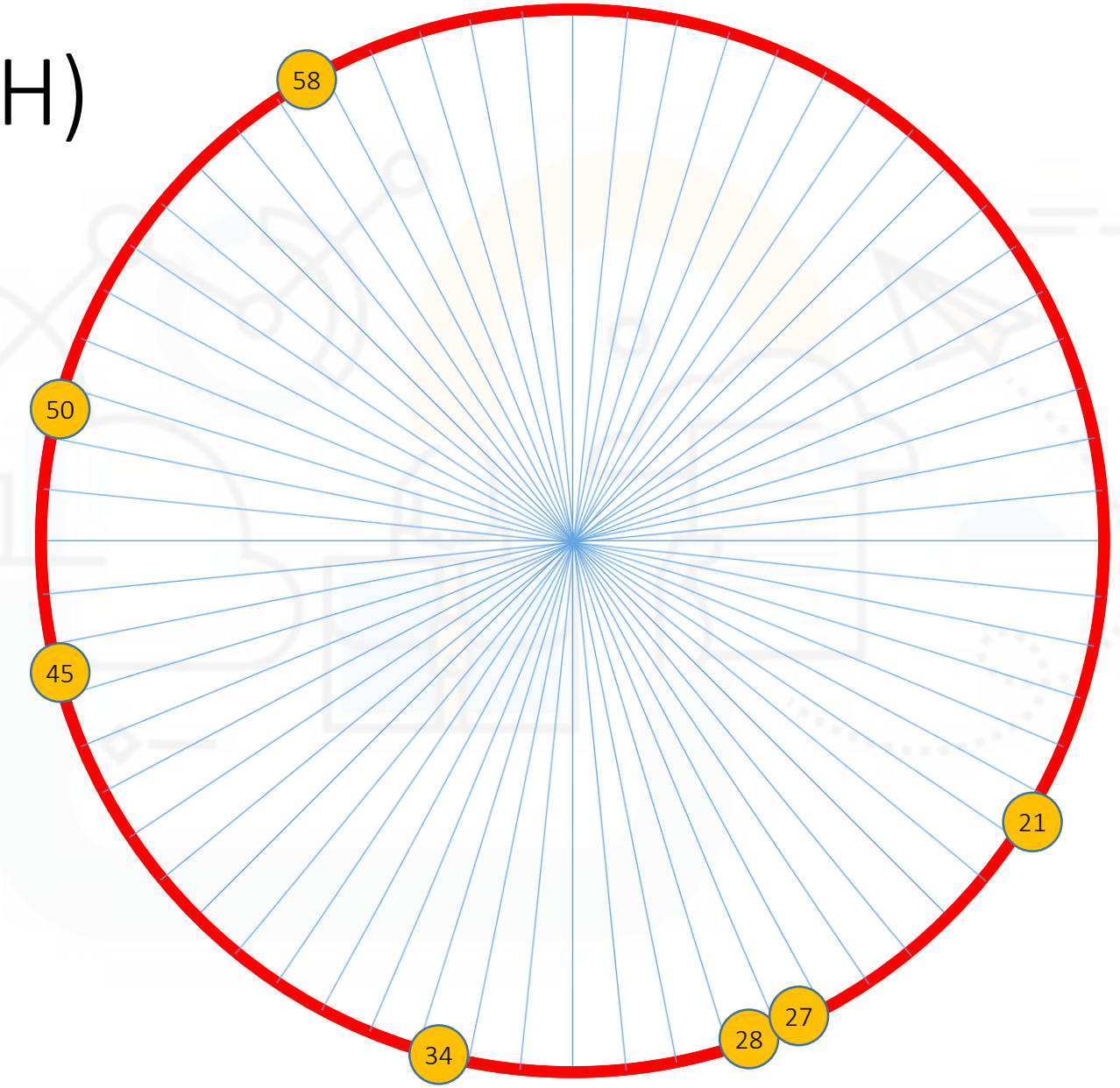
Consistent Hashing (CH)

- Hash objects onto a large ***ring***
  - mod m

Consistent Hashing (CH)

- Hash objects onto a large ***ring***
  - mod m
    - m = hash size
      - Here $2^6 = 64$
      - For MD5, $m = 2^{128}$

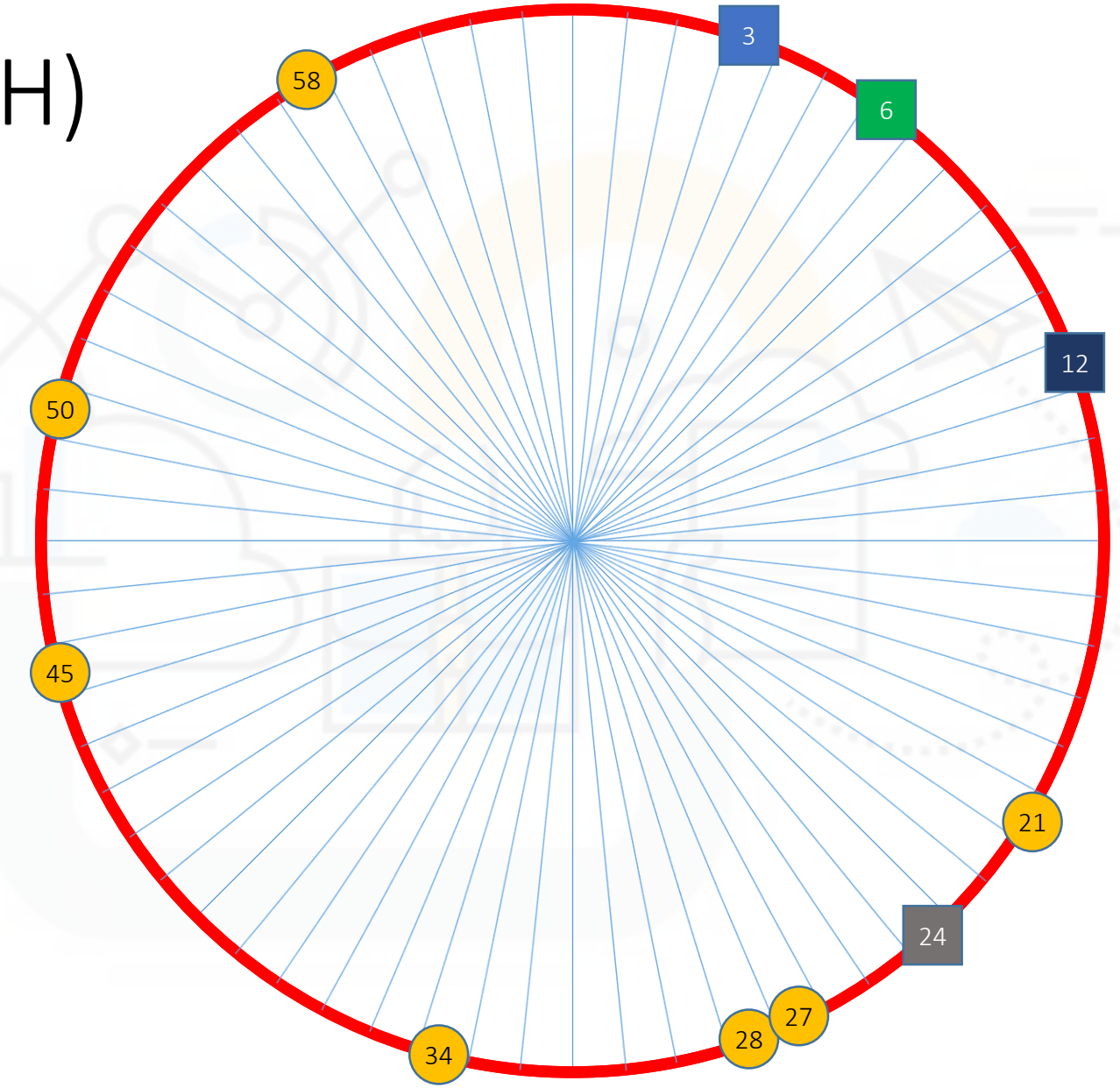
hash(ObjectName)	
011100	28
101101	45
011011	27
111010	58
100010	34
010101	21
110010	50



Consistent Hashing (CH)

- Hash ***nodes*** onto ring
  - Use some hash for node “names”
  - E.g., $n = 4$ servers

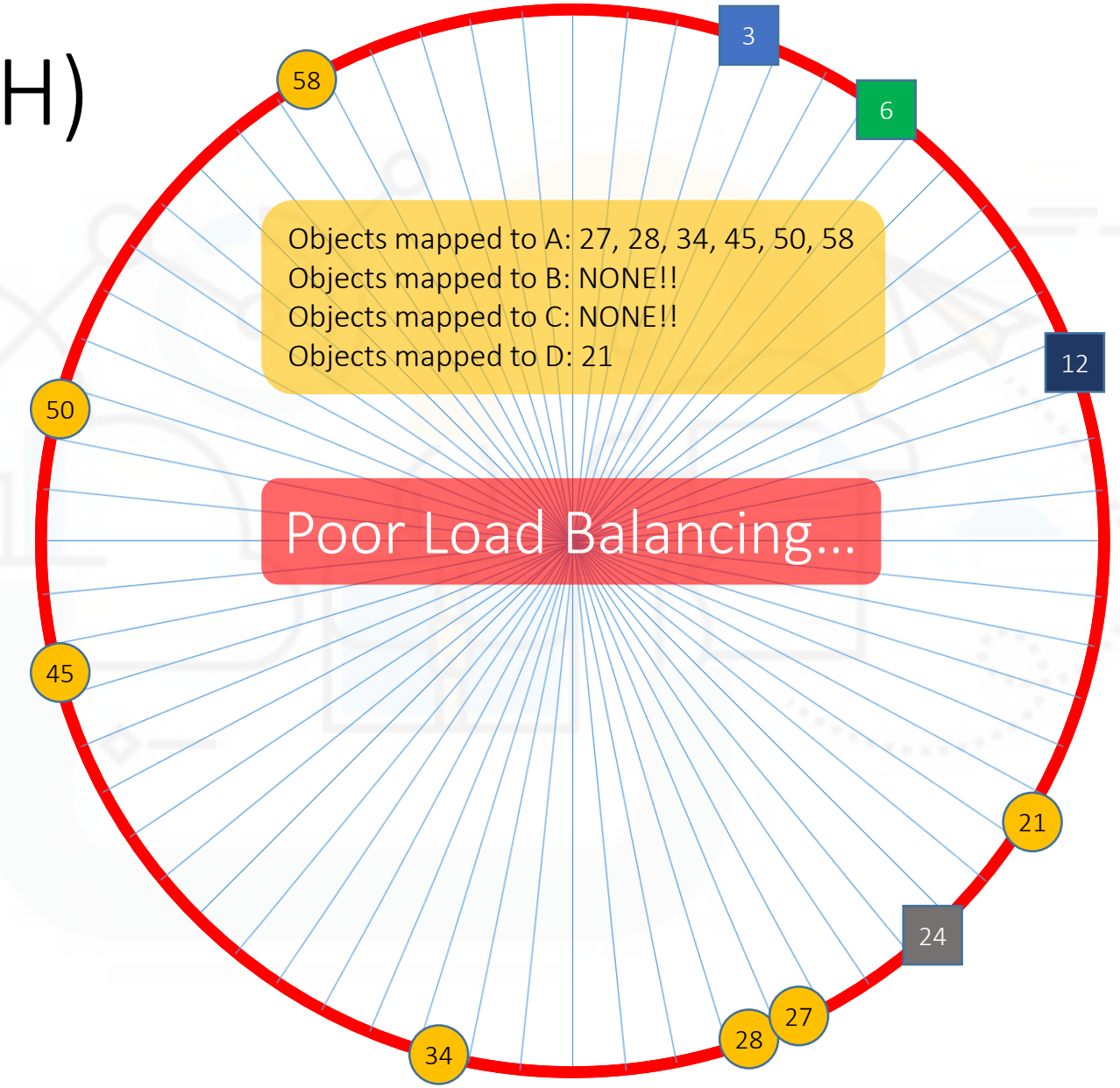
NodeName	hash(NodeName)	
A	000011	3
B	000110	6
C	001100	12
D	011000	24



Consistent Hashing (CH)

- Map each object to closest node
  - Clockwise / counter-clockwise
  - Just be consistent!!

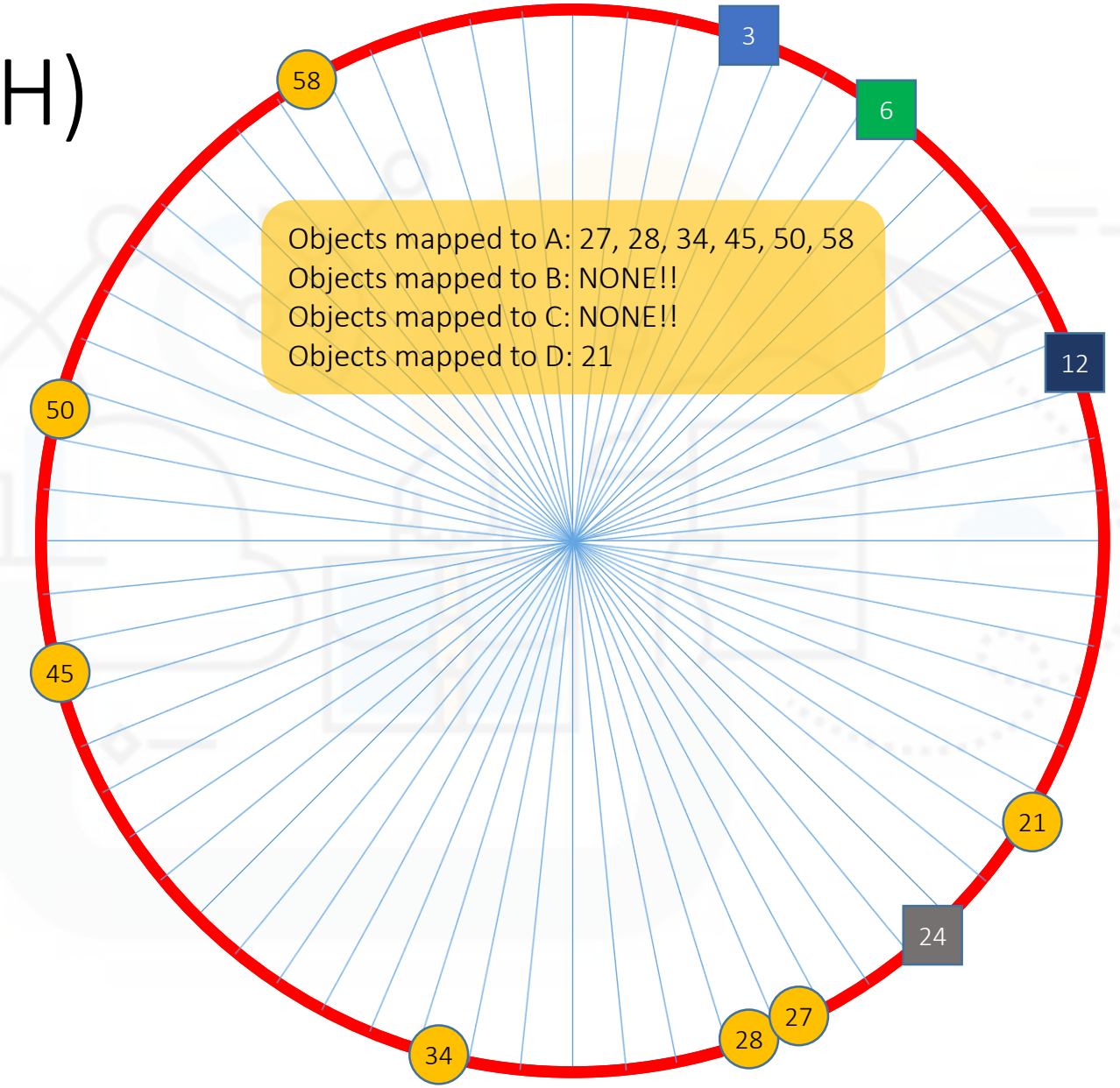
NodeName	hash(NodeName)	
A	000011	3
B	000110	6
C	001100	12
D	011000	24



Consistent Hashing (CH)

- Better load balancing:
  - Hash ***virtual node names*** onto ring

NodeName	hash(NodeName)	
A	000011	3
B	000110	6
C	001100	12
D	011000	24



Consistent Hashing (CH)

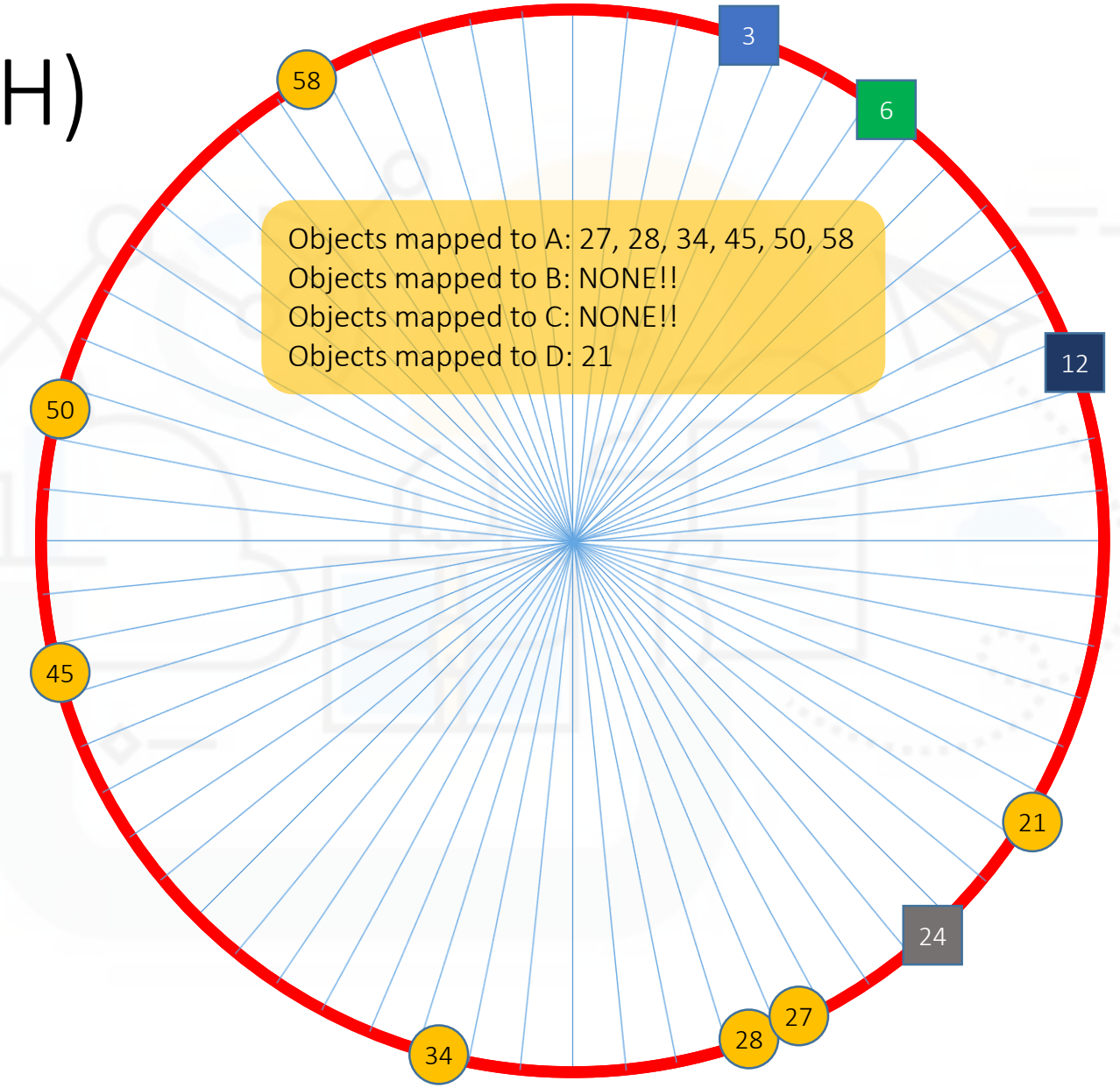
- Better load balancing:
  - Hash ***virtual node names*** onto ring

NodeName	hash(NodeName)	
A1	000011	3

NodeName	hash(NodeName)	
B1	000110	6

NodeName	hash(NodeName)	
C1	001100	12

NodeName	hash(NodeName)	
D1	011000	24



Consistent Hashing (CH)

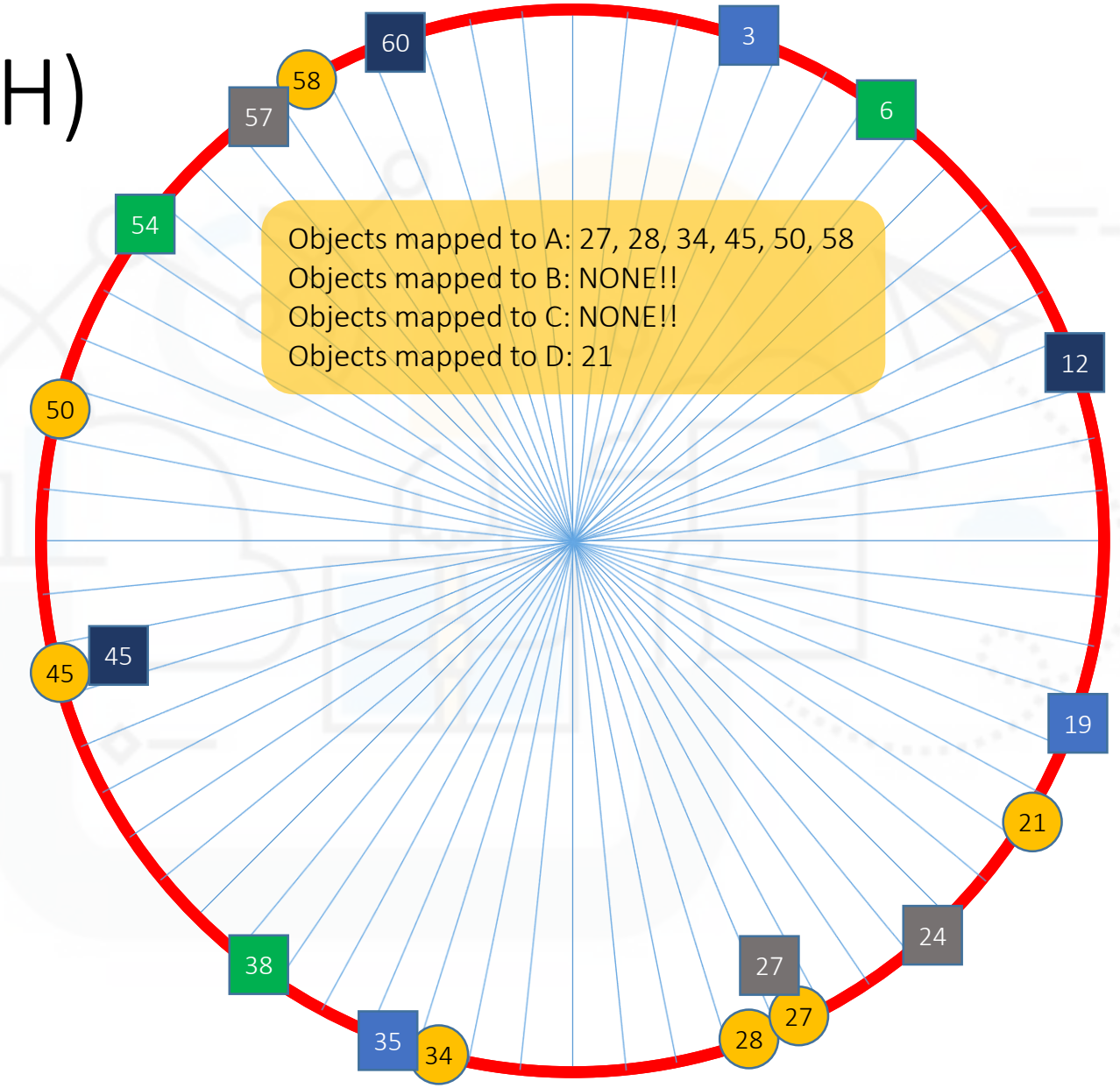
- Better load balancing:
  - Hash ***virtual node names*** onto ring

vNodeName	hash(vNodeName)	
A1	000011	3
A2	010011	19
A3	100011	35

vNodeName	hash(vNodeName)	
B1	000110	6
B2	101110	38
B3	110110	54

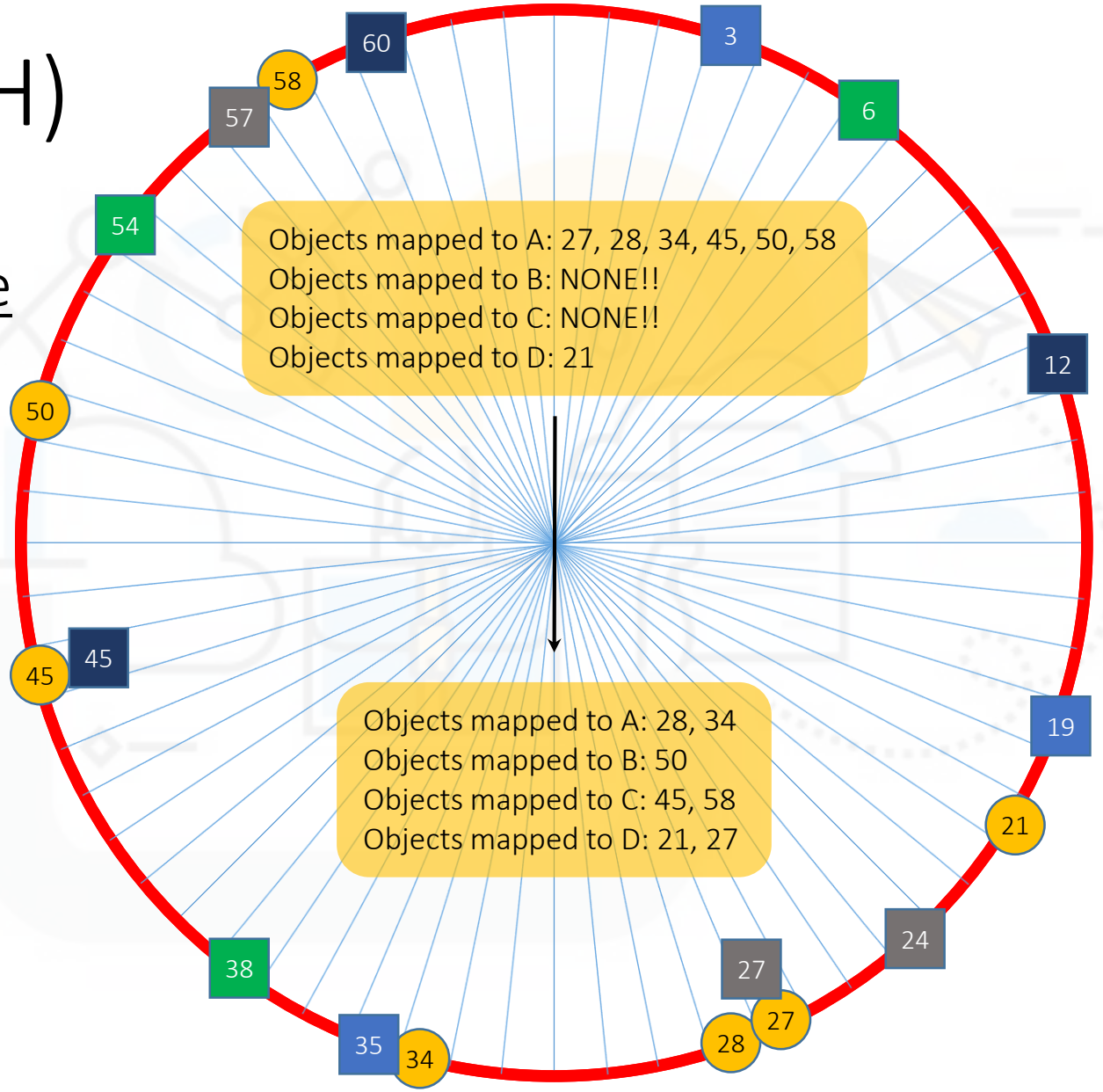
vNodeName	hash(vNodeName)	
C1	001100	12
C2	101101	45
C3	111100	60

vNodeName	hash(vNodeName)	
D1	011000	24
D2	011011	27
D3	111001	57



Consistent Hashing (CH)

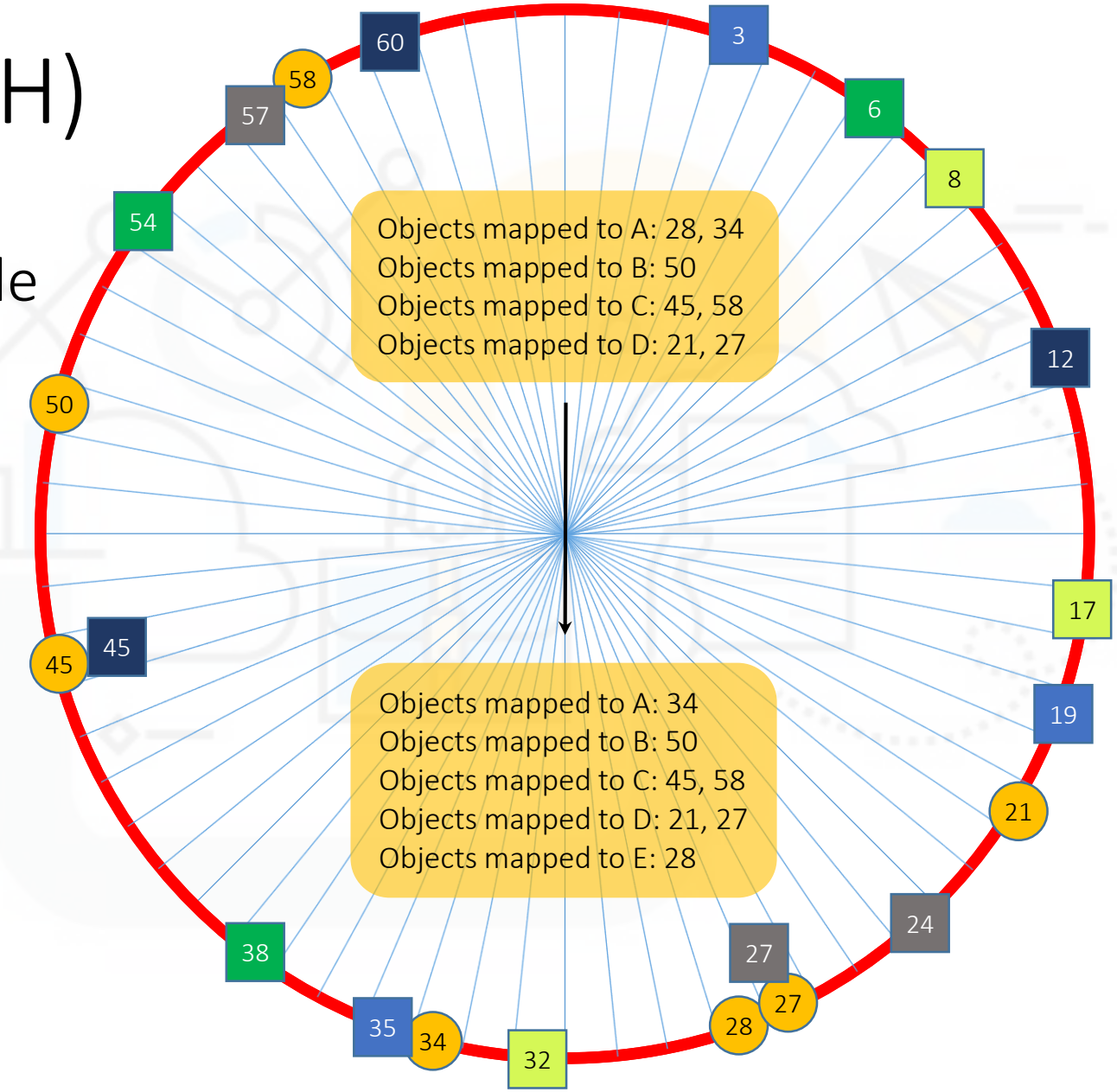
- Map each object to closest vNode
  - Consistently!



Consistent Hashing (CH)

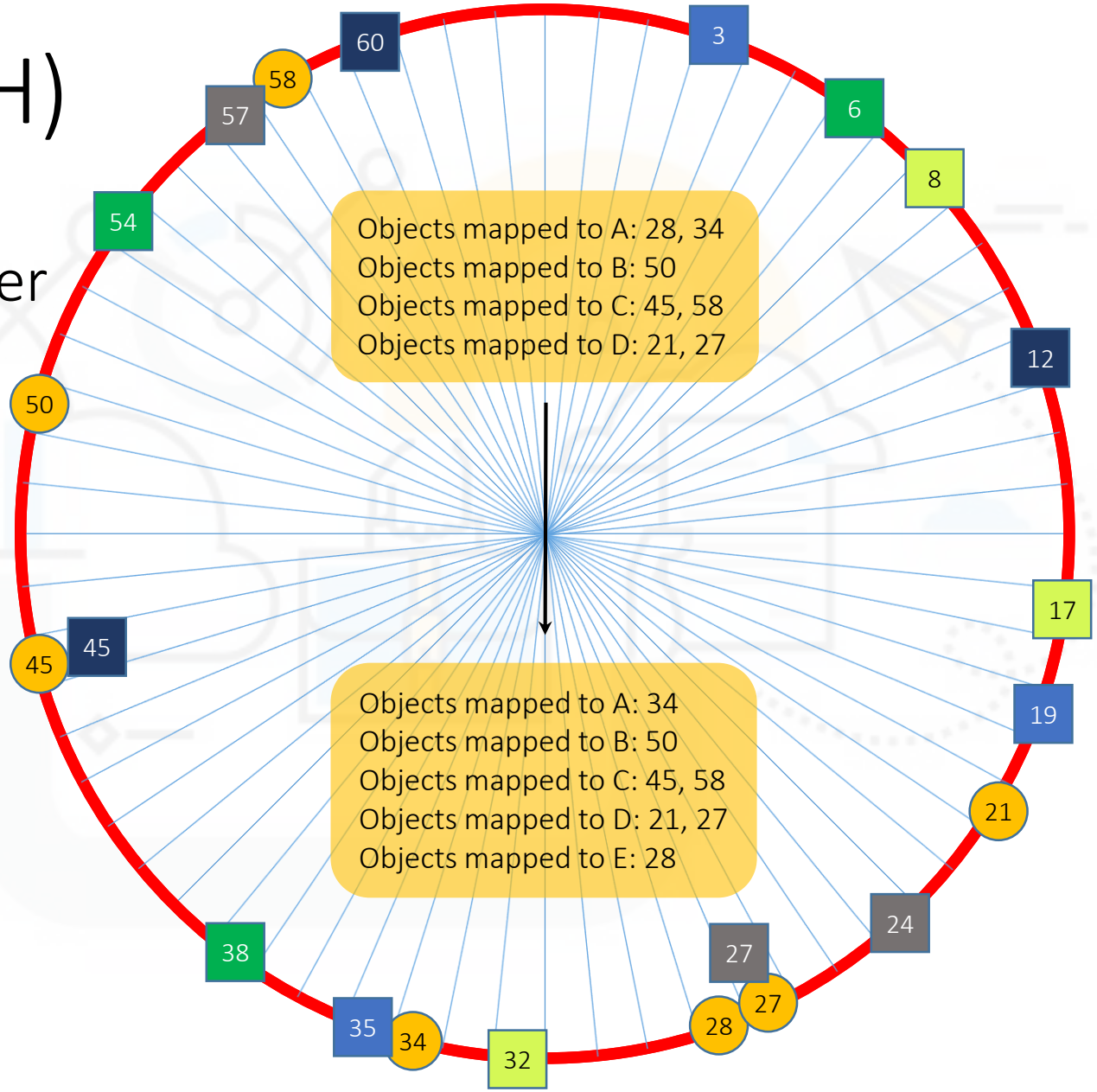
- The effect of adding another node
  - Only ***few, local*** re-mappings
  - Here, only object 28 re-mapped
    - From A to E

vNodeName	hash(vNodeName)	
E1	001000	8
E2	010001	17
E3	100000	32



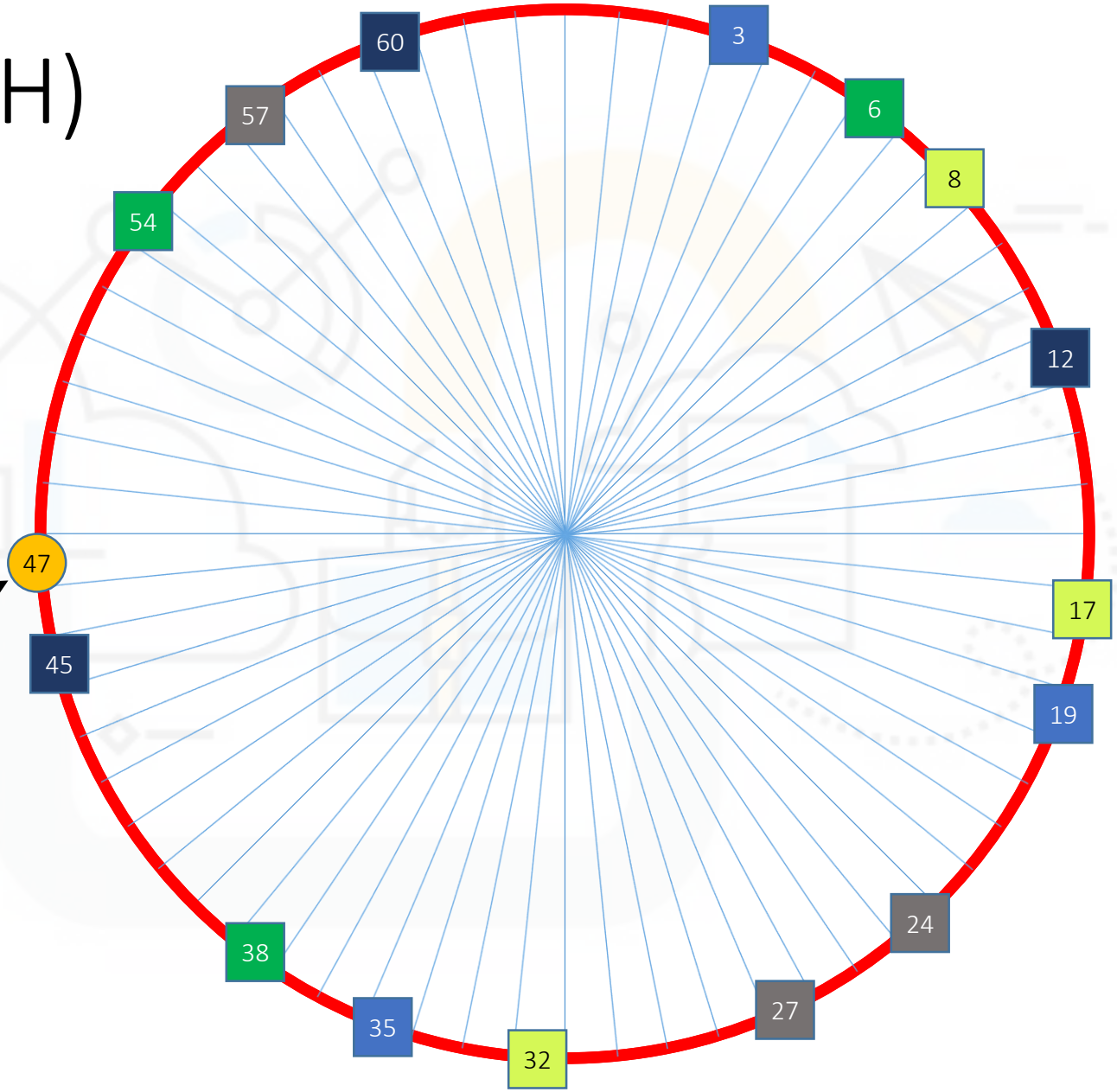
Consistent Hashing (CH)

- The effect of adding another server
  - Only ***few, local*** re-mappings
  - Here, only object 28 re-mapped
    - From A to E
  - No re-mapping between “old” vNodes
    - Hence, no remapping between “old” nodes
- Similarly for removing a node
  - Only few, local re-mappings
  - Only re-map items mapped to removed node



Consistent Hashing (CH)

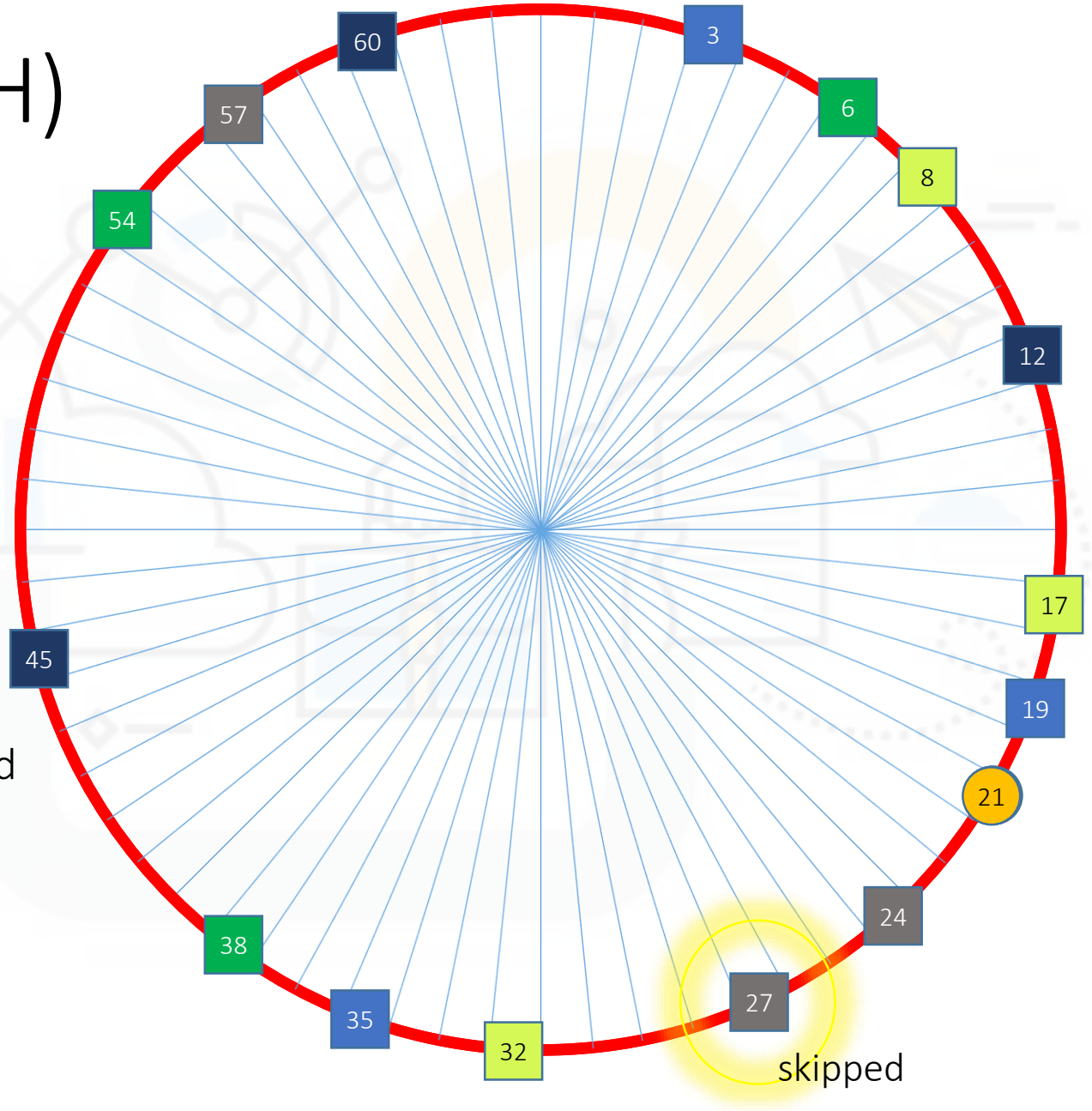
- Querying for an object
  - Find `hash(ObjectName)`
  - Move to closest vNode
    - In the right direction…
  - E.g., an object with `hash(ObjectName)` = 47
    - Closest vNode (clockwise)
      - 54 = `hash(B3)`
    - Can be found (if exists) at node B



vNodeName	hash(vNodeName)	
B1	000110	6
B2	101110	38
B3	110110	54

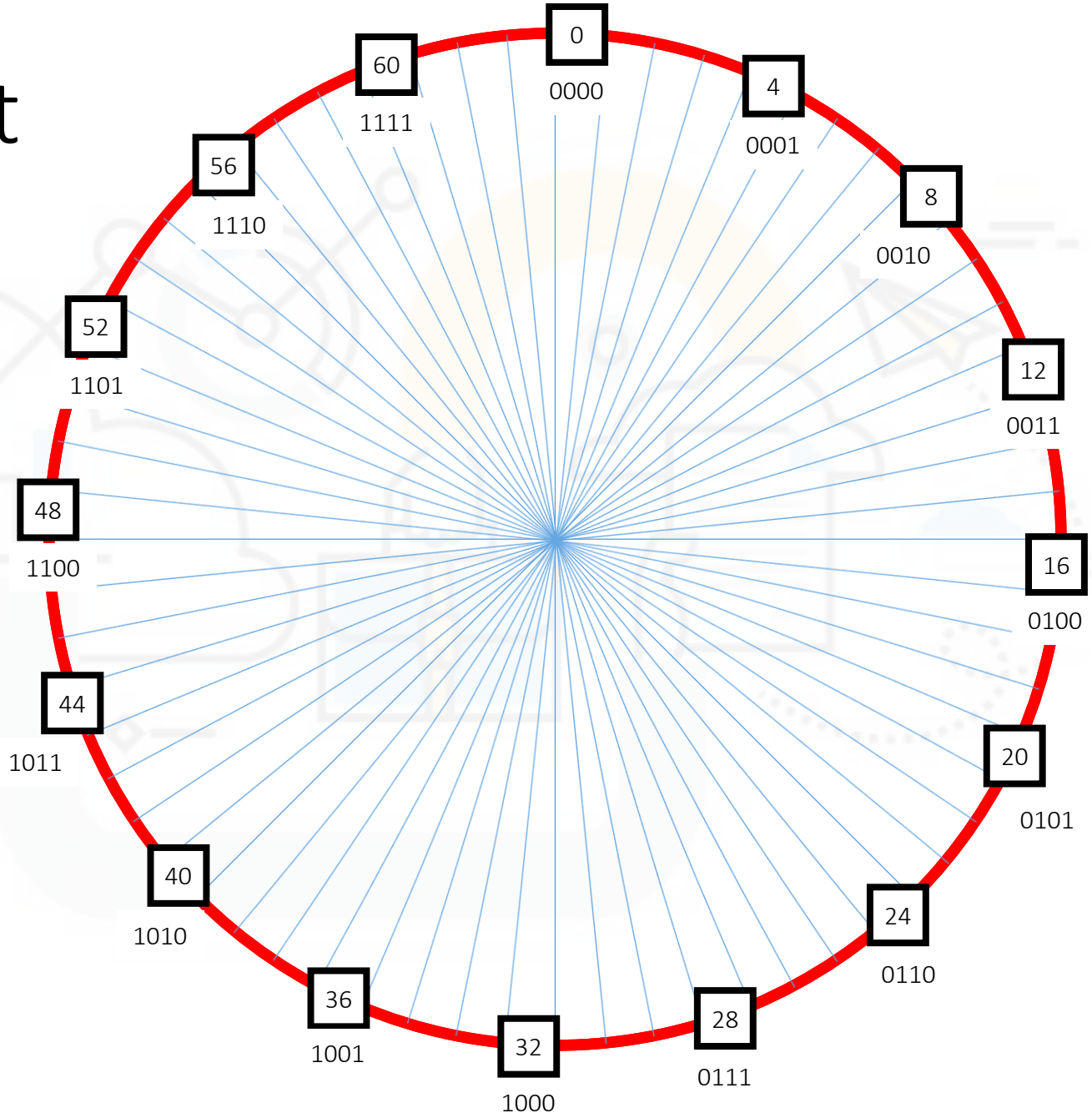
Consistent Hashing (CH)

- Inserting an object
  - Similar…
- Replication in k severs:
  - Similar, but
    - Should store in k closest ***distinct*** vNodes
    - Skip servers that were already selected
    - E.g., `hash(21)` → D, E, A



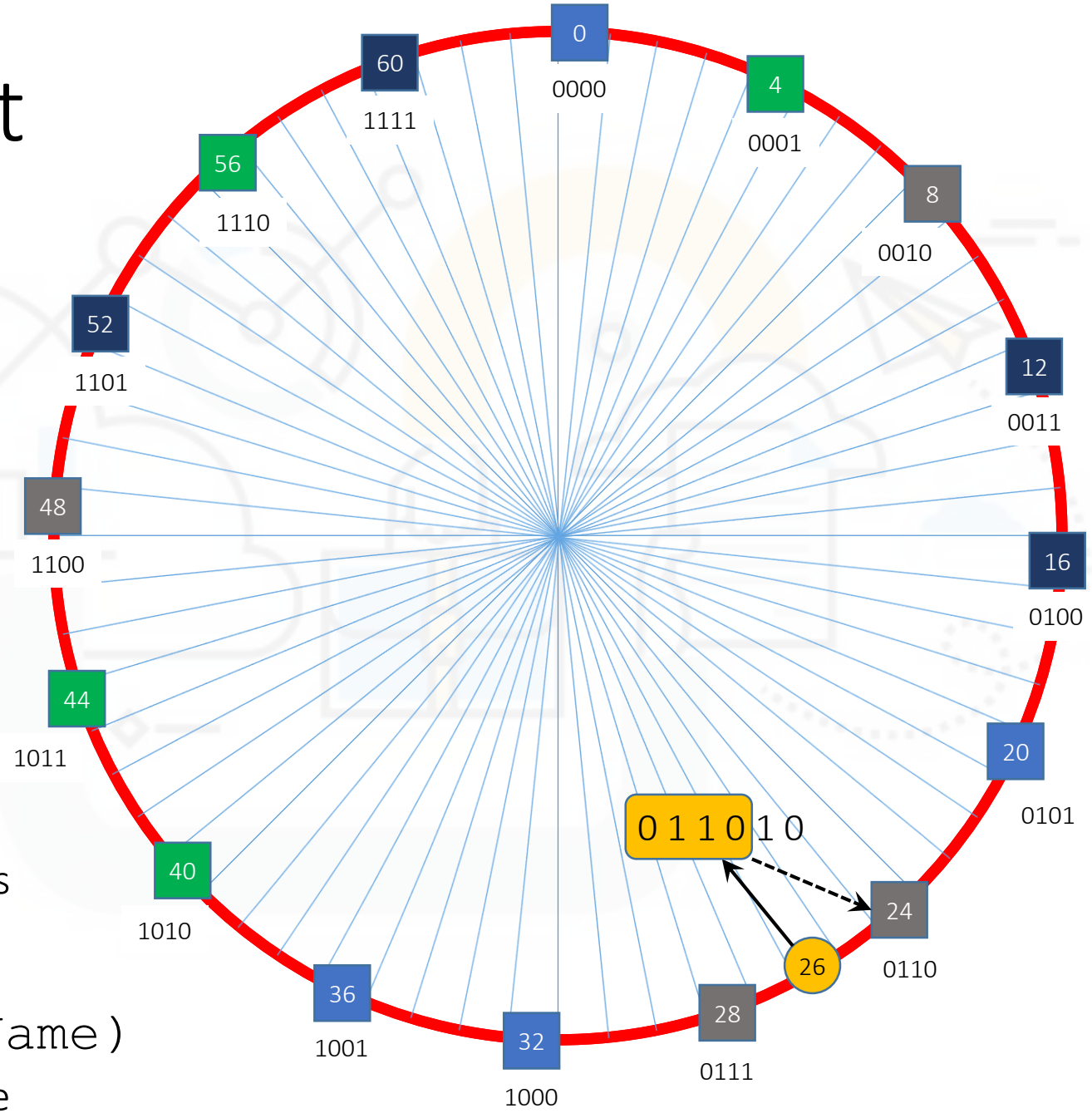
CH in OpenStack Swift

- Potential k vNode slots ***evenly*** spaced on ring
  - Example:
    - 4 nodes, with 4 vNodes each
    - k = 16 specific slots
    - Each defined uniquely by $\log(k)$ = 4 MSBs



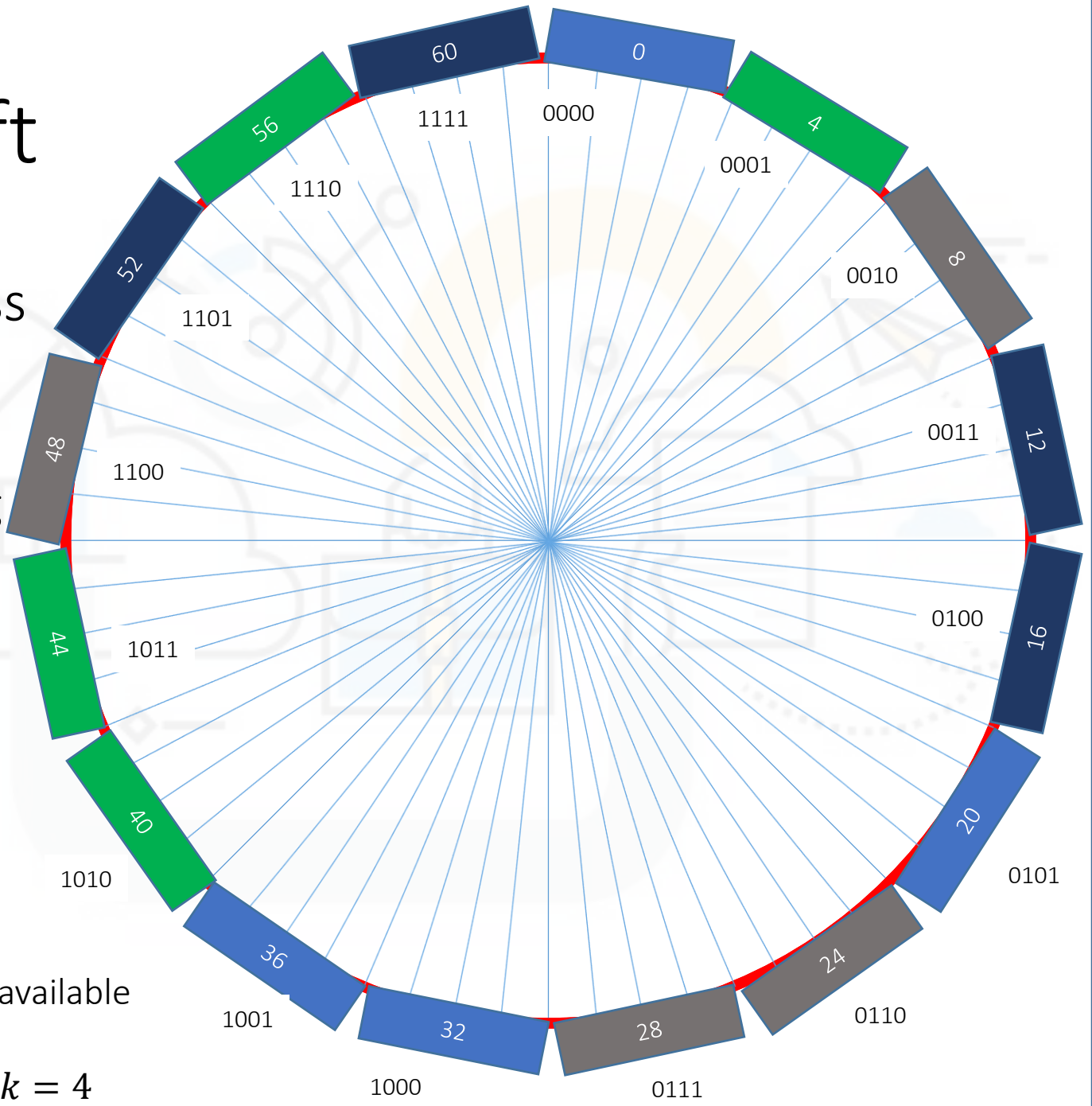
CH in OpenStack Swift

- Potential k vNode slots ***evenly*** spaced on ring
  - Example:
    - 4 nodes, with 4 vNodes each
    - k = 16 specific slots
    - Each defined uniquely by $\log(k)$ = 4 MSBs
- Randomly distribute vNodes
  - Each vNode "hashed" to slot
    - Unique slot, defined by $\log(k)$ MSBs
- Object location:
  - $\log(k)$ MSBs of `hash(ObjectName)`
    - In this illustration, counter-clockwise



CH in OpenStack Swift

- vNode distribution + MSB access
  - Induces the partition of the ring
- Partition
  - Each segment of slots on the ring
- Number of partitions
  - # potential vNode locations
    - Determined by the ***partition power***
      - $2^{\text{part\_power}}$ = #partitions
    - Larger -> more load balance
      - But also larger memory footprint
    - Rule of thumb:
      - `part_power` = $\lceil \log_2(k \cdot n) \rceil$
        - n = number of physical disks available
        - $k \sim 100$ vNodes / disk
      - In this illustration we have $n = 4, k = 4$



CH Elsewhere

- Key element in various technologies and applications
  - CH, or similar solutions
- E.g.,
  - Caching systems
    - Memcached distributed in-memory cache (multi-server configuration)
      - Speeding up DB performance by caching content in memory
    - Web caching was the original motivation for CH paper (Karger et al., (1997))
  - Distributed hash tables
    - Chord – can be viewed as a distributed version of CH
      - Used extensively in P2P networks
      - E.g., Bittorrent's tracker uses Kademlia, which is based on Chord
  - Distributed Databases
    - Cassandra (NoSQL) distributed datastore

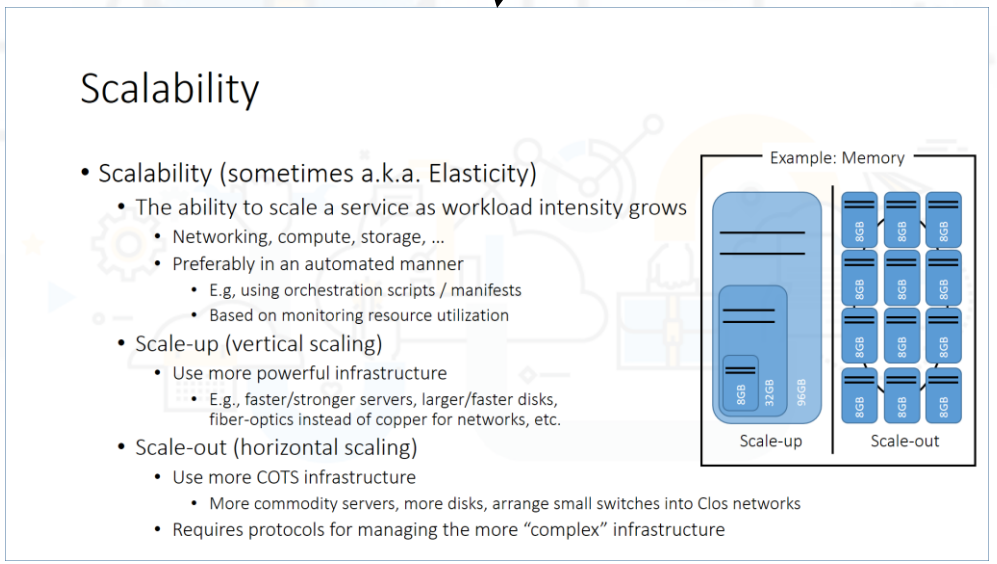
Outline

- Introduction to Storing Data in the Cloud
- Cloud Management & OpenStack
  - Architecture
  - Specific components
  - Misc
- Additional Topics
  - OpenStack Nova: Scheduling
  - OpenStack Swift: Consistent Hashing
  - High-Availability (HA) & Consistency

High-Availability (HA)

- So, what is HA all about?
  - Maintain service (full) functionality ***at all times***: “24/7/365”
    - (Try to) ensure that failure never happens
    - Identify failure (when it happens)
    - Install measures to recover from failure (after it happens)

this is NOT scalability!!



High-Availability (HA)

- So, what is HA all about?
  - Maintain service (full) functionality at all times: “24/7/365”
    - (Try to) ensure that failure never happens
    - Identify failure (when it happens)
    - Install measures to recover from failure (after it happens)
- How is HA achieved?
  - Redundancy
    - E.g, replication + load balancing
  - Monitoring
    - E.g., Anomaly detection over telemetry data, heartbeat check
  - Failover
    - Active/Active: 2 servers active at all times, serving requests concurrently, LBed
    - Active/Passive: first server active, second server kicks in upon failure of first

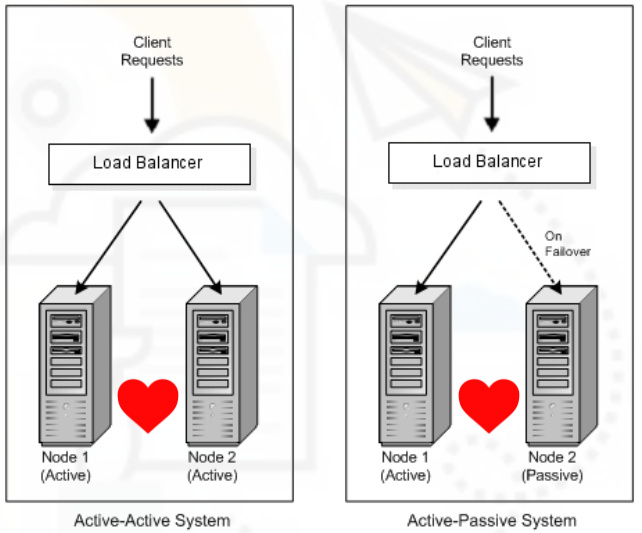


Image: oracle.com

usually across distinct geo-regions

operation cost vs. downtime

High-Availability (HA)

- How HA is measured?
  - 9s….
    - Percentage of time (minutes) that the system/service is available (in a year)

	Availability %	Downtime (min)	Downtime (year)	Referred to as…
	90	52,560.00	36.5 day	one nine
	99	5,256.00	4 day	two nines
	99.9	525.60	8.8 hour	three nines
	99.99	52.56	53 min	four nines
Gold Standard:	99.999	5.26	5.3 min	five nines
	99.9999	0.53	32 sec	six nines

- Guaranteed by Service-Level-Agreement (SLA)

High-Availability (HA)

- Why should I care about HA?
  - The Amazon incident (2013)

  - Was down around 15 - 40 minutes
    - Jury is still out on that…
  - Estimated sales loss of ~$66K / min → $1M - $2.5M loss during outage…
- Unavailable services affect
  - Business perspective: brand, consumer loyalty/satisfaction, revenue stream, …
  - Technical perspective: other services
    - Web server is up, but back-end DB isn't…
    - HA should be considered for entire infrastructure!

High-Availability (HA)

- OpenStack HA
  - Not “out-of-the-box”, but there are many tools to facilitate HA
  - Requires per-service configuration
- Some example mechanisms:
  - Virtual / floating IP
    - Allows restoring a service (e.g., in active/passive) by merely reassigning an IP
  - Load balancing
    - Stop directing traffic to failed node (e.g., using and configuring HAProxy in active/active)
  - Disjoint paths in the network topology
    - Ensures connectivity can be restored for each (single) link failure
- Tools:
  - HAProxy (http reverse-proxy LB), keepalived (health+LB), Pacemaker (cluster mgmt.)
  - memcached (redundant in-memory DB), RabbitMQ (HA message queues)

Consistency

- Originally a term defined for Databases
  - But nowadays applicable also to general distributed storage
- Transaction: set/sequence of operations to be applied as a single unit
  - E.g., money transfer:
    - Remove from account A, add to account B
  - Problem:
    - Race Conditions in multi-threaded programming
- Database properties/requirements
  - ACID
  - BASE

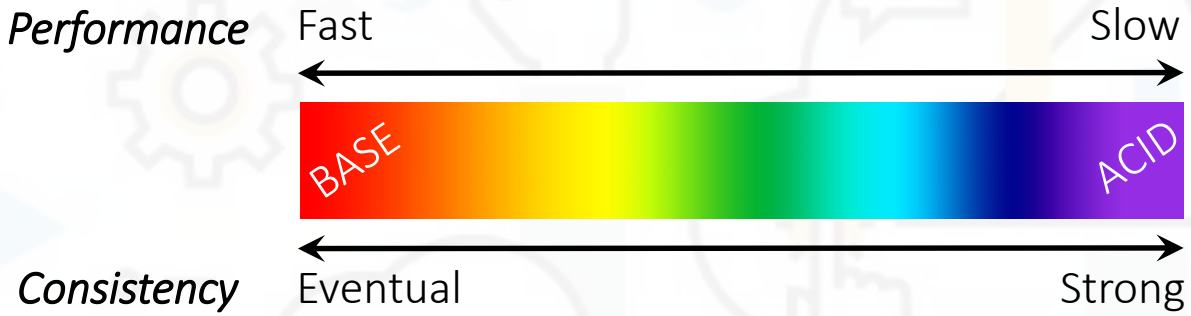
attainable for relatively structured data, e.g., relational DBs (SQL)

the (relaxed) properties applying to "Big-Data", e.g., NoSQL DBs

ACID	**A**tomicity	Transactions are all-or-nothing: Success iff all operations succeed. Otherwise: rollback!
	Consistency	Transactions move system between consistent states.
	Isolation	Transactions do not affect each other. Performed in-order.
	Durabitliy	Committed transactions persist (backups / recovery).
BASE	**B**asically **A**vailable	Data is distributed and replicated. (Mild) failure does not cause unavailability.
	Soft state	State might change even without inputs. Essentially related to lack of (strong) consistency.
	Eventually consistent	If no new input, the system will eventually become consistent.

The Best of Both Worlds

- ACID and BASE: Consistency extremes



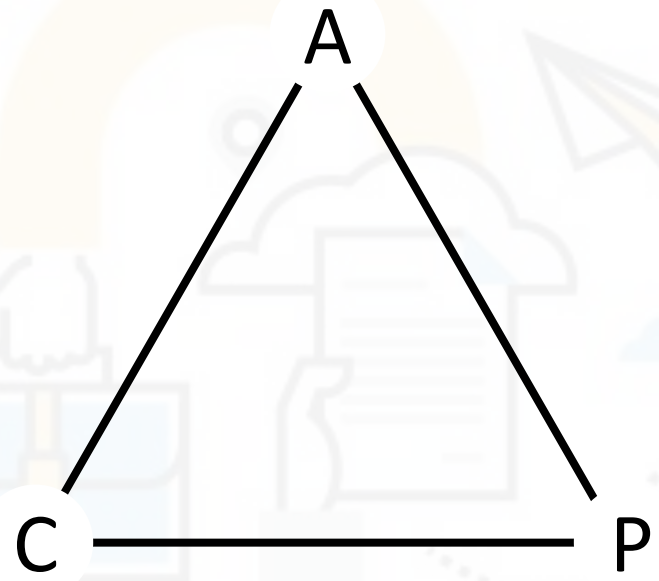
- Which approach would you pick?
  - Mission critical: ??
  - Online retail: ??
- Let's try and have both:
  - Accurate AND fast!!

ACID	**A**tomicity
	Transactions are all-or-nothing: Success iff all operations succeed. Otherwise: rollback!
	Consistency
	Transactions move system between consistent states.
	Isolation
	Transactions do not affect each other. Performed in-order.
	Durabitliy
	Committed transactions persist (backups / recovery).

BASE	**B**asically **A**vailable
	Data is distributed and replicated. (Mild) failure does not cause unavailability.
	Soft state
	State might change even without inputs. Essentially related to lack of (strong) consistency.
	Eventually consistent
	If no new input, the system will eventually become consistent.

The Best?? The CAP Theorem

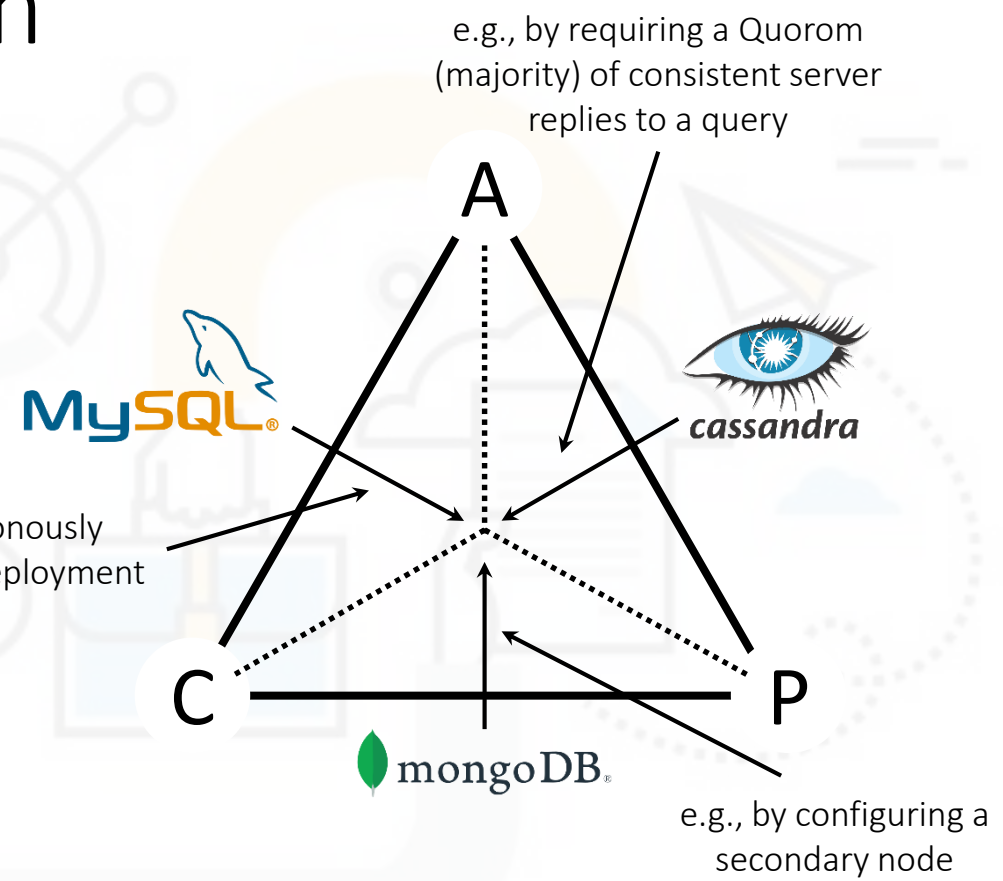
- Smaller set of requirements:
  - Consistency
    - Each client has the same view of the data
  - Availability
    - Each client can reach the data
      - Even upon some (but not all) node failure
      - Clearly requires distributing the data…
  - Partition-tolerance
    - Each client can still reach data even if some (but not all) of the network fails (i.e., messages are lost)
- Goal: CAP architectures!
  - We'll see it's not very hard…



- Theorem (Brewer, Gilbert&Lynch):
  - No system can satisfy all 3 properties of CAP
    - Must choose (at most) 2…

The Best?? The CAP Theorem

- No silver bullet
  - Any system must balance the tradeoffs between the properties
- Some examples:
  - Cassandra: A+P
    - Replication (availability), read might get stale copy
  - MongoDB: C+P
    - Single primary node ensuring consistency
  - MySQL: A+C
    - Each cluster is considered as a single node system
  - Some allow to “smoothly” tune the tradeoffs



- Theorem (Brewer, Gilbert&Lynch):
  - No system can satisfy all 3 properties of CAP
    - Must choose (at most) 2…

(Partial) Bibliography

- Marinescu, “Cloud Computing: Theory and Practice”, Morgan Kaufman (2018)
- https://www.ibm.com/cloud/learn/storage
- Weil et al., “Ceph: A Scalable, High-Performance Distributed File System”, OSDI (2006)
- Weil et al., “RADOS: a scalable, reliable storage service for petabyte-scale storage clusters”, PDSW (2007)
- Fisk, “Mastering Ceph”, Packt (2019)
- Shrivastwa and Sarat, “Learning OpenStack”, Packt (2015)
- Selvaraj, “OpenStack Bootcamp”, Packt (2017)
- Solberg & Silverman, “OpenStack for Architects”, Packt (2017)
- Carmin & Goldenberg, “Red Hat Enterprise Linux OpenStack Platform – Architecture and Terminology” (2015)
- Common NFVI Telco Taskforce (CNTT), “Reference Architecture: OpenStack”, (2020) https://github.com/cntt-n/CNTT
- https://www.openstack.org/
- van Vugt, “LFS152x: Introduction to OpenStack”, Linux Foundation (2016)
- Denton, “Learning OpenStack Networking (Neutron) Essentials”, Packt (2015)

(Some More Partial) Bibliography

- Khedher, “Extending OpenStack”, Packt (2018)
- “OpenStack Swift Architecture”, https://www.swiftstack.com/docs/introduction/openstack_swift.html
- https://docs.openstack.org/devstack/latest/
- https://www.rdoproject.org/
- Laski, “Nova Cells V2”, OpenStack Summit (2016)
- Karger et al., “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”, STOC (1997)
- Karger et al., “Web Caching with Consistent Hashing”, Computer Networks 31 (1999)
- Stoica et al., “Chord: a scalable peer-to-peer lookup protocol for internet applications”, IEEE/ACM Trans. Netw. 11(1) (2003)
- Clark et al., “Live Migration of Virtual Machines”, NSDI (2005)
- Brewer, “Towards Robust Distributed Systems”, PODC (2000)
- Gilbert & Lynch, “Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services”, SIGACT News 33(2) (2002)
- Pritchett, “BASE: An ACID Alternative”, ACM Queue 6(3) (2008)
- Brown, “Brewer's CAP Theorem”, http://www.julianbrowne.com/ (2009)