

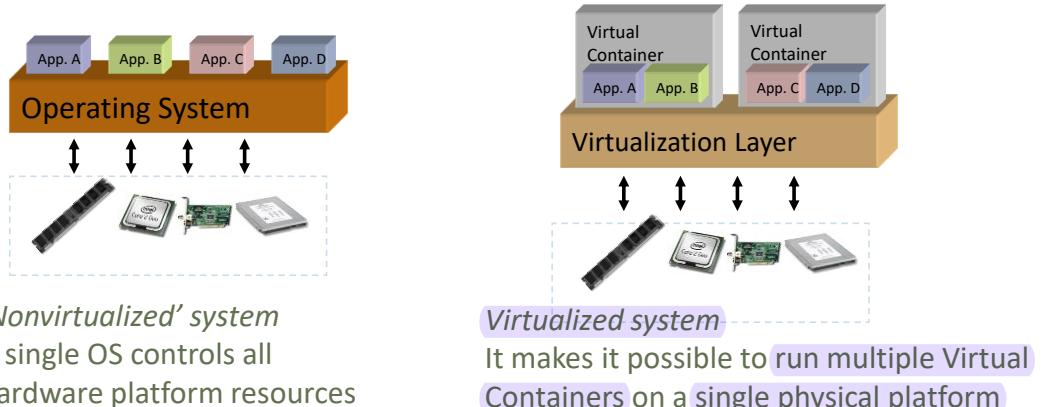
Virtualization

GIANLUCA REALI



What is virtualization?

- Compute Virtualization includes **CPU virtualization**, **memory virtualization** and **I/O virtualization**, producing virtual memory, storage, network, etc.
- Virtualization basically allows one computer to do the job of multiple computers, by sharing the resources of a single hardware across multiple environments



In the past

One operating systems in one machine, so the OS had completely control of the resources in that machine.

Virtualization Era

Virtualization consists of a software layer in between the machine and the operating system. Essentially what this software layer does is to divide the resources of the machine among all the guest operating systems. The Virtualization Layer is in charge of multiplexing the hardware resources to several operating systems. Each OS has the illusion that it controls the complete hardware but, in fact, the machine can now host a number of operating systems because the virtualization layer makes all the switching behind scenes.

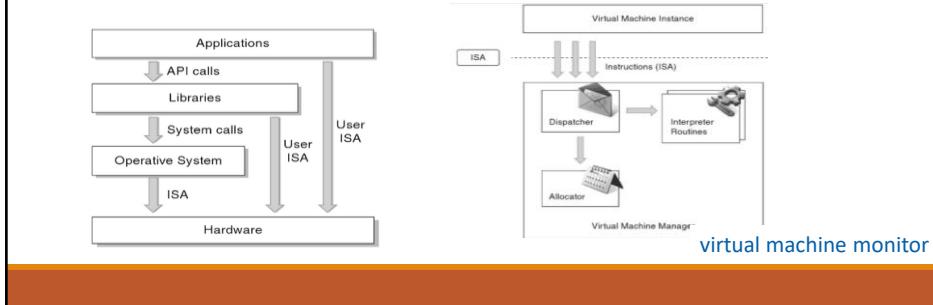
Supporting multiple instances of Operating Systems: Homogeneous or Heterogeneous

One physical machine can host several Linux/Window copies



Virtualization Requirements

- Virtual Machine Monitor
- A virtual machine monitor is a control program comprising:
 - A dispatcher
 - An allocator
 - A set of interpreters, one per privileged instruction.



ISA: Instruction Set Architecture. Model of the hardware that defines the instruction set of the processor. Essentially it consists of the machine language. Some instructions may be used by applications. Others are reserved for the OS.
ABI: application binary interface

Dispatcher: intercepts sensitive instructions executed by VMs and leaves control to the modules set up to manage the situation;

Allocator: provides virtual machines with the necessary resources avoiding conflicts (hardware resource management);

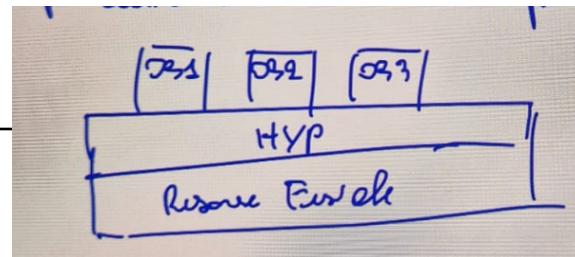
Interpreter: simulates instructions that refer to resources in ways that reflect their execution in the virtual machine environment (indispensable since VMs do not have direct access to physical resources).

Finally, the purpose of systems virtualization is to transparently share the hardware resources of a physical machine among the operating systems of multiple virtual machines, to make them execute as many instructions as possible directly on the processor without VMM interaction and to solve potential causes of malfunctions due to any lack of architecture virtualization requirements.

A VMM provides a *duplicate, or essentially identical* to the original machine, environment for programs. “Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies.”

It does so *efficiently*, requiring “a statistically dominant subset of the virtual processor’s instructions be executed directly by the real processor, with no software intervention by the VMM. This statement rules out traditional emulators and complete software interpreters (simulators) from the virtual machine umbrella.” Thus programs that run in this environment show only minor decreases in speed.

It is in complete control of system resources (memory, peripherals, and the like). This requires two conditions: (i) it must not be possible for a program running in the created environment to access any resource not allocated to it (*isolation*), and (ii) it is possible under certain circumstances to regain control of resources already allocated.



Type 1 hypervisors

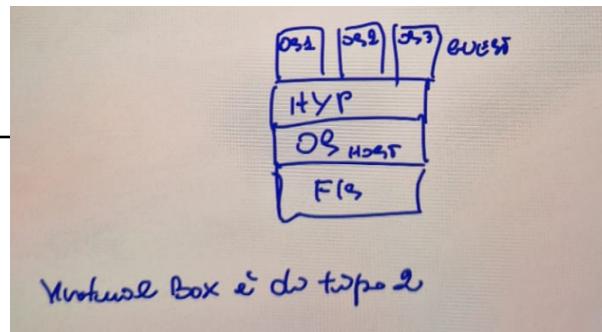
- Type 1 –hypervisor: bare metal, it has direct access to hardware resources and does not need to access the host OS. Kind of customized OS implementing VMM and does not run other applications.
- It responds to privileged CPU instructions or protection instructions sent by VMs, schedules VM queues, returns VMs the results of physical hardware processing.
- VMM creates and manages virtual environments.
- Pros: possible implementation of different guest OS types. Cons: the kernel of the virtualization layer is hard to develop.

Type 1 Virtual Machine Monitor (VMM) features

Two types of VMM can also be identified, based on the location of the physical machine in the environment:
1. Type 1 VMM is placed immediately above the hardware and has all the mechanisms of a normal kernel or operating system in terms of memory, peripheral and processor management; in addition, it implements the mechanisms for managing virtual machines. Virtual machines running above VMM have their own operating system and are called "guest" machines.



Type 2 hypervisors



- Called Hosted hypervisor. Physical resources are managed by host OS.
 - VMM provides virtualization as application. VMM obtains resources by calling host OS services. The VM, after creation, is scheduled by the VMM as a process of the host OS.
- Pros: Easy to implement. Cons: Only applications supported by the host OS can be installed and used. High performance overhead.

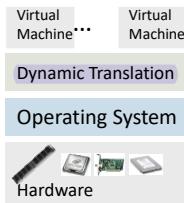
VMM type 2 is a normal process running under an operating system called "host". It directly manages the virtual machines, which are its sub-processes, while the management of the hardware is entrusted to the host system.



Evolution of Software solutions

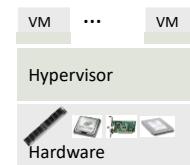
1st Generation: Full virtualization (Binary rewriting)

- Software Based
- QEMU, VMware and Microsoft



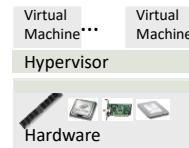
2nd Generation: Paravirtualization

- Cooperative virtualization
- Modified guest
- VMware, Xen



3rd Generation: Silicon-based (Hardware-assisted) virtualization

- Unmodified guest
- VMware and Xen on virtualization-aware hardware platforms



Server virtualization approaches



Virtualization Logic

Virtual Machine Monitor

Each virtual machine interfaces with its host system via the virtual machine monitor (VMM). Being the primary link between a VM and the host OS and hardware, the VMM provides a crucial role. The VMM primarily:

Presents emulated hardware to the virtual machine

Isolates VMs from the host OS and from each other

Throttles individual VM access to system resources, preventing an unstable VM from impacting system performance

Passes hardware instructions to and from the VM and the host OS/hypervisor

When full virtualization is employed, the VMM will present a complete set of emulated hardware to the VM's guest operating system. This includes the CPU, motherboard, memory, disk, disk controller, and network cards. For example, Microsoft Virtual Server 2005 emulates an Intel 21140 NIC card and Intel 440BX chipset. Regardless of the actual physical hardware on the host system, the emulated hardware remains the same.

The next significant role of the VMM is to provide isolation. The VMM has full control of the physical host system's resources, leaving individual virtual machines with access only to their emulated hardware resources. The VMM contains no mechanisms for inter-VM communication, thus requiring that two virtual machines wishing to exchange data do so over the network.

Another major role of the VMM is to manage host system resource access. This is important, as it can prevent over-utilization of one VM from starving out the performance of other VMs on the same host. Through the system configuration console, system hardware resources such as the CPU, network, and disk access can be throttled, with maximum usage percentages assigned to each individual VM. This allows the VMM to properly schedule access to host system resources as well as to guarantee that critical VMs will have access to the amount of hardware resources they need to sustain their operations.

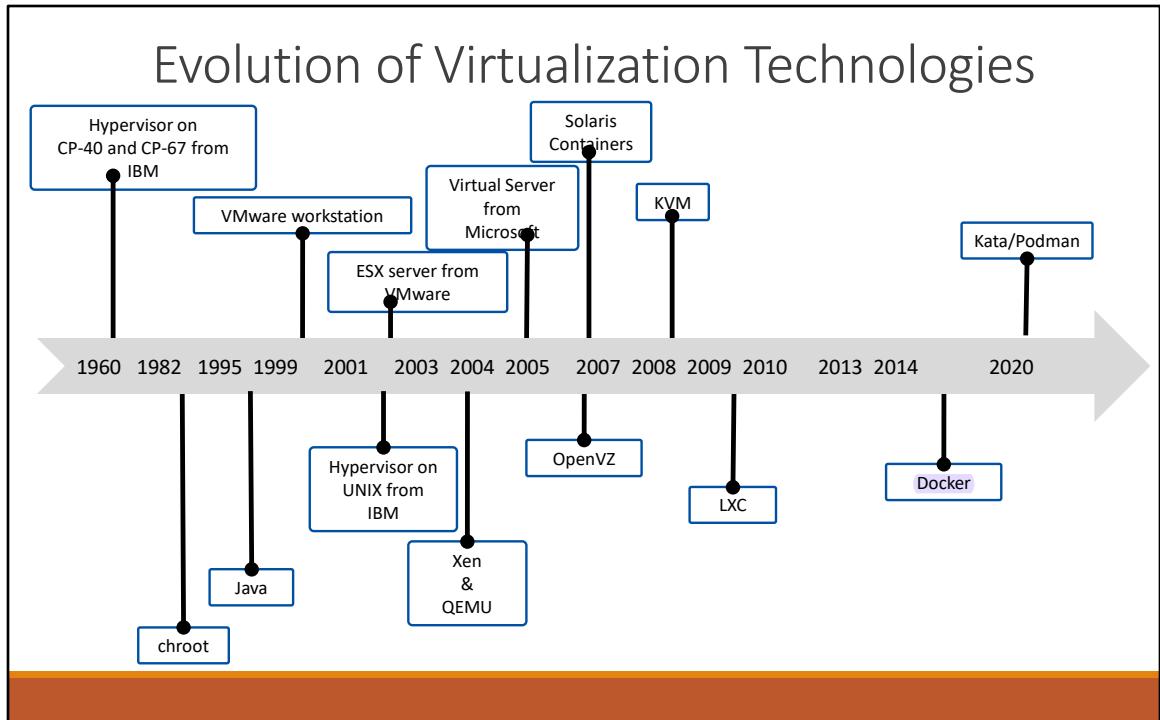
Host OS/Hypervisor

The primary role of the host operating system or hypervisor is to work with the VMM to coordinate access to the physical host system's hardware resources. This includes scheduling access to the CPU as well as the drivers for communication with the physical devices on the host, such as its network cards.

The term hypervisor is used to describe a lightweight operating shell that has the sole purpose of providing VM hosting services. The hypervisor differs from a traditional OS in that the OS may be designed for other roles on the network. As it is tailored to VM hosting, a hypervisor solution generally offers better performance and should have fewer security vulnerabilities because it runs few services and contains only essential code. Hypervisors written for hardware-assisted virtualization can embed themselves much deeper into the system architecture and offer superior performance improvements as a result.

Like any traditional OS, a hypervisor-based OS still contains its own operating system code; therefore, maintaining security updates is still important. Unlike a traditional OS, hypervisors are vendor specific, so any needed hypervisor patches or security updates will come directly from the virtualization software vendor. Because hypervisors are vendor-centric, individual device support often comes directly from the virtualization vendors. Hence, it is important for the organization to ensure that any planned virtualization products are compatible with its existing or planned system hardware. When hosting VMs on a traditional OS such as SUSE Linux Enterprise Server or Windows Server "Longhorn," the organization will find that while the host OS has a larger footprint than a hypervisor, it does provide additional flexibility with hardware devices. With SAN integration, for example, if the host OS does not recognize a Fibre Channel host bus adapter (HBA), the administrator can download the appropriate driver from the vendor's website. With a hypervisor, the administrator will need to get the driver from the virtualization software vendor, or learn that the device is not supported.

Both hypervisors and operating systems have their strengths and weaknesses. Operating systems provide greater device support than hypervisors, but also require attention to ensure that they are current on all patches and security updates. Hypervisors run on minimal disk and storage resources, but patches and device drivers must come directly from the virtualization software vendor.



Server virtualization started back in the early 1960's and was pioneered by companies like General Electric (GE), Bell Labs, and International Business Machines (IBM), to run legacy software on newer mainframe hardware, and to support newer mainframe hardware capable of more than one simultaneous user. The CP-67 system was the first commercial mainframe to support virtualization. The CP approach to time sharing allowed each user to have their own complete operating system which effectively gave each user their own computer. The main advantages of using virtual machines vs a time sharing operating system was more efficient use of the system since virtual machines were able to share the overall resources of the mainframe, instead of having the resources split equally between all users. There was better security since each user was running in a completely separate operating system, and it was more reliable since no one user could crash the entire system; only their own operating system.

By the late 1980's *software emulators* were developed to allow users to run DOS and Windows applications on their Unix and Mac workstations. In 1999 VMware developed virtualization technology that *simulated enough hardware* to allow an unmodified guest OS to be run in isolation. In 2001 VMware released ESX Server that does not require a host operating system to run virtual machines. This is known as a *type 1 hypervisor*. That year they also released GSX Server that allowed users to run virtual machines on top of an existing operating system, such as Microsoft Windows. This is known as a *type 2 hypervisor*. Xen was the first open-source x86 hypervisor, released in 2003 by researchers at Cambridge University. The Xen hypervisor is a small, lightweight type 1 hypervisor derived from work done on the Linux kernel. The Xen hypervisor can run unmodified fully virtualized guests, or paravirtualized guests that use a special API to communicate with the hypervisor. KVM (Kernel-based Virtual Machine) was merged into the Linux kernel mainline in February 2007. It is a virtualization infrastructure for the Linux kernel that turns it into a type 2 hypervisor.

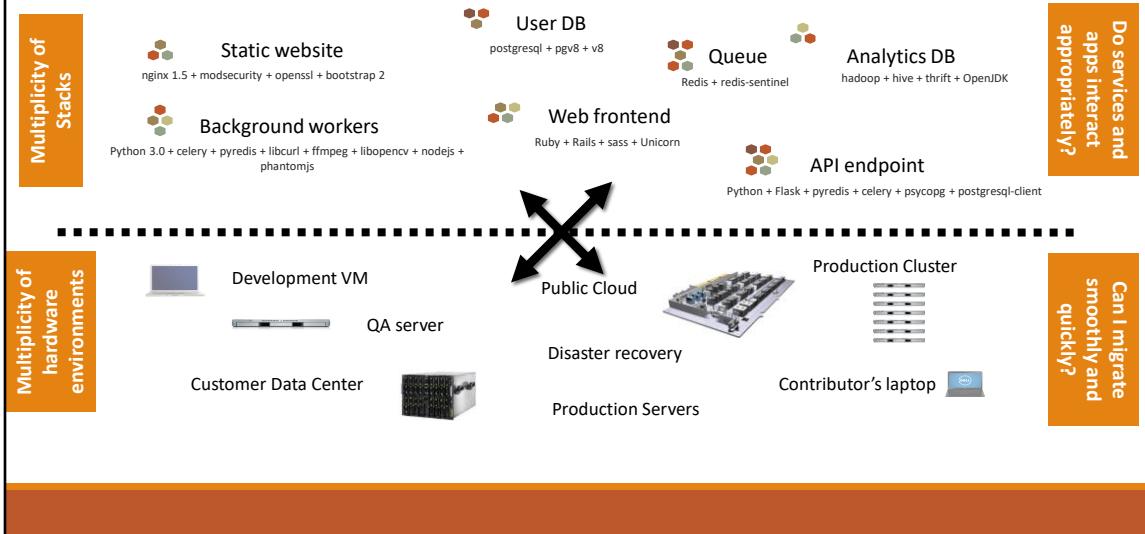


Containers – Introduction

- Containers virtualize the OS just like hypervisors virtualizes the hardware
- Containers enable any payload to be **encapsulated** as a lightweight, portable self-sufficient container, that can be manipulated using standard operations and run consistently on any hardware platform.
- **Wraps up** a piece of software **in a complete filesystem** that contains everything it needs to run such as : code, runtime, system tools, libraries etc., **they share the OS kernel** and bins/libs where needed, otherwise each of them operate in a self contained environment.



The Challenge



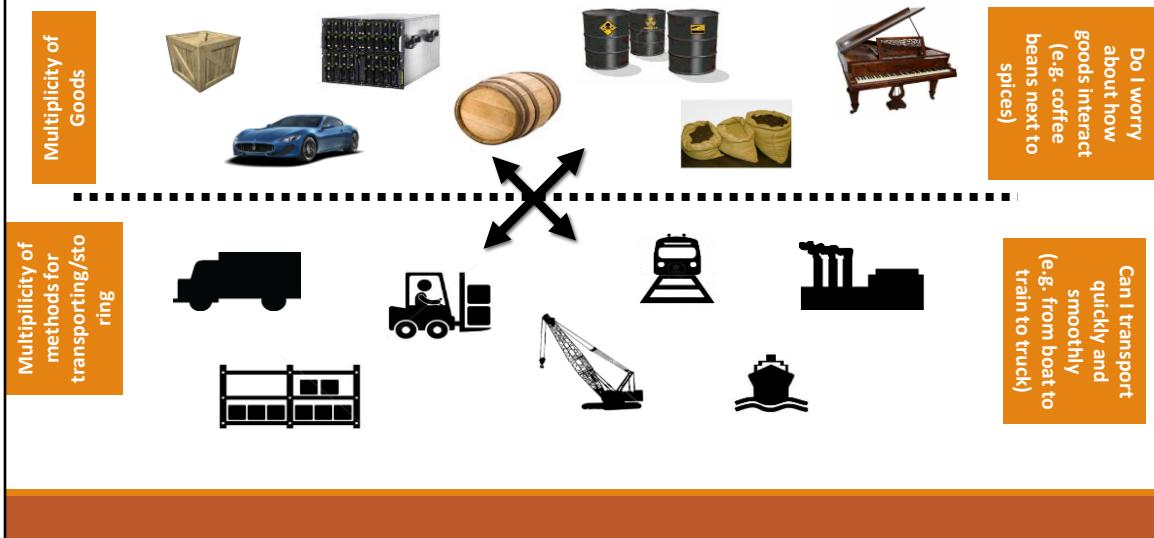


The Matrix From Hell

Static website	?	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?	?
	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers	



Cargo Transport Pre-1960





Also a matrix from hell

	?	?	?	?	?	?	?	?
	?	?	?	?	?	?	?	?
	?	?	?	?	?	?	?	?
	?	?	?	?	?	?	?	?
	?	?	?	?	?	?	?	?
	?	?	?	?	?	?	?	?

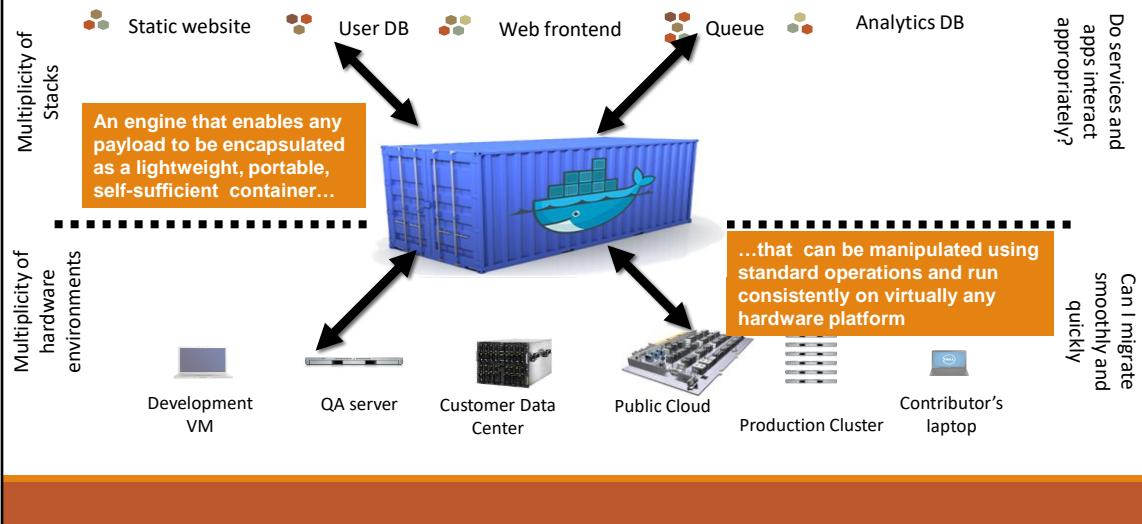


Solution: Intermodal Shipping Container





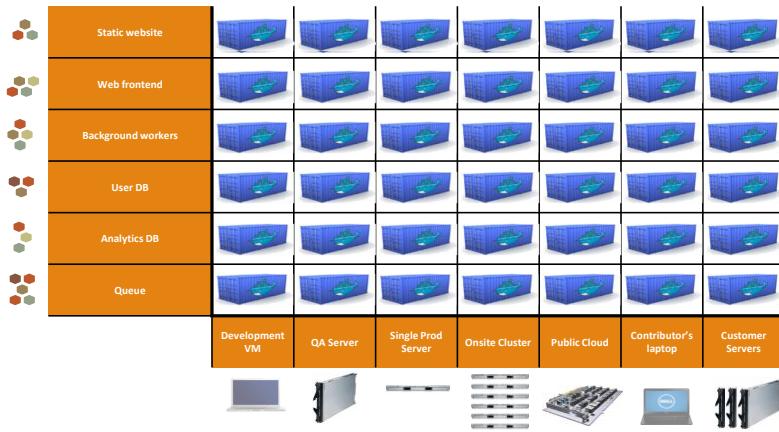
Docker is a shipping container system for code



A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker eliminates the matrix from Hell

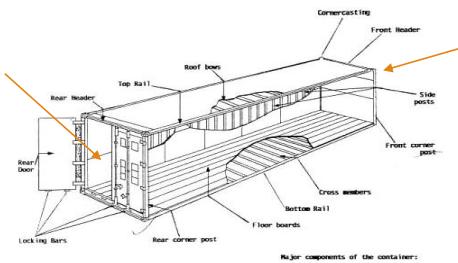




Why it works—separation of concerns

Dan the Developer

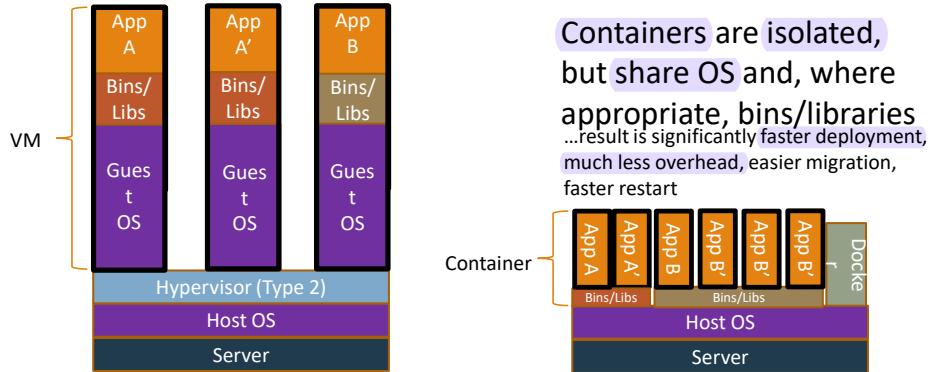
- Worries about what's "inside" the container
 - His code
 - His Libraries
 - His Package Manager
 - His Apps
 - His Data
- All Linux servers look the same



Oscar the Ops Guy

- Worries about what's "outside" the container
 - Logging
 - Remote access
 - Monitoring
 - Network config
- All containers start, stop, copy, attach, migrate, etc. the same way

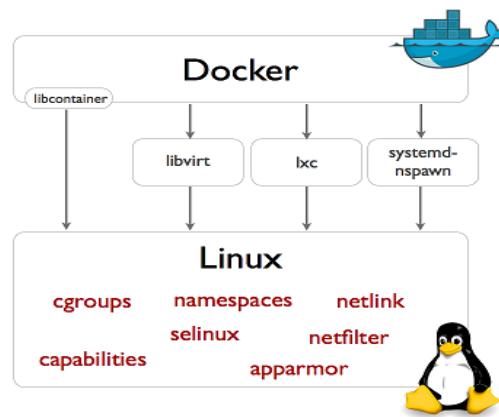
Containers vs. VMs





Virtualization Technologies

Docker – Underlying Technology

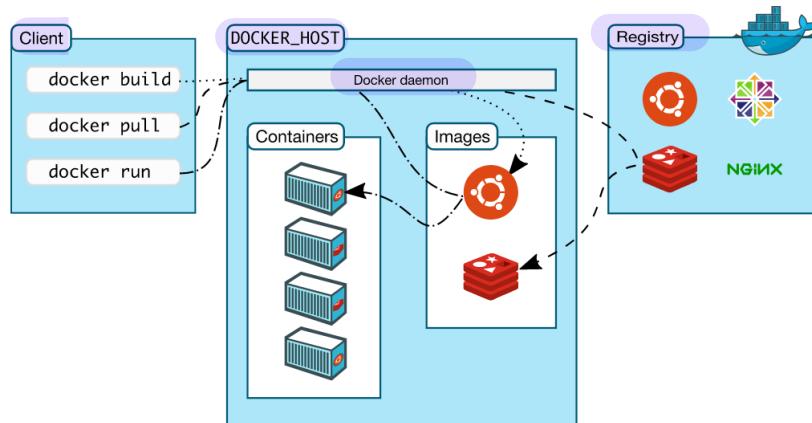


Docker makes use of several Linux kernel features to deliver the functionality.

Linux kernel namespaces essentially isolate what the application can see from the operating environment, including the process tree, network, user IDs and mounted file systems, while cgroups provide isolation of resources, including CPU, memory, block I / O devices and the network. Docker includes the libcontainer library to directly use the virtualization features of the Linux kernel, in addition to abstract virtualization interfaces such as libvirt, LXC and systemd-nspawn.



Docker architecture



Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



Docker architecture

The Docker daemon: The Docker daemon (`dockerd`) listens for Docker API requests and manages Docker objects such as **images, containers, networks, and volumes**. A daemon can also communicate with other daemons to manage Docker services.

The Docker client: The Docker client (`docker`) is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to `dockerd`, which carries them out. The `docker` command uses the Docker API. The Docker client can communicate with more than one daemon.



Docker architecture

Docker registries A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

- When you use the `docker pull` or `docker run` commands, the required images are pulled from your configured registry. When you use the `docker push` command, your image is pushed to your configured registry.

Docker Desktop is an easy-to-install application for your Mac or Windows environment that enables you to build and share containerized applications and microservices.

- Docker Desktop includes the Docker daemon (`dockerd`), the Docker client (`docker`), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

the Docker Dashboard, which gives you a quick view of the containers running on your machine. The Docker Dashboard is available for Mac and Windows. It gives you quick access to container logs, lets you get a shell inside the container, and lets you easily manage container lifecycle (stop, remove, etc.).



Docker objects

When you use Docker, you are creating and using **images**, **containers**, **networks**, **volumes**, and other objects.

Images An *image* is 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, as well as the configuration details needed to make your application run.

Containers A container is 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.

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. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

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.



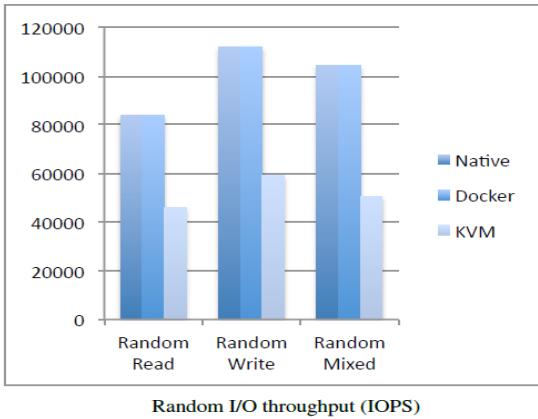
Example: docker run

```
$ docker run -i -t ubuntu /bin/bash
```

- If you do not have the **ubuntu image** locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.
- Docker creates a **new container**, as though you had run a docker container create command manually.
- Docker allocates a **read-write filesystem** to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
- Docker creates a **network interface** to connect the container to the default network, since you did not specify any networking options. This includes **assigning an IP address to the container**. By default, containers can connect to external networks using the host machine's network connection.
- Docker starts the container and executes **/bin/bash**. Because the container is running interactively and attached to your terminal (due to the **-i** and **-t** flags), you can provide input using your keyboard while the output is logged to your terminal.
- When you type exit to terminate the **/bin/bash command**, the container stops but is not removed. You can start it again or remove it.

When you run this command, the following happens (assuming you are using the default registry configuration):

I/O Performance



IBM Research Report July, 2014

IOV method	throughput (Mb/s)	CPU utilization
bare-metal	950	20%
device assignment	950	25%
paravirtual	950	50%
emulation	250	100%

- netperf TCP_STREAM sender on 1Gb/s Ethernet (16K msgs)
- Device assignment best performing option
- Device assignment still 25% worse than bare metal.

IBM®System x3650 M4 server
two 2.4-3.0 GHz Intel Sandy Bridge-EP Xeon E5-2665 processors
16 cores (plus HyperThreading)
256 GB of RAM

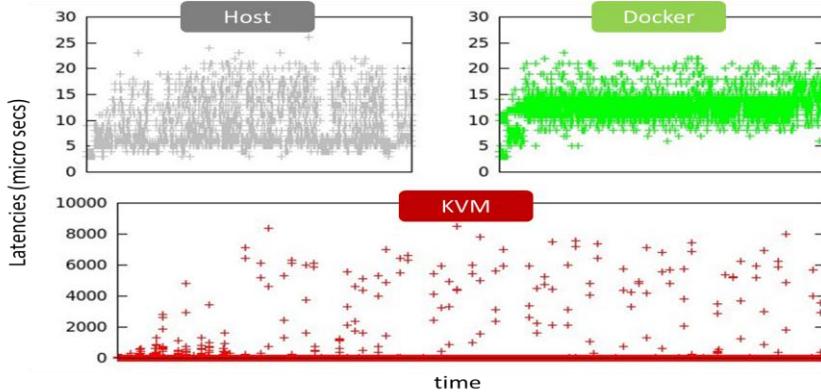
SAN-like block storage is commonly used in the cloud to provide high performance and strong consistency.

The Random I/O throughput benchmarks are slower on KVM because each I/O operation must go through QEMU (emulation).

KVM network paravirtualization does not have to go through an emulated NIC, but the paravirtualization results in an extreme amount of context switching, and interrupt processing still goes through the hypervisor.

Real-time Latency

Cyclictest



Intel Ivy bridge based 4 core with hyper-threading (8 logical cores) each running @ 2.2 GHz.
8 GB RAM

Latency is defined as the time interval between the occurrence of an event and the time when that event is "handled" (usually running a thread).

Examples:

- The time between when an interrupt occurs and the thread waiting for that interrupt is run
- The time between a timer expiration and the thread waiting for that timer to run
- The time between the receipt of a network packet and when the thread waiting for that packet runs

Cyclictest is one of the widely used tool for RT performance benchmarking. It measures the latency of response to a stimulus.

From the benchmarking figure:

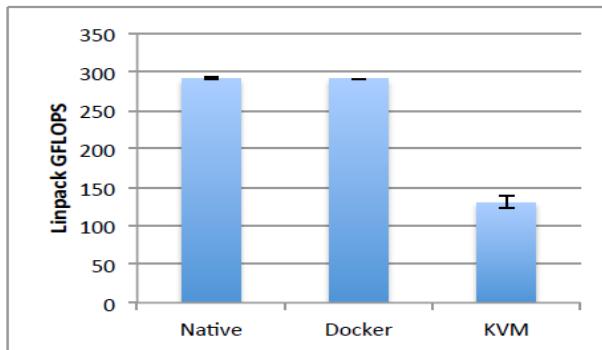
Host – Latencies under 25 micro seconds

Docker – Same as the host

KVM – Latencies as high as 9000 micro seconds



Math Performance

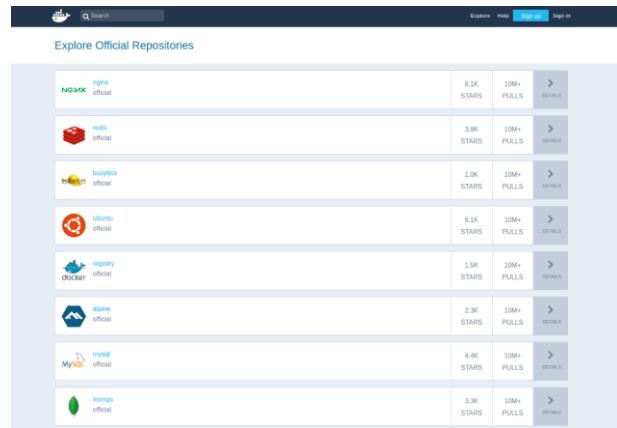


IBM Research Report July, 2014

IBM®System x3650 M4 server
two 2.4-3.0 GHz Intel Sandy Bridge-EP Xeon E5-2665 processors
16 cores (plus HyperThreading)
256 GB of RAM

Mathematics can be much slower with KVM because math libraries cannot tune themselves as well with the system-provided information that KVM offers. By being unable to detect the exact nature of the system, the execution employs a more general algorithm with consequent performance penalties.

Docker Hub



Docker Hub is the world's **largest repository of container images** with an array of content sources including container community developers, open source projects and independent software vendors (ISV) building and distributing their code in containers. Users get access to free public repositories for storing and sharing images or can choose subscription plan for private repos.



Dockerfile

It is possible to build automatically your own images reading instructions from a Dockerfile

```
FROM centos:7
RUN yum install -y python-devel python-virtualenv
RUN virtualenv /opt/indico/venv
RUN pip install indico
COPY entrypoint.sh /opt/indico/entrypoint.sh
EXPOSE 8000
ENTRYPOINT /opt/indico/entrypoint.sh
```

Containers are designed for running specific tasks and processes, not for hosting operating systems. You create a container to serve a single unit task. Once it completes the given task, it stops. Therefore, the container life-cycle depends on the ongoing process inside of it. Once the process stops, the container stops as well.

A Dockerfile defines this process. It is a script made up of instructions on how to build a Docker image.

The Dockerfile is a text file that contains the instructions needed to create a new container image. These instructions may include identifying an existing image to use as a base, commands to run during the image creation process, and a command that will be run when deploying new instances of the container image.

Docker build is the Docker engine command that uses a Dockerfile and triggers the image creation process.



Dockerfile components

The **FROM** statement sets the container image that will be used during the process of creating a new image.

For example, when you use the statement

◦ **FROM mcr.microsoft.com/windows/servercore**

- the resulting image derives and has a dependency on the Windows Server Core base operating system image.
- If the specified image is not present on the system where the Docker build process is running, the Docker engine will attempt to download the image from a public or private image registry.

<https://docs.microsoft.com/it-it/virtualization/windowscontainers/manage-docker/manage-windows-dockerfile>

<https://docs.docker.com/engine/reference/builder/>



Dockerfile components

Environment variables are supported by the following [list of instructions](#) in the Dockerfile:

- ADD, COPY, ENV, EXPOSE, FROM, LABEL, STOPSIGNAL, USER, VOLUME, WORKDIR, ONBUILD

The **COPY** instruction copies files and directories to the file system container. The files and directories must be in a path relative to the Dockerfile.

- **COPY ["<source>", "<destination>"]**

Linux format: COPY test1.txt c:\temp\

Windows format: COPY test1.txt c:/temp/



Dockerfile components

The **RUN** instruction specifies the commands to execute and capture in the new container image.

These commands can include items, including installing software, creating files and directories, and creating the environment configuration. Two forms exist:

- exec form

```
RUN ["<executable>", "<param 1>", "<param 2>"]
```

- shell form

```
RUN <command>
```

the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows

La differenza tra il formato exec e il formato della shell è la modalità di RUN esecuzione dell'istruzione. Quando si usa il formato exec, il programma specificato viene eseguito in modo esplicito.

Di seguito è riportato un esempio del modulo exec:

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019
RUN ["powershell", "New-Item", "c:/test"]
```

L'immagine risultante esegue il comando: powershell New-Item c:/test

Al contrario, l'esempio seguente esegue la stessa operazione in formato shell:

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019
RUN powershell New-Item c:\test
```

L'immagine risultante ha un'istruzione di esecuzione di cmd /S /C powershell New-Item c:\test .



Dockerfile components

The **EXPOSE** instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified. By default, EXPOSE assumes TCP. You can also specify UDP:

- EXPOSE 80/tcp
- EXPOSE 80/udp



Dockerfile components

There are **two types of instructions** that can define the process running in the container:

- **CMD**

- CMD defines default commands and/or parameters for a container.
CMD is an instruction that is best to use if you need a default command which users can easily override.

- **ENTRYPOINT**

- ENTRYPPOINT is preferred when you want to define a container with a specific executable.

<https://phoenixnap.com/kb/docker-cmd-vs-entrypoint>

Docker CMD defines the default executable of a Docker image. You can run this image as the base of a container without adding command-line arguments. In that case, the container runs the process specified by the CMD command.

The CMD instruction is only utilized if there is no argument added to the **run** command when starting a container. Therefore, if you add an argument to the command, you override the CMD.

ENTRYPOINT is the other instruction used to configure how the container will run. Just like with CMD, you need to specify a command and parameters.

Assume to launch the command `sudo docker run [image_name] [parameter]`

When there is no command-line argument (the parameter), the container will run the default CMD instruction and executes the CMD instruction. However, if you add an argument when starting a container, it overrides the CMD instruction.

In case of ENTRYPPOINT, Docker does not override the initial instruction. It merely added the new parameter to the existing command.



Dockerfile components

The **CMD** and **ENTRYPOINT** instruction allows you to configure a container that will run as an executable. Two forms exist:

- exec form, which is the preferred form:

```
CMD ["executable", "param1", "param2"]
```

```
ENTRYPOINT ["executable", "param1", "param2"]
```

- shell form

```
CMD command param1 param2
```

```
ENTRYPOINT command param1 param2
```

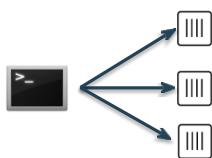
the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows



Docker Compose: Multi Container Applications

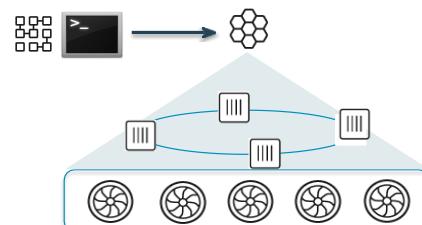
naive

- Build and run one container at a time
- Manually connect containers together
- Must be careful with dependencies and start up order



compose

- Define multi container app in compose.yml file
- Single command to deploy entire app
- Handles container dependencies
- Works with Docker Swarm, Networking, Volumes, Universal Control Plane





Docker Compose: Multi Container Applications

```
version: '2' # specify docker-compose version
# Define the services/containers to be run services:

angular: # name of the first service
build: client # specify the directory of the Dockerfile ports:
- "4200:4200" # specify port forwarding

- express: #name of the second service
build: api # specify the directory of the Dockerfile ports:
- "3977:3977" #specify ports forwarding

database: # name of the third service
image: mongo # specify image to build container from ports:
- "27017:27017" # specify port forwarding
```



Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Using Compose is basically a three-step process:

- 1.Define your app's environment with a Dockerfile so it can be reproduced anywhere.
- 2.Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
- 3.Run docker compose up and the [Docker compose command](#) starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.

A *db* service is defined. It will be brought up and shut down whenever we run docker-compose up/down

You can to specify the alternate path to the Dockerfile in the build section of the Docker Compose file.



Docker Compose: Multi Container Applications

It allows running multi-container Docker applications reading instructions from a docker-compose.yml file

```
version: "2"
services:
  my-application:
    build: .
    ports:
      - "8000:8000"
    environment:
      - CONFIG_FILE
  db:
    image: postgres
  redis:
    image: redis
    command: redis-server --save "" --appendonly no
    ports:
      - "6379"
```

Docker Compose provides a way to orchestrate multiple containers that work together.

Examples include a service that processes requests and a front-end web site, or a service that uses a supporting function such as a Redis cache. If you are using the microservices model for your app development, you can use Docker Compose to factor the app code into several independently running services that communicate using web requests.

CONFIG_FILE includes environment variables.

--appendonly no start redis in no persistent storage mode

Redis provides a different range of persistence options:

- **RDB (Redis Database):** The RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
- **AOF (Append Only File):** The AOF persistence logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset. Commands are logged using the same format as the Redis protocol itself, in an append-only fashion. Redis is able to [rewrite](#) the log in the background when it gets too big.
- **No persistence:** If you wish, you can disable persistence completely, if you want your data to just exist as long as the server is running.
- **RDB + AOF:** It is possible to combine both AOF and RDB in the same instance. Notice that, in this case, when Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

Basic Docker Commands

```
$ docker image pull node:latest
$ docker image ls
$ docker container run -d -p 5000:5000 --name node node:latest
$ docker container ps
$ docker container stop node(or <container id>)
$ docker container rm node (or <container id>)
$ docker image rmi (or <image id>)
$ docker build -t node:2.0
$ docker image push node:2.0
$ docker --help
```



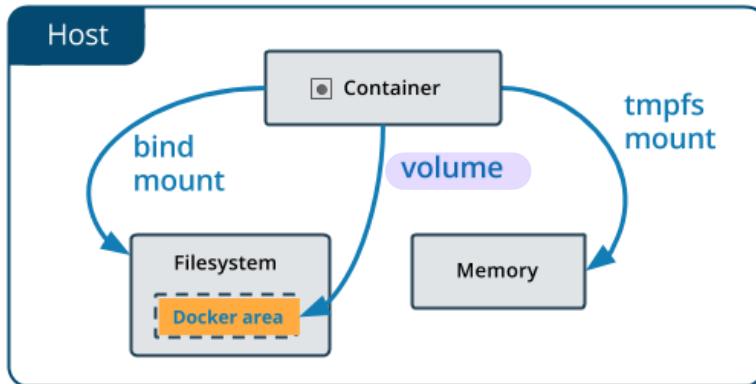
Docker Volumes

- Volumes mount a directory on the host into the container at a specific location
- Can be used to share (and persist) data between containers
 - Directory persists after the container is deleted
 - Unless you explicitly delete it
- Can be created in a Dockerfile or via CLI

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.



Why Use Volumes



Bind mounts have been around since the early days of Docker. Bind mounts have limited functionality compared to [volumes](#). When you use a bind mount, a file or directory on the *host machine* is mounted into a container. The file or directory is referenced by its absolute path on the host machine. By contrast, when you [use a volume](#), a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents.

The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available. If you are developing new Docker applications, consider using [named volumes](#) instead. You can't use Docker CLI commands to directly manage bind mounts.

While [bind mounts](#) are dependent on the directory structure and OS of the host machine, [volumes](#) are completely managed by Docker. Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

In addition, volumes are often a better choice than persisting data in a container's writable layer, because a volume does not increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container.

If your container generates non-persistent state data, consider using a [tmpfs mount](#) to avoid storing the data anywhere permanently, and to increase the container's performance by avoiding writing into the container's writable layer.



Docker Networking overview

- One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads.
- Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not.
- Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.
- Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality



Docker Network Drivers

- **bridge:** The default network driver. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
- **host:** For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly.
- **overlay:** Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.
- **ipvlan:** IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration.
- .

- **bridge:** The default network driver. If you don't specify a driver, this is the type of network you are creating. **Bridge networks are usually used when your applications run in standalone containers that need to communicate.** See [bridge networks](#).
- **host:** For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. See [use the host network](#).
- **overlay:** Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See [overlay networks](#).
- **ipvlan:** IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration. See [IPvlan networks](#).
- **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
- **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- **Third-party network plugins** allow you to integrate Docker with specialized network stacks.



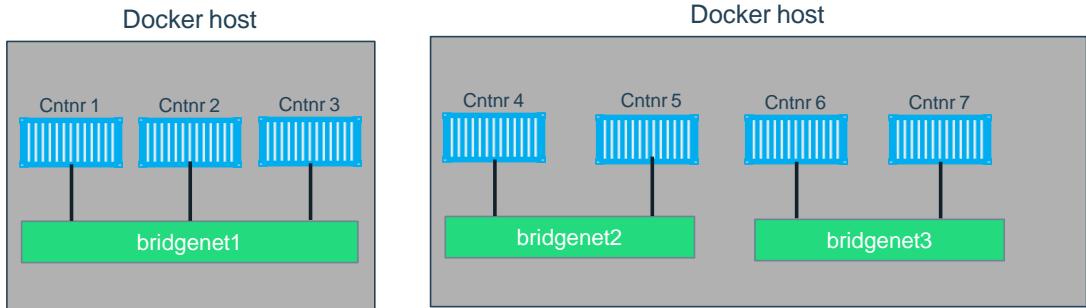
Docker Network Drivers

- **macvlan**: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.
- **none**: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services.
- **Network plugins**: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

- **macvlan**: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See [Macvlan networks](#).
- **none**: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See [disable container networking](#).
- **Network plugins**: You can install and use third-party network plugins with Docker. These plugins are available from [Docker Hub](#) or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.



What is Docker Bridge Networking

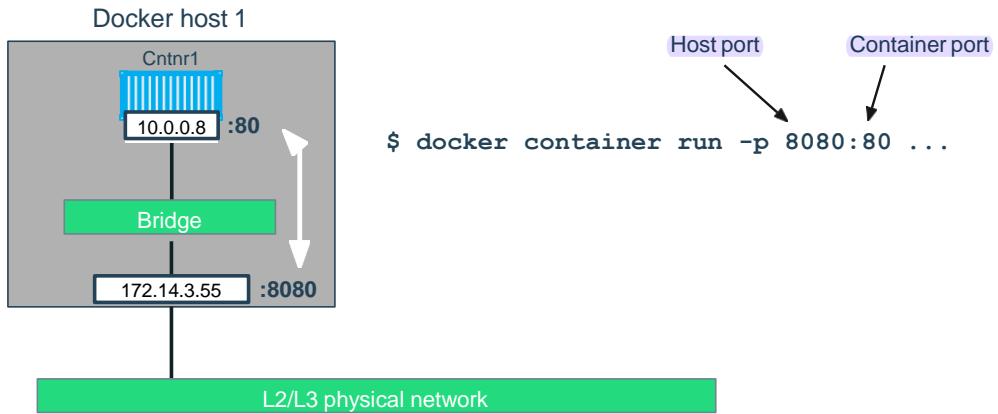


```
docker network create -d bridge --name bridgenet1
```

You can specify the subnet, the IP address range, the gateway, and other options.



Docker Bridge Networking and Port Mapping



When you create or remove a user-defined bridge or connect or disconnect a container from a user-defined bridge, Docker uses tools specific to the operating system to manage the underlying network infrastructure (such as adding or removing bridge devices or configuring iptables rules on Linux). These details should be considered implementation details. Let Docker manage your user-defined networks for you.



Container Orchestration

- In order to scale containers, you need a **container orchestration tool**—a framework for managing multiple containers.
- Today, the most prominent container orchestration platforms are [Docker Swarm](#) and [Kubernetes](#). They both come with **advantages** and **disadvantages**, and they both serve a particular purpose.
- [Docker Swarm](#) is an **open-source container orchestration platform** built and maintained by Docker.
- [Kubernetes](#) is an **open source container orchestration platform** that was initially designed by Google to manage their containers.

Docker Swarm is straightforward to install, especially for those just jumping into the container orchestration world. It is lightweight and easy to use. Also, Docker Swarm takes less time to understand than more complex orchestration tools. It provides automated load balancing within the Docker containers, whereas other container orchestration tools require manual efforts.

Kubernetes has a **more complex cluster structure** than Docker Swarm.

It usually has a builder and worker nodes architecture divided further into pods, namespaces, config maps, and so on.



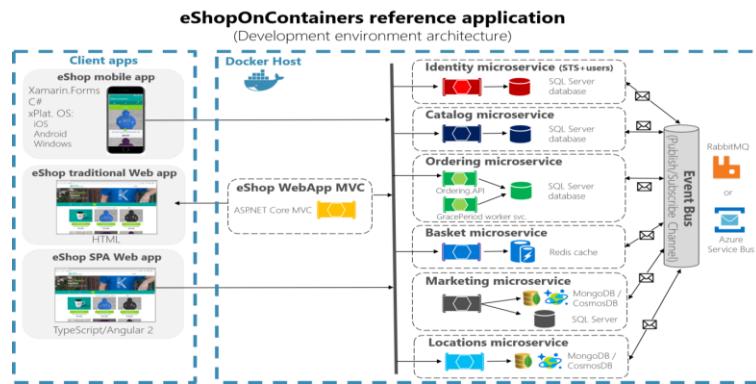
Container Orchestration

Kubernetes advantages.

- It has a large open-source community, and Google backs it.
- It supports every operating system.
- It can sustain and manage large architectures and complex workloads.
- It is automated and has a self-healing capacity that supports automatic scaling.
- It has built-in monitoring and a wide range of available integrations.
- It is offered by all three key cloud providers: Google, Azure, and AWS.



An example of microservice-based system



<https://blogs.msdn.microsoft.com/dotnet/2017/08/02/microservices-and-docker-containers-architecture-patterns-and-development-guidance/>



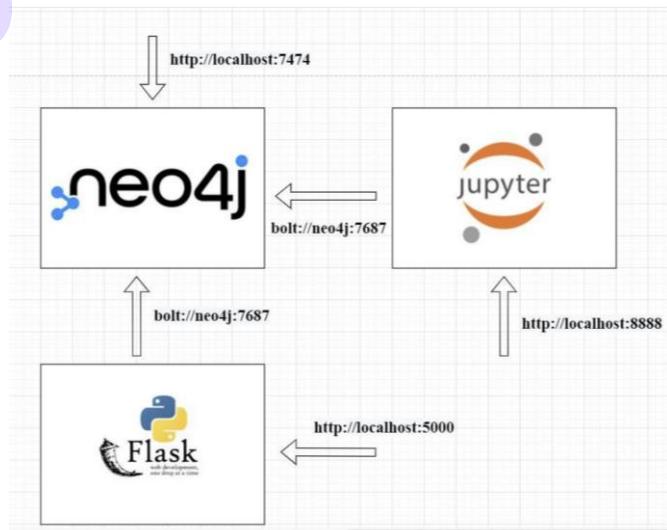
Example: multi-container application

Docker

- **Flask:** micro-framework written in Python for the development of web applications, according to the **Web Server Gateway Interface (WSGI)**, pronounced *whiskey* or *WIZ-ghee*) protocol for web servers to forward requests to web applications, written in the Python.
- **Neo4j:** graph database software in which information is stored in the form of nodes, relationships and properties. The Cypher query language is used to interact with this database
- **Jupyter Notebook:** interactive computing environment supporting different programming languages. In this case, it is used for illustrative purposes to interact with Neo4j and enter data into the database, through instructions written in Python.

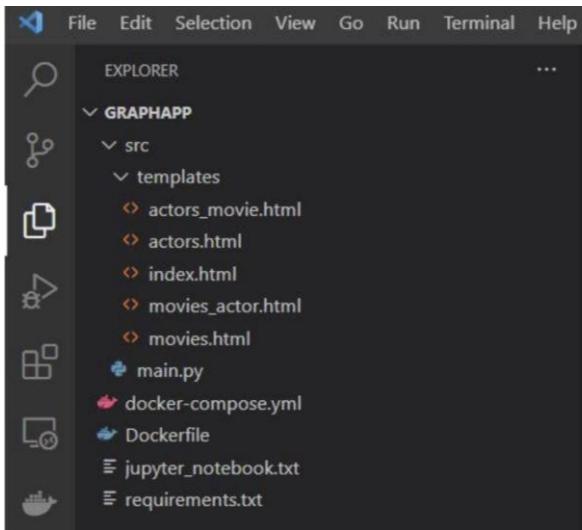


Architecture





Front-end



- Within the 'templates' directory we find the 'index.html' file, which represents the front-end of the application that manages the interaction with the user,
- the other files with the same extension are used to return results, from the server to the user, resulting from the possible actions that the user can perform through the interface accessible at the endpoint: `http://localhost: 5000.`



Server-side

The 'main.py' file is the Python script that represents the server side of the application, which interacts with Neo4j through a special driver and manages requests coming from the client.

It is implemented by using Flask.



The Dockerfile

```
Dockerfile X
Dockerfile > ...
1  FROM python:3.9
2
3  WORKDIR /GraphApp
4
5  COPY src/main.py .
6
7  COPY src/templates/index.html ./templates/index.html
8
9  COPY src/templates/actors.html ./templates/actors.html
10
11 COPY src/templates/movies.html ./templates/movies.html
12
13 COPY src/templates/movies_actor.html ./templates/movies_actor.html
14
15 COPY src/templates/actors_movie.html ./templates/actors_movie.html
16
17 COPY requirements.txt requirements.txt
18
19 RUN pip install -r requirements.txt
20
```

The 'Dockerfile' contains the commands for the creation of the image that includes the front-end and the back-end of the application, without the database, which is instantiated in another container.

- The Python image is downloaded from the Docker Hub, on which the new image.
- The 'GraphApp' directory is created in the container and set as working directory for the instructions below.
- The 'main.py' file, the 'templates' directory and the related files mentioned above, it is also copied the 'requirements.txt' file which contains the libraries necessary for the operation of 'main.py'
- The command "pip install" which installs the libraries specified in the 'requirements.txt' file.



The Docker Compose

```
File Edit Selection View Go Run Terminal Help
dockerc-compose.yml ×
dockerc-compose.yml
version: '3.3'
services:
  app:
    build: .
    container_name: graphapp
    restart: always
    depends_on:
      - neo4j
    ports:
      - "5000:5000"
    command: python main.py
    networks:
      - graph_network
neo4j:
  image: neo4j:4.3.7-enterprise
  container_name: neo4j
  restart: always
  ports:
    - "7474:7474"
    - "7687:7687"
  volumes:
    - ./data:/data
  environment:
    - NEO4J_ACCEPT_LICENSE AGREEMENT=yes
    - NEO4J_AUTH=neo4j/pierluigi
  networks:
    - graph_network
jupyter:
  image: jupyter/minimal-notebook:latest
  container_name: jupyter
  restart: always
  depends_on:
    - neo4j
  ports:
    - "8888:8888"
  networks:
    - graph_network
networks:
  graph_network:
    name: graph_network
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
```

To allow the containers to communicate with each other, a network is created virtual internal and it is possible to note, precisely, the presence of 'networks' in the definition of each service mentioned above.

- The first service refers to the 'Flask' application. With 'build' creates the image using the Dockerfile in the current directory. The dependence from the 'neo4j' service means that this container is started after the container containing the database.
- 'ports' shows the mapping between the external port of the host and the internal one of the container.
- The second service refers to Neo4j, whose image is downloaded from the Docker Hub. The first port is for the 'http' protocol and the second for the 'bolt' protocol, used from the Python driver for communicating with the database. Since this service is created, with 'volumes', a volume in such a way as to store data persistently. Furthermore, some environment variables are included.
- The third service refers to the Jupyter Notebook. The image is downloaded from the Docker Hub.



Jupyter notebook

The 'jupyter_notebook.txt' file contains the Python instructions that are executed inside the Jupyter Notebook in order to upload the data inside Neo4j.

docker compose up/down



Kubernetes: An Introduction

Gianluca Reali

Summary:

- What is Kubernetes
- Basic Objects
- Kubernetes Control Plane
- Kubernetes Networking Model
- Container-to-Container Networking
- Pod-to-Pod Networking
- Pod-to-Service Networking
- Internet-to-Service Networking
- The CNI
- Kubernetes CNI Plugins
- Volumes
- Demo

What is Kubernetes

- Open source and Production-Grade container orchestration platform
- Google spawns billions of containers per week with these systems
- Based on Borg and Omega
- Google donated the Kubernetes project to the newly formed Cloud Native Computing Foundation (CNCF) in 2015.



"κυβερνήτης"
(kubernetes) is Greek
for "pilot" or
"helmsman of ship"

What is Kubernetes

With Kubernetes you can:

Orchestrate containers across multiple hosts.

Control and automate application deployments and updates

Mount and add storage to run stateful apps

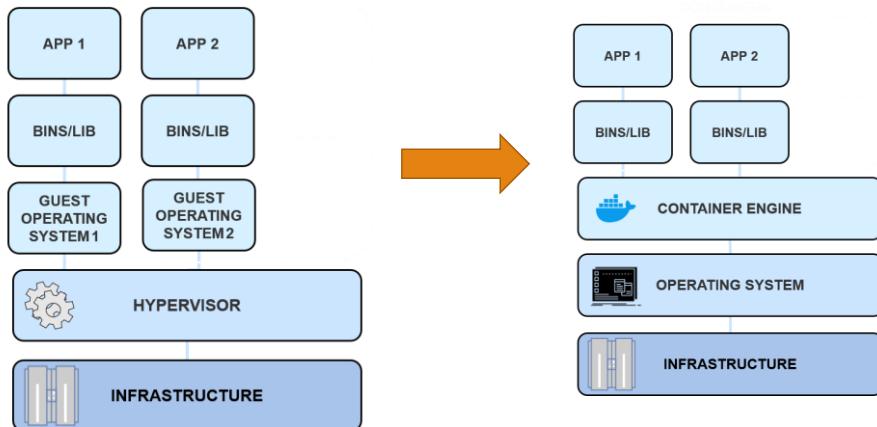
Scale containerized applications and their resources on the fly

- All services within Kubernetes are natively Load Balanced.
- Can scale up and down dynamically.

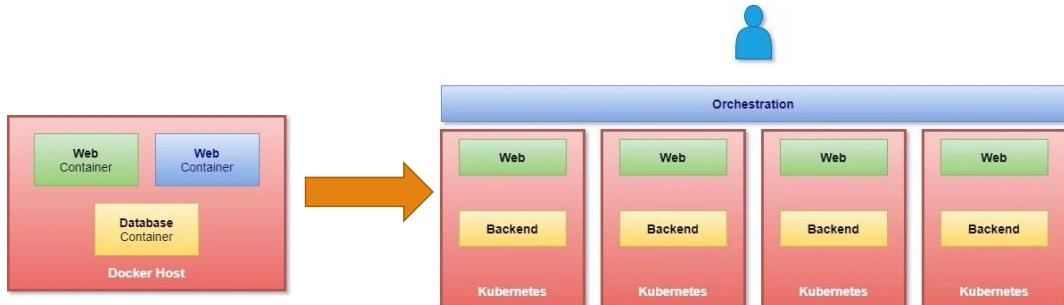
Declaratively manage services (applications are always running the way you intended them to run)

Health-check and self-heal your apps with autoposition, autorestart, autoreplication, and autoscaling

... what is a container?



Container Orchestration



- Your application is now **highly available** as now we have multiple instances of your application across multiple nodes
- The user traffic is load balanced across various containers
- When demand increases deploy more instances of the applications seamlessly and within a matter of seconds and we have the ability to do that at a service level when we run out of hardware resources then **scale the number of underlying nodes up and down** without taking down the application and this all can be done easily using a set of declarative object configuration file.

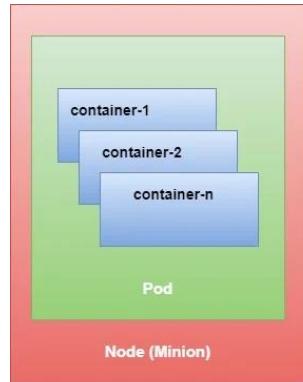
It is possible that your application from container 1 is dependent on some other application from another container such as database, message, logging service in the production environment. You may also need the ability to scale up the number of containers during peak time, for example I am sure you must be familiar with Amazon sale during holidays when they have a bunch of extra offers on all products. In such case they need to **scale up** their resources for applications to be able to handle more number of users. Once the festive offer is finished then they would again need to **scale down** the amount of containers with applications. To enable this functionality we need an underlying platform with a set of resources and capabilities. The platform needs to **orchestrate** the connectivity between the containers and automatically scale up or down based on the load. This while process of deploying and managing containers is known as **container orchestration** **Kubernetes is thus a container orchestration technology** used to orchestrate the deployment and management of hundreds and thousands of containers in a cluster environment. There are multiple similar technologies available today, docker has it's own orchestration software i.e. Docker Swarm, Kubernetes from Google and Mesos from Apache.

Kubernetes Pod

A **Pod** is the Kubernetes fundamental building block, comprised of **one or more containers**, a **shared networking layer**, and **shared filesystem volumes**.

As an **ephemeral unit** with **unique IP address**, a pod is controlled and managed by workload resources controllers via PodTemplates.

These templates, specifications for creating pods, are included in other objects named **Deployments**, **Jobs** and **DaemonSets**.



A basic architectural diagram of Kubernetes and the relationship between containers, pods, and physical worker nodes which were referred as Minion in past.

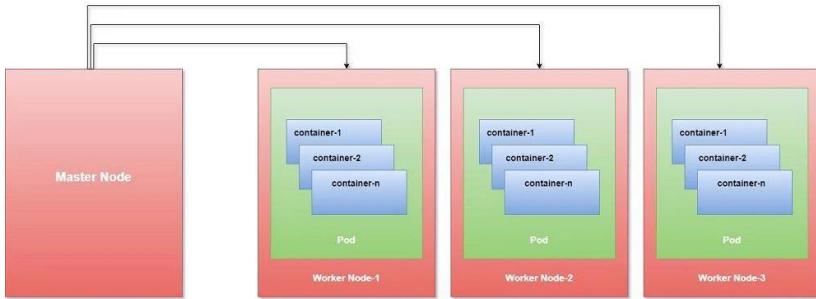
Nodes (Minion)

- You can think of these as container clients.
- These are individual hosts (physical or virtual) on which Docker would be installed to host different containers within your managed cluster
- Each Node will run ETCD (key pair management and communication service, used by Kubernetes for exchanging messages and reporting Cluster status) as well as the Kubernetes proxy

A Pod is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers. We're not implying that a pod always includes more than one container—it's common for pods to contain only a single container. The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node—it never spans multiple worker nodes. These containers are guaranteed (by the cluster controller) to be located on the same host machine in order to facilitate sharing of resources.

are assigned unique IP address within each cluster. These allow an application to use ports without having to worry about conflicting port utilization. Pods can contain definitions of disk volumes or share and then provide access from those to all the members (containers) with the pod.

Kubernetes Architecture: cluster

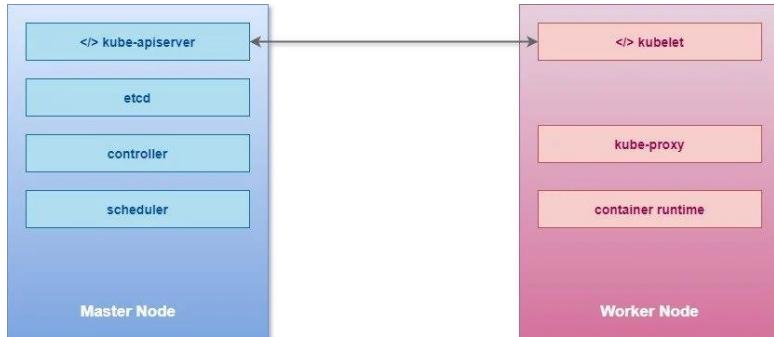


A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Each cluster includes at least one worker node;

- The worker nodes host Pods, that implement of the applications.
- The master manages the worker nodes and the Pods in the cluster.

- A **Kubernetes Cluster** consists of Master and Client node setup where we will have one Master or **Controller** node along with multiple Client nodes also referred as **worker** nodes or in minions.
- A **Master** is a node with Kubernetes installed and is responsible for the actual orchestration of containers on the worker nodes. It will contain all the information of cluster nodes, monitor each node and if a worker node fails then it will move the workload from the failed node to another worker node.
- A **Node** is a worker machine which can be a physical or virtual machine on which Kubernetes is installed.
- Kubernetes does not deploy containers directly into the worker nodes, the containers are encapsulated into a Kubernetes object known as **Pods**.
- A pod is a single instance of an application and they are the smallest deployable units of computing that you can create and manage in Kubernetes.
- You will deploy containers inside these pods where you will deploy your application

Kubernetes Components



Master

- The master is the control plane of Kubernetes.
- It consists of several components, such as an API server, a scheduler, and a controller manager.
- The master is responsible for the global state of the cluster, cluster-level scheduling of pods, and handling of events. Usually, all the master components are set up on a single host.
- When considering high-availability scenarios or very large clusters, you will want to have master redundancy.

The master is in charge of:

- exposing the Kubernetes (REST) API,
- scheduling applications,
- managing the cluster,
- directing communications across the entire system,
- monitoring the containers running in each node as well as the health of all the registered nodes.

API Server

- The API server acts as a frontend for Kubernetes
- It can easily scale horizontally as it is stateless and stores all the data in the etcd cluster.
- The users, management devices, command line interfaces, all talk to API server to interact with Kubernetes cluster

etcd key store

- It is a distributed reliable key value store
- Kubernetes uses it to store the entire cluster state
- In a small, transient cluster a single instance of etcd can run on the same node with all the other master components, but for more substantial clusters, it is typical to have a **three-node or even five-node etcd cluster for redundancy and high availability.**
- It is responsible for implementing locks within the clusters to ensure that there are no conflicts between the masters

Scheduler

Kube-scheduler is responsible for scheduling pods into nodes. This is a very complicated task as it needs to consider multiple interacting factors, such as the following:

- **Resource requirements**
- **Service requirements**
- **Hardware/software policy constraints**
- **Node affinity and anti-affinity specifications**
- **Pod affinity and anti-affinity specifications**
- **Taints and tolerations** ([Node affinity](#)) is a property of [Pods](#) that *attracts* them to a set of [nodes](#) (either as a preference or a hard requirement). *Taints* are the opposite -- they allow a node to repel a set of pods.)
- **Data locality**
- **Deadlines**

Controllers

- The controllers are the brain behind Orchestration.
- They are responsible for noticing and responding when Nodes, containers or end points goes down
- The controller makes decision to bring up new containers in such case

Container Runtime

It is the underlying software that is used to run containers. In our case we will be using Docker as the underlying container but there are other options as well such as:

- Docker (via a CRI shim)
- rkt (direct integration to be replaced with Rktlet)
- CRI-O
- Frakti (Kubernetes on the Hypervisor, previously Hypernetes)
- rktlet (CRI implementation for rkt)
- CRI-containerd

The major design policy is that Kubernetes itself should be completely decoupled from specific runtimes. The **Container Runtime Interface (CRI)** enables it.

kubelet

It is the agent that runs on each nodes in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected. That includes the following:

- Receiving pod specs
- Downloading pod secrets from the API server
- Mounting volumes
- Running the pod's containers (via the configured runtime)
- Reporting the status of the node and each pod
- Running the container startup, liveness, and readiness probes

kube-proxy

- Kube-proxy does low-level network housekeeping on each node
- Containers run on the server nodes, but they interact with each other as they are running in a unified networking setup.
- kube-proxy makes it possible for containers to communicate, although they are running on different nodes.
- It reflects the Kubernetes services locally and can perform TCP and UDP forwarding.
- It finds cluster IPs via environment variables or DNS.

Kubernetes Control Plane: Master

API Server

- The API server acts as a frontend for Kubernetes
- It can easily scale horizontally as it is stateless and stores all the data in the etcd cluster.
- The users, management devices, command line interfaces, all talk to API server to interact with Kubernetes cluster

etcd key store

- It is a distributed reliable key value store
- Kubernetes uses it to store the entire cluster state
- In a small, transient cluster a single instance of etcd can run on the same node with all the other master components.
- For more substantial clusters, it is typical to have a three-node or even five-node etcd cluster for redundancy and high availability.
- It is responsible for implementing locks within the clusters to ensure that there are no conflicts between the masters

Master

- The master is the control plane of Kubernetes.
- It consists of several components, such as an API server, a scheduler, and a controller manager.
- The master is responsible for the global state of the cluster, cluster-level scheduling of pods, and handling of events. Usually, all the master components are set up on a single host.
- When considering high-availability scenarios or very large clusters, you will want to have master redundancy.

The master is in charge of:

- exposing the Kubernetes (REST) API,
- scheduling applications,
- managing the cluster,
- directing communications across the entire system,
- monitoring the containers running in each node as well as the health of all the registered nodes.

Kubernetes Control Plane: Master

Scheduler

- Kube-scheduler is responsible for scheduling pods into nodes. This is a very complicated task as it needs to consider multiple interacting factors, such as the following:
 - Resource requirements
 - Service requirements
 - Hardware/software policy constraints
 - Node affinity and anti-affinity specifications
 - Pod affinity and anti-affinity specifications
 - Taints and tolerations
 - Data locality
 - Deadlines

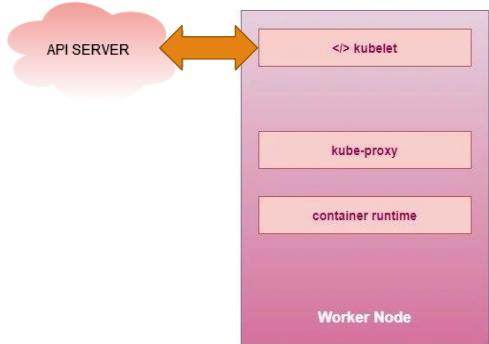
Controllers

- The controllers are the brain behind Orchestration.
- They are responsible for noticing and responding when Nodes, containers or end points goes down
- The controller makes decision to bring up new containers in such case

- The Kubernetes master runs the following components that form the control plane:
- **API server:** the front-end for the Kubernetes control plane that exposes the Kubernetes API
 - kube-apiserver, component of Kubernetes designed to scale horizontally
- **etcd:** is a persistent, lightweight, distributed key-value data store that maintains the entire state of the cluster at time
- **scheduler:** manages new created Pods and selects nodes to host them
 - kube-scheduler
- **Controller manager:** control loop that checks the state of the cluster through the apiserver and makes changes attempting to transfer the current cluster state into the desired cluster state
 - kube-controller-manager

- cloud-controller-manager
- **Node affinity** ensures that pods are hosted on particular node. **Pod affinity** ensures two pods to be co-located in a single node.
- **Taints** are the opposite -- they allow a node to repel a set of pods.
- **Tolerations** are applied to pods. Tolerations allow the scheduler to schedule pods with matching taints. Tolerations allow scheduling but don't guarantee scheduling: the scheduler also evaluates other parameters as part of its function.
 - Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.
- **Deadline:** When setting step **Timeout** settings, it's important to know that Kubernetes has an optional deadline parameter that specifies the number of seconds you want Kubernetes to wait for your Deployment to progress before the system reports back that the Deployment has [failed progressing](#).

Kubernetes Architecture: Worker



- Each worker node has an agent, the Kubelet.
- Kubelet talks with API server and runs the assigned workloads on the node.
- Kube-Proxy handles the traffic coming from and to the pods within the node.
- Addons for cluster functions
- DNS

Node components run on each node, keeping the pods running and providing the Kubernetes runtime environment.

Kubelet: An agent that runs on every node of the cluster. It makes sure that the containers are running in a pod. The kubelet receives a set of PodSpecs which are provided through various mechanisms, and makes sure that the containers described in these PodSpecs are functioning properly and are healthy. The kubelet does not manage containers that were not created by Kubernetes.

kube-proxy: It is a proxy running on each node of the cluster, responsible for managing the Kubernetes Services. Kube proxies keep networking rules on nodes. These rules allow communication to the other nodes of the cluster or to the outside. The kube-proxy uses the operating system libraries whenever possible; otherwise the kube-proxy manages the traffic directly.

Container Runtime: The container runtime is the software that is responsible for running the containers. Kubernetes supports several container runtimes: Docker, containerd, cri-o, rktlet and all Kubernetes CRI (Container Runtime Interface) implementations.

Addons: Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster functionality. Since addons provide cluster-level functionality, resources that need a namespace are placed in the kube-system namespace.

DNS: While other addons are not strictly required, all Kubernetes clusters should be equipped with a cluster DNS, since many applications need it. Cluster DNS is an additional DNS server to other DNS servers on the network, and specifically takes care of DNS records for Kubernetes services. Containers run by Kubernetes automatically use this server for DNS resolution.

Kubernetes worker components

kubelet

- It is the agent that runs on each nodes in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected. That includes the following:
 - Receiving pod specs
 - Downloading pod secrets from the API server
 - Mounting volumes
 - Running the pod's containers (via the configured runtime)
 - Reporting the status of the node and each pod
 - Running the container startup, liveness, and readiness probes

kube-proxy

- Kube-proxy does low-level network housekeeping on each node
- Containers run on the server nodes, but they interact with each other as they are running in a unified networking setup.
- kube-proxy makes it possible for containers to communicate, although they are running on different nodes.
- It reflects the Kubernetes services locally and can perform TCP and UDP forwarding.
- It finds cluster IPs via environment variables or DNS.

Kubernetes worker components

Container Runtime

- It is the underlying software that is used to run containers. In our case we will be using Docker as the underlying container but there are other options as well such as:
 - Docker (via a CRI shim)
 - rkt (direct integration to be replaced with Rktlet)
 - CRI-O
 - Frakti (Kubernetes on the Hypervisor, previously Hypernetes)
 - rktlet (CRI implementation for rkt)
 - CRI-containerd
- The major design policy is that Kubernetes itself should be completely decoupled from specific runtimes. The **Container Runtime Interface (CRI)** enables it.

The CRI is a **plugin interface** which enables the kubelet to use a wide variety of container runtimes, without having a need to recompile the cluster components.

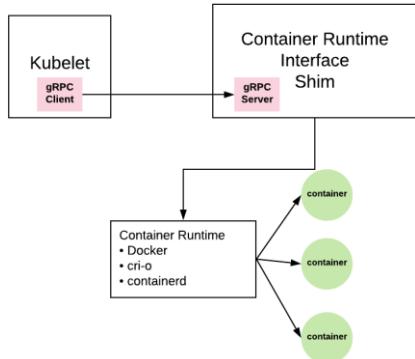
You need a working container runtime on each Node in your cluster, so that the kubelet can launch Pods and their containers.

The Container Runtime Interface (CRI) is the main protocol for the communication between the kubelet and Container Runtime.

The Kubernetes Container Runtime Interface (CRI) defines the main gRPC protocol for the communication between the node components kubelet and container runtime.

gRPC is a modern open source high performance Remote Procedure Call (RPC) framework.

Container Runtime Interface



kubelet interacts with the Container Runtime Interface using gRPC to create and destroy containers on a worker node

When it's time to create or destroy a container on a node, kubelet sends a message to the gRPC server running on the node's CRI instance to do the deed, then the CRI interacts with the container runtime engine installed on the worker node to do what is necessary.

For example, when kubelet wants to create a container, it uses its gRPC client to send a `CreateContainerRequest` message to the RPC (remote procedure call) function `CreateContainer()` that's hosted on the CRI component. Once container creation is completed, the CRI returns a `CreateContainerResponse` message

See <https://www.mulesoft.com/api-university/grpc-real-world-kubernetes-container-runtime-interface>

A container runtime shim is a lightweight daemon launching the *container runtime* and controlling the container process. The shim's process is tightly bound to the container's process but is completely detached from the manager's process. All the communications between the container and the manager happen through the shim.

Creation of a simple Pod

A Creating Pods using YAML file

- Pods and other Kubernetes resources are usually created by posting a JSON or YAML manifest to the Kubernetes REST API endpoint.

```
[root@controller ~]# cat nginx.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

Creation of a simple Pod

To create the Pod:

```
[root@controller ~]# kubectl create -f nginx.yml  
pod/nginx created
```

Verify the newly created pod, a container for nginx is being created inside the pod:

```
[root@controller ~]# kubectl get pods -o wide  
NAME          READY   STATUS    RESTARTS   AGE   IP           NODE      ...  
nginx         1/1     Running   0          43s   10.36.0.4   worker-1.example.com ...  
...
```

You can use kubectl describe to get more details of a specific resource which in this case is Pod.

```
[root@controller ~]# kubectl describe pod nginx  
...
```

The Kubernetes command-line tool, [kubectl](#), allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. For more information including a complete list of kubectl operations, see the [kubectl reference documentation](#).

Accessing a Pod

We can use kubectl to set up a proxy that will forward all traffic from a local port we specify to a port associated with the Pod we determine. This can be performed using **kubectl port-forward** command.

kubectl port-forward makes a specific Kubernetes API request. That means the system running it needs access to the API server, and any traffic will get tunneled over a single HTTP connection. We use this command to access container content by forwarding one (or more) local ports to a pod. This command is very useful mostly when you would want to troubleshoot a misbehaving pod.

```
kubectl port-forward <resource-type/resource-name> [local_port]<pod_port>
```

For the created nginx Pod:

```
kubectl port-forward pods/nginx 8080:80
```

In this method we will forward the traffic to port **8080 on localhost** (on controller) to port **80 on worker node** based nginx container. This is forwarding any and all traffic that gets created on your local machine at TCP port 8080 to TCP port 80 on the Pod nginx.

The same port forwarding scheme can be used also for other Kuberbetes objects, that are described in this presentation

Sample command to perform port forwarding on **deployment**:

```
kubectl port-forward deployment/my-deployment 8080:80
```

Sample command to perform port forwarding on **replicaset**:

```
kubectl port-forward replicaset/my-replicaset 8080:80
```

Sample command to perform port forwarding on **service**:

```
kubectl port-forward service/my-service 8080:80
```

Kubernetes Namespaces

- Kubernetes uses namespaces to organize objects in the cluster.
- You can think of each namespace as a folder that holds a set of objects.
- Namespace implements strict resource separation
- Resource limitation through quota can be implemented at a Namespace level also
- Use namespaces to separate customer environments within one Kubernetes cluster
- By default, the `kubectl` command-line tool interacts with the `default` namespace.
- If you want to use a different namespace, you can pass kubectl the `--namespace` flag.
- For example, `kubectl --namespace=mystuff` references objects in the `mystuff` namespace.
- If you want to interact with all namespaces - for example, to list all Pods in your cluster you can pass the `--all-namespaces` flag.

Using multiple namespaces allows you to split complex systems with numerous components into smaller distinct groups. This is useful in scenarios wherein you want to split and limit resources across different resources. Resource names only need to be unique within a namespace. Two different namespaces can contain resources of the same name.

Although namespaces allow you to isolate objects into distinct groups, which allows you to operate only on those belonging to the specified namespace, they don't provide any kind of isolation of running objects. For example, you may think that when different users deploy pods across different namespaces, those pods are isolated from each other and can't communicate, but that's not necessarily the case. Whether namespaces provide network isolation depends on which networking solution is deployed with Kubernetes. When the solution doesn't provide inter-namespace network isolation, if a pod in namespace foo knows the IP address of a pod in namespace bar, there is nothing preventing it from sending traffic, such as HTTP requests, to the other pod.

Kubernetes Namespaces

Four namespaces are defined when a cluster is created:

- **default:** this is where all the Kubernetes resources are created by default
- **kube-node-lease:** This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane (Node controller) can detect node failure.
- **kube-public:** a namespace that is world-readable. Generic information can be stored here but it's often empty
- **kube-system:** contains all infrastructure pods

Leases

Distributed systems often have a need for leases, which provide a mechanism to lock shared resources and coordinate activity between members of a set. In Kubernetes, the lease concept is represented by Lease objects in the coordination.k8s.io API Group, which are used for system-critical capabilities such as node heartbeats and component-level leader election.

Node heartbeats

Kubernetes uses the Lease API to communicate kubelet node heartbeats to the Kubernetes API server. For every Node , there is a Lease object with a matching name in the kube-node-lease namespace. Under the hood, every kubelet heartbeat is an update request to this Lease object, updating the spec.renewTime field for the Lease. The Kubernetes control plane uses the time stamp of this field to determine the availability of this Node.

Kubernetes Namespaces

To get the list of available namespaces, operating in the default namespace:

```
[root@controller ~]# kubectl get ns
NAME      STATUS  AGE
default   Active  14d
kube-node-lease  Active  14d
kube-public  Active  14d
kube-system  Active  14d
```

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs.

Alternatively you can also use kubectl **get all** in all the available namespaces:

```
[root@controller ~]# kubectl get all --all-namespaces
NAMESPACE     NAME           READY  STATUS    RESTARTS  AGE
kube-system   pod/coredns-f9fd979d6-nmsq5   1/1    Running   4          14d
...
NAMESPACE     NAME           TYPE    CLUSTER-IP   EXTERNAL-IP PORT(S)  AGE
default       service/kubernetes  ClusterIP  10.96.0.1    443/TCP        14d
...
```

Creating a namespace

To create a Kubernetes namespace using YAML file we would need the **KIND** and **apiVersion**. To get the KIND value of a namespace we will list down the api-resources:

```
[root@controller ~]# kubectl api-resources | grep -iE 'namespace|KIND'
NAME           SHORTNAMES   APIGROUP      NAMESPACED   KIND
namespaces     ns          ""            false        Namespace
```

Now that we have the KIND value, we can use this to get the respective apiVersion:

```
[root@controller ~]# kubectl explain Namespace | head -n 2
KIND:  Namespace
VERSION: v1
```

Now let's create a custom-namespace.yaml file with the following listing's contents:

```
[root@controller ~]# cat app-ns.yml
apiVersion: v1
kind: Namespace
metadata:
  name: app
```

Creating a namespace

Now, use kubectl to post the file to the Kubernetes API server:

```
[root@controller ~]# kubectl create -f create-namespace.yml  
namespace/app created
```

Using kubectl command: You can also create namespaces with the dedicated kubectl create namespace command:

```
[root@controller ~]# kubectl create ns dev  
namespace/dev created
```

Note the possibility of using both declarative and imperative styles.

To get a much detailed output of individual namespace (e.g. default):

```
[root@controller ~]# kubectl describe ns default
```

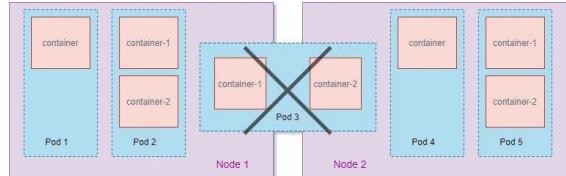
To get the details of namespace in YAML format:

```
[root@controller ~]# kubectl get ns default -o yaml
```

Detailing Kubernetes Pods

The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node and **it never spans multiple worker nodes**

Usually you don't need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as [Job](#) or [Deployment](#). If your Pods need to track state, consider the [StatefulSet](#) resource.



- We already know that a pod is a co-located group of containers and represents the basic building block in Kubernetes.
- Instead of deploying containers individually, you always deploy and operate on a pod of containers.
- We're not implying that a pod always includes more than one container, it's common for pods to contain only a single container.

Workload resources for managing pods

Pods in a Kubernetes cluster are used in two main ways:

- **Pods that run a single container.** The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
- **Pods that run multiple containers that need to work together.** A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service.
- Cases where you might want to run multiple containers in a Pod
 - **Sidecar container:** a container that enhances the primary application, for instance for logging purpose
 - **Ambassador container:** a container that represents the primary container to the outside world, such as proxy
 - **Adapter container:** used to adapt the traffic or data pattern to match the traffic or data pattern in other applications in the cluster
- When using multi-container Pods, the containers typically share data through shared storage.

Whenever we create a pod, a pause container image such as `gcr.io/google_containers/pause:0.8.0` is implicitly required. What is that pause container's purpose? The pause container essentially holds the network namespace for the pod. It does nothing useful and its container image (see [its Dockerfile](#)) basically contains a simple binary that goes to sleep and never wakes up (see [its code](#)). However, when the top container, such as nginx container used before, dies and gets restarted by kubernetes, all the network setup will still be there. Normally, if the last process in a network namespace dies, the namespace will be destroyed. Restarting nginx container without pause would require creating all new network setup. With pause, you will always have that one last thing in the namespace.

Example of Sidecar Scenario

```
[root@controller ~]# cat create-sidecar.yml
kind: Pod
apiVersion: v1
metadata:
  name: sidecar-pod
spec:
  volumes:
    - name: logs
      emptyDir: {}
  containers:
    - name: app
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /var/log/date.txt; sleep 10; done"]
    volumeMounts:
      - name: logs
        mountPath: /var/log
    - name: sidecar
      image: centos/httpd
      ports:
        - containerPort: 80
      volumeMounts:
        - name: logs
          mountPath: /var/www/html
[root@controller ~]# kubectl create -f create-sidecar.yml
```

For a Pod that defines an emptyDir volume, the volume is created when the Pod is assigned to a node. As the name says, the emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

The `{}` at the end means we do not supply any further requirements for the `emptyDir`.

If the `-c` option is present in `sh`, then commands are read from `string`. If there are arguments after the `string`, they are assigned to the positional parameters, starting with `$0`.

Example of Sidecar Scenario

We can connect to this pod using:

```
[root@controller ~]# kubectl exec -it sidecar-pod -c sidecar -- /bin/bash  
[root@sidecar-pod ~]#
```

Now we can use curl to check the content of date.txt where we were appending the date command output every 10 seconds in a loop:

```
[root@sidecar-pod ~]# curl http://localhost/date.txt  
Fri Nov 27 05:43:00 UTC 2020  
Fri Nov 27 05:43:10 UTC 2020  
Fri Nov 27 05:43:20 UTC 2020  
Fri Nov 27 05:43:30 UTC 2020  
Fri Nov 27 05:43:40 UTC 2020  
Fri Nov 27 05:43:50 UTC 2020  
Fri Nov 27 05:44:00 UTC 2020  
Fri Nov 27 05:44:11 UTC 2020  
Fri Nov 27 05:44:21 UTC 2020
```

Kubernetes Controllers

In robotics and automation, a control loop is a non-terminating loop that regulates the state of a system, as a thermostat in a room.

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

A controller tracks at least one Kubernetes resource type. These objects have a **spec** field that represents the desired state. The Job controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it Finished.

Kubernetes comes with a set of **built-in controllers that run inside the kube-controller-manager**. These built-in controllers provide important core behaviors.

<https://kubernetes.io/docs/concepts/architecture/controller/>

Kubernetes Objects

Kubernetes objects are persistent entities provided by Kubernetes for deploying, maintaining, and scaling applications.

Kubernetes uses these entities to represent the state of your cluster.

The objects can describe:

- Which containerized applications are being executed, and sometimes on which nodes.
- The resources made available to the executed applications
- The policies governing the executed applications, such as restart policies, upgrades, and fault-tolerance

Kubernetes Objects

Different methods to create objects in Kubernetes

- There are two approaches to create different kind of objects in Kubernetes **Declarative** and **Imperative**
- The recommended way to work with kubectl is declarative way , by writing your manifest files and applying `kubectl {apply|create} -f manifest.yml`
- With YAML file you have more control over the different properties you can add to your container compared to imperative method
- With imperative you can just use `kubectl` command line to create different objects
- The one challenge with declarative method of creating objects would be to creating a YAML file, to overcome this you can get the YAML file content from any existing object, for example:

```
kubectl get <object> -o yaml
```

and then use that as a template to create another object

```
kubectl {replace|apply} -f nginx.yaml.
```

Kubernetes Workload Resources

- Job
- ReplicaSet
- StatefulSets
- Deployment
- DaemonSet
- CronJob
- ReplicationController

[Deployment](#) is an object which can own ReplicaSets and update them and their Pods via declarative, server-side rolling updates. While ReplicaSets can be used independently, today they're mainly used by Deployments as a mechanism to orchestrate Pod creation, deletion and updates. When you use Deployments you don't have to worry about managing the ReplicaSets that they create. Deployments own and manage their ReplicaSets. As such, it is recommended to use Deployments when you want ReplicaSets.

Automatic Clean-up for Finished Jobs

TTL-after-finished [controller](#) provides a TTL (time to live) mechanism to limit the lifetime of resource objects that have finished execution. TTL controller only handles [Jobs](#).

ReplicationController ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available. If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods.

Job

- A Job creates one or more Pods and ensures that a specified number of them **successfully terminate**.
- Pods normally are created as run forever. To create a Pod that runs for a limited duration, use Jobs instead
- Jobs are useful for tasks, like backup, calculation, batch processing and more
- A Pod that is started by a Job must have its **restartPolicy** set to **OnFailure** or **Never**
 - OnFailure will re-run the container on the same Pod
 - Never will re-run the failing container in a new Pod
- Three different Job types exist, which can be created by specifying completions and parallelism parameters:
 - **Non-parallel Jobs:** one Pod is started, unless the Pod fails (completions=1, parallelism=1)
 - **Parallel Jobs with a fixed completion count:** the Job is complete after successfully running as many times as specified in jobs.spec.completions (completions=n, parallelism=m)
 - **Parallel Jobs with a work queue:** multiple Jobs are started, when **one** completes successfully, the Job is complete (completions=1, parallelism=m)
 - When completions and parallelism parameters are unset under .spec, both are defaulted to 1

A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

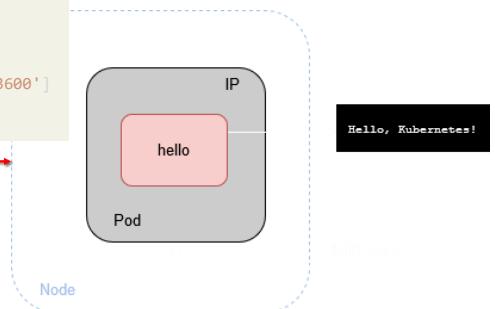
You can also use a Job to run multiple Pods in parallel.

See: <https://kubernetes.io/docs/concepts/workloads/controllers/job/>

Basic Object Example

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    #this is a spec template
    spec:
      containers:
        - name: hello
          image: busybox
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
#the pod template ends here
```

A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete.



If you'd like your container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a restartPolicy of "Always" or "OnFailure". Otherwise you can use "Never"

OnFailure means that **the container will only be restarted if it exited with a non-zero exit code** (i.e. something went wrong). This is useful when you want accomplish a certain task with the pod, and ensure that it completes successfully - if it doesn't it will be restarted until it does.

Example: Running job pods sequentially

If you need a Job to run more than once, you set completions to how many times you want the Job's pod to run.

```
[root@controller ~]# cat pod-simple-job.yml
apiVersion: batch/v1
kind: Job
metadata:
  name: pod-simple-job
spec:
  completions: 3
  template:
    spec:
      containers:
        - name: sleepy
          image: alpine
          command: ["/bin/sleep"]
          args: ["5"]
      restartPolicy: Never
```

```
[root@controller ~]# kubectl create -f pod-simple-job.yml
job.batch/pod-simple-job created
```

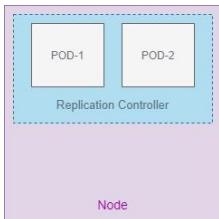
After some time, once all the jobs are completed:

```
[root@controller ~]# kubectl get jobs
NAME      COMPLETIONS DURATION AGE
pod-simple-job 3/3       35s      5m7s
```

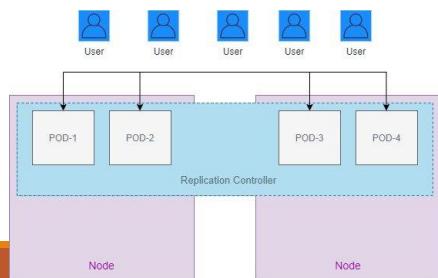
ReplicationController and ReplicaSet

A ReplicationController is a Kubernetes resource that ensures its pods are always kept running.

- If the pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the pod was evicted from the node, the ReplicationController notices the missing pod and creates a replacement pod.
- The ReplicationController in general, are meant to create and manage multiple copies (replicas) of a pod

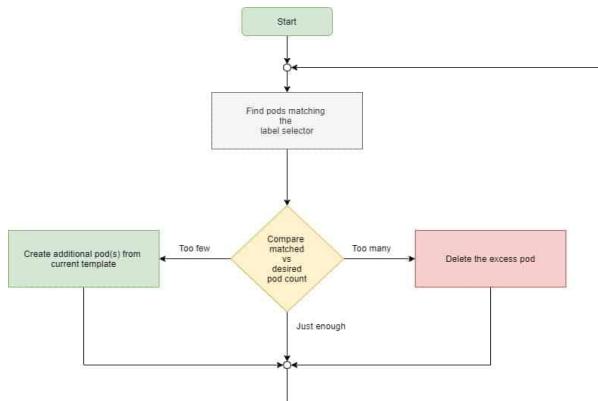


It is possible that your Node is out of resources while creating new pods with Replication controllers or replica sets, in such case it will automatically create new pods on another available cluster node



ReplicationController and ReplicaSet

A ReplicationController's task is to make sure that an **exact number of pods always matches its label selector**. If it doesn't, the ReplicationController takes the appropriate action to reconcile the actual with the desired number. The ReplicationController in general, are meant to create and manage multiple copies (replicas) of a pod



A ReplicationController has three essential parts:

- A **label selector**, which determines what pods are in the ReplicationController's scope
- A **replica count**, which specifies the desired number of pods that should be running
- A **pod template**, which is used when creating new pod replicas

ReplicationController and ReplicaSet

```
[root@controller ~]# cat replication-controller.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: dev
spec:
  replicas: 3
  selector:
    app: myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: dev
    spec:
      containers:
        - name: nginx-container
          image: nginx
```

Horizontally scaling pods

You've seen how ReplicationControllers make sure a specific number of pod instances is always running. Because it's incredibly simple to change the desired number of replicas, this also means scaling pods horizontally is trivial.

Assuming you suddenly expect that load on your application is going to increase so you must deploy more pods until the load is reduced, in such case you can easily scale up the number of pods runtime.

For example, for having 6 replicas:

```
[root@controller ~]# kubectl scale rc myapp-rc --replicas=6
replicationcontroller/myapp-rc scaled
```

For downscaling to 3 replicas:

```
[root@controller ~]# kubectl scale rc myapp-rc --replicas=3
replicationcontroller/myapp-rc scaled
```

ReplicationController and ReplicaSet

Comparing a ReplicaSet to a ReplicationController

- A ReplicaSet behaves exactly like a ReplicationController, but it has more expressive pod selectors.
- Whereas a ReplicationController's label selector only allows matching pods that include a certain label, a ReplicaSet's selector also allows matching pods that lack a certain label or pods that include a certain label key, regardless of its value.
 - Also, for example, a single ReplicationController can't match pods with the label `env=production` and those with the label `env=devel` at the same time. It can only match either pods with the `env=production` label or pods with the `env=devel` label. But a single ReplicaSet can match both sets of pods and treat them as a single group.
 - Similarly, a ReplicationController can't match pods based merely on the presence of a label key, regardless of its value, whereas a ReplicaSet can. For example, a ReplicaSet can match all pods that include a label with the key `env`, whatever its actual value is (you can think of it as `env=*`).

Replica Controller is deprecated and replaced by ReplicaSets.

The replica set and the replication controller's key difference is that **the replication controller only supports equality-based selectors whereas the replica set supports set-based selectors**.

Deployments are recommended over ReplicaSets.

ReplicationController and ReplicaSet

```
[root@controller ~]# cat replica-set.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: dev
  spec:
    containers:
      - name: nginx-container
        image: nginx
```

The only difference is in the selector, instead of listing labels the pods need to have directly under the selector property, you're specifying them under **selector.matchLabels**. This is the simpler (and less expressive) way of defining label selectors in a ReplicaSet.

Se al posto di myapp nel file si scrive pippo il replicaset viene creato lo stesso

ReplicationController and ReplicaSet

```
[root@controller ~]# cat replica-set.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
labels:
  app: myapp
  type: dev
spec:
  replicas: 3
  selector:
    matchExpressions:
    - key: app
      operator: In
      values:
      - myapp
  template:
    metadata:
      name: myapp-pod
    labels:
      app: myapp
      type: dev
  spec:
...
...
```

Now, we will rewrite the selector to use the more powerful `matchExpressions` property:

Here, this selector requires the pod to contain a label with the “`app`” key and the label’s value must be “`myapp`”.

You can add additional expressions to the selector. As in the example, each expression must contain a key, an operator, and possibly (depending on the operator) a list of values. You’ll see four valid operators:

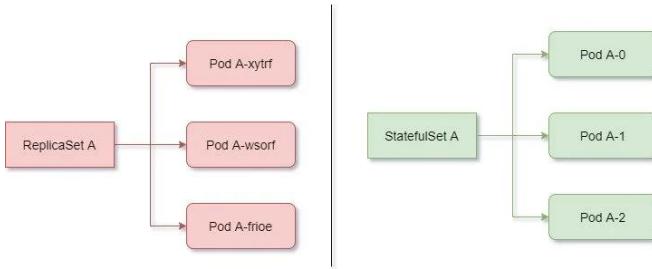
- **In**: Label’s value must match one of the specified values.
- **NotIn**: Label’s value must not match any of the specified values.
- **Exists**: Pod must include a label with the specified key (the value isn’t important). When using this operator, you shouldn’t specify the `values` field.
- **DoesNotExist**: Pod must not include a label with the specified key. The `values` property must not be specified.

StatefulSet

We learned about ReplicaSets which creates multiple pod replicas from a single pod template. These replicas don't differ from each other, apart from their name and IP address.

Instead of using a ReplicaSet to run these types of pods, we can create a **StatefulSet** resource, which is specifically tailored to applications where **instances of the application must be treated as completely alike individuals**, with each one having a stable name and state.

Each pod created by a StatefulSet is assigned an ordinal index (zero-based), which is then used to derive the pod's name and hostname, and to attach stable storage to the pod. The names of the pods are thus predictable, because each pod's name is derived from the StatefulSet's name and the ordinal index of the instance. Rather than the pods having random names, they're nicely organized,



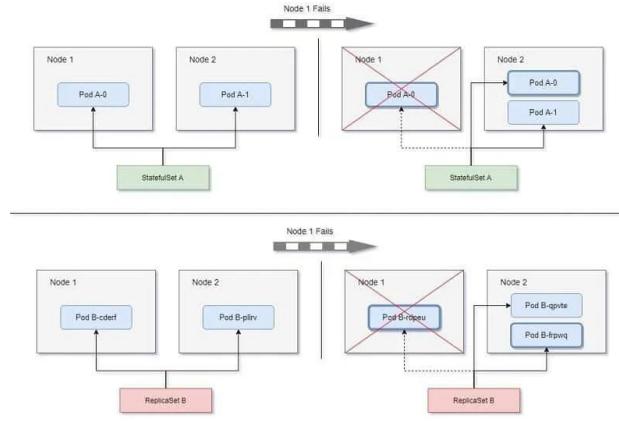
StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.

Using StatefulSets

StatefulSet

When a pod instance managed by a StatefulSet disappears (because the node the pod was running on has failed, it was evicted from the node, or someone deleted the pod object manually), the StatefulSet makes sure it's replaced with a new instance—similar to how ReplicaSets do it. But in contrast to ReplicaSets, the replacement pod gets the same name and hostname as the pod that has disappeared.



To summarise, Kubernetes StatefulSet manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods.

Limitations

- The storage for a given Pod must either be provisioned by a [PersistentVolume Provisioner](#) based on the requested storage class, or pre-provisioned by an admin.
- Deleting and/or scaling a StatefulSet down will not delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources.
- StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.
- StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the StatefulSet down to 0 prior to deletion.
- When using Rolling Updates with the default Pod Management Policy (OrderedReady), it's possible to get into a broken state that requires manual intervention to repair.

DaemonSet

DaemonSet: ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

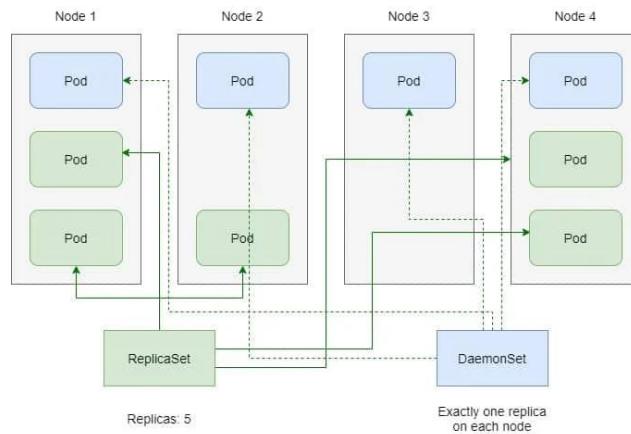
DaemonSets are used to ensure that some or all of your K8S nodes run a copy of a pod, which allows you to run a daemon on every node.

When you add a new node to the cluster, a pod gets added to match the nodes. Similarly, when you remove a node from your cluster, the pod is put into the trash. Deleting a DaemonSet cleans up the pods that it previously created.

A Daemonset is another controller that manages pods like Deployments, ReplicaSets, and StatefulSets. It was created for one particular purpose: ensuring that the pods it manages to run on all the cluster nodes. As soon as a node joins the cluster, the DaemonSet ensures that it has the necessary pods running on it. When the node leaves the cluster, those pods are garbage collected.

DaemonSets are used in Kubernetes when you need to run one or more pods on all (or a subset of) the nodes in a cluster. The typical use case for a DaemonSet is logging and monitoring for the hosts. For example, a node needs a service (daemon) that collects health or log data and pushes them to a central system or database (like ELK stack). DaemonSets can be deployed to specific nodes either by the nodes' user-defined labels or using values provided by Kubernetes like the node hostname.

DaemonSet



A Kubernetes **DaemonSet** ensures a copy of a Pod is running across a set of nodes in a Kubernetes cluster. DaemonSets are used to deploy system daemons such as log collectors and monitoring agents, which typically must run on every node.

DaemonSets share similar functionality with ReplicaSets; both create Pods that are expected to be long-running services and ensure that the desired state and the observed state of the cluster match.

When to use - DaemonSet and/or ReplicaSet?

Given the similarities between DaemonSets and ReplicaSets, it's important to understand when to use one over the other.

- ReplicaSets should be used when your application is completely decoupled from the node and you can run multiple copies on a given node without special consideration.
- DaemonSets should be used when a single copy of your application must run on all or a subset of the nodes in the cluster.

Using a DaemonSet to run a pod on every node

A DaemonSet deploys pods to all nodes in the cluster, unless you specify that the pods should only run on a subset of all the nodes. This is done by specifying the `node-Selector` property in the pod template, which is part of the DaemonSet

definition (similar to the pod template in a ReplicaSet or ReplicationController).

DaemonSet

```
[root@controller ~]# cat fluentd-daemonset.yml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  labels:
    app: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
        - name: fluentd
          image: fluent/fluentd:v0.14.10
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 200Mi
      volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
```

Resource limits are useful if don't want these daemons to eat up all system's resources.

Once you have a valid DaemonSet configuration in place, you can use the `kubectl create` command to submit the DaemonSet to the Kubernetes API. In this section we will create a DaemonSet to ensure the fluentd HTTP server is running on every node in our cluster.

- **Ei = EiB = Exbibyte.** $1Ei = 2^{60} = 1,152,921,504,606,846,976$ bytes
- **Pi = PiB = Pebibyte.** $1Pi = 2^{50} = 1,125,899,906,842,624$ bytes
- **Ti = TiB = Tebibyte.** $1Ti = 2^{40} = 1,099,511,627,776$ bytes
- **Gi = GiB = Gibibyte.** $1Gi = 2^{30} = 1,073,741,824$ bytes
- **Mi = MiB = Mebibyte.** $1Mi = 2^{20} = 1,048,576$ bytes
- **Ki = KiB = Kibibyte.** $1Ki = 2^{10} = 1,024$ bytes

To deploy:

```
kubectl apply -f ./fluentd-daemonset.yml
```

To delete:

```
kubectl delete daemonset fluentd-elasticsearch --namespace=kube-system
```

The key difference between kubectl apply and create is that **apply creates Kubernetes objects through a declarative syntax, while the create command is imperative**. The important thing to understand about kubectl create vs. kubectl apply is that you use kubectl create to create Kubernetes resources imperatively at the command-line or [declaratively](#) against a manifest file. However, you can use kubectl create declaratively only to create a new resource.

CronJob

- At the configured time, Kubernetes will create a Job resource according to the Job template configured in the CronJob object. When the Job resource is created, one or more pod replicas will be created and started according to the Job's pod template
- Let us first understand the basics of cron job. You can get the structure of different fields in crontab in [/etc/crontab](#) file:

```
# For details see man 4 crontabs

# Example of job definition:
# .----- minute (0 - 59)
# | .----- hour (0 - 23)
# | | .----- day of month (1 - 31)
# | | | .---- month (1 - 12) OR jan,feb,mar,apr ...
# | | | | .--- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# | | | |
# * * * * * user-name command to be executed
```

ob resources run their pods immediately when you create the Job resource. But many batch jobs need to be run at a specific time in the future or repeatedly in the specified interval. In Linux- and UNIX-like operating systems, these jobs are better known as cron jobs. Kubernetes supports them, too.

A cron job in Kubernetes is configured by creating a CronJob resource. The schedule for running the job is specified in the well-known cron format, so if you're familiar with regular cron jobs, you'll understand Kubernetes' CronJobs in a matter of seconds.

CronJob

- Example: `0 5 * * 1 tar -zcf /var/backups/home.tgz /home/`
- The first section of a cron job, a 0 in this example, refers to the minute that the job will happen. Since 0 is used here, the minutes section is :00.
- The next section is the hour; 5 in this case. Therefore, we can gather so far that the command will be run at 5:00 a.m. (cron uses military time).
- The third section refers to the day of the month, but we have an asterisk (*) here. The asterisk, in this case, means that it doesn't matter what day of the month it is; just run it.
- The fourth section is another asterisk. In this example, it means we don't care which month it is. So far, we have a command we want to run at 5:00 a.m., regardless of the date.
- The fifth field is set to 1 in our example, and this field represents the day of the week. Since we have 1 here, it means that we want to run the command on Mondays. (Sunday would have been 0).
- Finally, the command that we want to execute is listed.
- If we put it all together, this line means we want to run the following command **every Monday at 5:00 a.m.**

CronJob

- Imagine you need to run the batch job every 2 minutes. To do that, create a CronJob resource with the following specification.
- The highlighted section is the template for the Job resources that will be created by this CronJob where our defined task will run **every 2 minutes**.

```
[root@controller ~]# cat pod-cronjob.yml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pod-cronjob
spec:
  schedule: "*/2 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: pod-cronjob
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo hello from k8s cluster
  restartPolicy: OnFailure
```

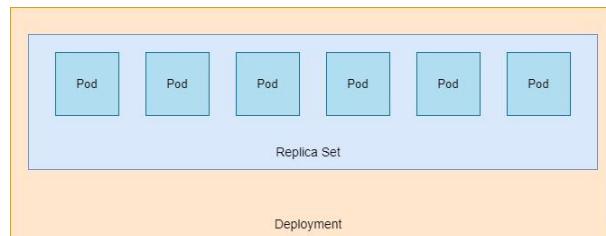
Step values can be used in conjunction with ranges. Following a range with <number> specifies skips of the number's value through the range. For example, 0-23/2 can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is 0,2,4,6,8,10,12,14,16,18,20,22). Steps are also permitted after an asterisk, so if you want to say "every two hours", just use */2.

Deployment

Deployment: Deployment provides declarative updates for [Pods](#) and [ReplicaSets](#) (multiple replicas of pods).

- When running Pods in datacenter, additional features may be needed such as scalability, updates and rollback etc which are offered by Deployments
- A Deployment is a [higher-level resource](#) meant for deploying applications and updating them declaratively, instead of doing it through a ReplicationController or a ReplicaSet, which are both considered lower-level concepts.
- When you create a Deployment, a ReplicaSet resource is created underneath. Replica-Sets replicate and manage pods, as well. Thus, when a Deployment is used, the actual pods are created and managed by the Deployment's ReplicaSets, not by the Deployment directly
- The [desired state](#) of ReplicaSet is described in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Deployment



In the deployment spec, following properties are managed:

- **replicas:** explains how many copies of each Pod should be running
- **strategy:** explains how Pods should be updated
- **selector:** uses matchLabels to identify how labels are matched against the Pod
- **template:** contains the pod specification and is used in a deployment to create Pods

Example:

Creation of the Deployment:
kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Desired state

pod template

Deployment

The name of a ReplicaSet object is composed of the name of the controller plus a randomly generated string (for example, nginx-v1-m33mv). The pods created by the Deployment include an additional numeric value in the middle of their names. What is that exactly?

The number corresponds to the hashed value of the pod template in the Deployment and the ReplicaSet managing these pods.

```
[[root@controller ~]# kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
nginx         1/1     Running   5          4d1h
nginx-deploy-d98cc8bdb-48ppw  1/1     Running   0          80s
nginx-deploy-d98cc8bdb-nvcb5  1/1     Running   0          80s
```

```
[root@controller ~]# cat rolling-nginx.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolling-nginx
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  selector:
    matchLabels:
      app: rolling-nginx
  template:
    metadata:
      labels:
        app: rolling-nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.9
```

Example:

Use of strategy: Deployment for **RollingUpdate**

The RollingUpdate options are used to guarantee a certain **minimal** and **maximal** number of Pods to be always available:

maxUnavailable: The maximum number of Pods that can be unavailable during updating. The value could be a percentage (the default is 25%) or an integer. If the value of maxSurge is 0, which means no tolerance of the number of Pods over the desired number, the value of maxUnavailable cannot be 0.

maxSurge: The maximum number of Pods that can be created over the desired number of ReplicaSet during updating. The value could be a percentage (the default is 25%) or an integer. If the value of maxUnavailable is 0, which means the number of serving Pods should always meet the desired number, the value of maxSurge cannot be 0

Here we want four replicas and we have set the update strategy to RollingUpdate. The MaxSurge value is 1, which is the maximum above the desired number of replicas, while maxUnavailable is 1. This means that throughout the process, we should have at least 3 and a maximum of 5 running pods.

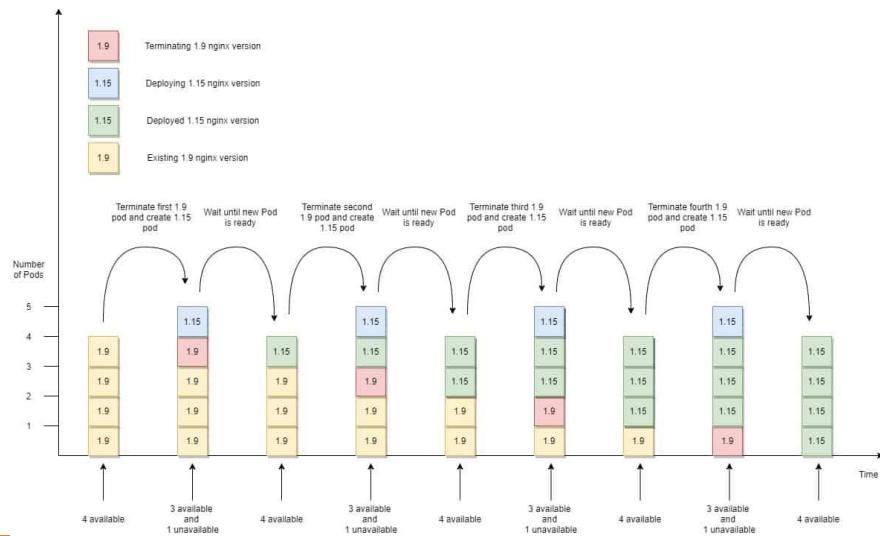
Initially, the pods run the first version of your application, let's suppose its image is tagged as **v1**. You then develop a newer version of the app and push it to an image repository as a new image, tagged as **v2**. You'd next like to replace all the pods with this new version.

You have two ways of updating all those pods. You can do one of the following:

•**Recreate:** Delete all existing pods first and then start the new ones. This will lead to a temporary unavailability.

•**Rolling Update:** Updates Pod one at a time to guarantee availability of the application. This is the preferred approach and you can further tune its behaviour.

Rollout history



Label and Selector

- Any object in Kubernetes may have key-value pairs associated with it. You will see them on pods, replication controllers, replica sets, services, and so on.
- Kubernetes refers to these key-value pairs as labels.
- Labels do not provide uniqueness.
- In general, we expect many objects to carry the same label(s).
- Labels are queryable — which makes them especially useful in organizing things. The mechanism for this query is a **label selector**. A label selector is a string that identifies which labels you are trying to match. There are currently two types of selectors: **equality-based** and **set-based selectors**.
- Via a label selector, the client/user can identify a set of objects.
- The label selector is the core grouping primitive in Kubernetes. Labels are used for organization and selection of subsets of objects, and can be added to objects at creation time and/or modified at any time during cluster operations.

Example

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

The Deployment selects Pods with the label «app:nginx»

Pod label «app: nginx»

version of the Kubernetes API used to create this object
type of object you want to create
data that helps uniquely identifying the object, including a name string, UID, etc
specific format of the object spec

Label and Selector

Equality-based selector

- An equality-based test is just a “IS/IS NOT” test.
- For example: `tier = frontend` will return all pods that have a label with the key “tier” and the value “frontend”. On the other hand, if we wanted to get all the pods that were not in the frontend tier, we would say:

`tier != frontend`

- You can also combine requirements with commas like so:

`tier != frontend, game = super-shooter-2`

This would return all pods that were part of the game named super-shooter-2 but were not in its frontend tier.

Label and Selector

Set-based selectors

- Set-based tests, on the other hand, are of the “IN/NOT IN” variety. For example:

`environment in (production, qa)`

`tier notin (frontend, backend)`

Partition

The first test returns pods that have the `environment` label and a value of either `production` or `qa`. The next test returns all the pods not in the `frontend` or `backend` tiers. Finally, the third test will return all pods that have the `partition` label—no matter what value it contains.

How to limit Kubernetes resources (CPU & Memory)

There were three different resource types for which requests and limits could be imposed on a Pod and Container:

- CPU
- Memory
- Hugepages (Kubernetes v1.14 or newer)

CPU and memory are collectively referred to as compute resources, or just resources. Compute resources are measurable quantities that can be requested, allocated, and consumed

We can define a **soft limit** value for the allowed resources for individual Pod and Containers and an **upper limit** above which the usage would be denied. In Kubernetes such soft limit is defined as **requests** while the hard limit is defined as **limits**.

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its request for that resource specifies. However, a container is not allowed to use more than its resource limit.

Huge pages can be consumed via container level resource requirements using the resource name `hugepages-<size>`, where `<size>` is the most compact binary notation using integer values supported on a particular node. For example, if a node supports 2048KiB and 1048576KiB page sizes, it will expose a schedulable resources `hugepages-2Mi` and `hugepages-1Gi`. Unlike CPU or memory, huge pages do not support overcommit. Note that when requesting hugepage resources, either memory or CPU resources must be requested as well.

A pod may consume multiple huge page sizes in a single pod spec. In this case it must use medium: `HugePages-<hugepagesize>` notation for all volume mounts.

```
apiVersion: v1
kind: Pod
metadata:
  name: huge-pages-example
```

```
spec:  
  containers:  
    - name: example  
      image: fedora:latest  
      command:  
        - sleep  
        - inf  
      volumeMounts:  
        - mountPath: /hugepages-2Mi  
          name: hugepage-2mi  
        - mountPath: /hugepages-1Gi  
          name: hugepage-1gi  
      resources:  
        limits:  
          hugepages-2Mi: 100Mi  
          hugepages-1Gi: 2Gi  
          memory: 100Mi  
        requests:  
          memory: 100Mi  
    volumes:  
      - name: hugepage-2mi  
        emptyDir:  
          medium: HugePages-2Mi  
      - name: hugepage-1gi  
        emptyDir:  
          medium: HugePages-1Gi
```

How to limit Kubernetes resources (CPU & Memory)

Each Container of a Pod can specify one or more of the following:

- spec.containers[].resources.limits.cpu
- spec.containers[].resources.limits.memory
- spec.containers[].resources.limits.hugepages-<size>
- spec.containers[].resources.requests.cpu
- spec.containers[].resources.requests.memory
- spec.containers[].resources.requests.hugepages-<size>

If a Container specifies its own memory limit, but does not specify a memory request, Kubernetes automatically assigns a memory request that matches the limit. Similarly, if a Container specifies its own CPU limit, but does not specify a CPU request, Kubernetes automatically assigns a CPU request that matches the limit.

If you do not specify a CPU limit for a Container, then one of these situations applies:

- The Container has no upper bound on the CPU resources it can use. The Container could use all of the CPU resources available on the Node where it is running.
- The Container is running in a namespace that has a default CPU limit, and the Container is automatically assigned the default limit. Cluster administrators can use a LimitRange to specify a default value for the CPU limit.

How to limit Kubernetes resources (CPU & Memory)

Defining CPU limit

- Limits and requests for CPU resources are measured in cpu units.
- **One cpu**, in Kubernetes, is equivalent to **1 vCPU/Core** for cloud providers and **1 hyperthread** on bare-metal Intel processors.
- Whenever we specify CPU requests or limits, we specify them in terms of CPU cores.
- Because often we want to request or limit the use of a pod to some fraction of a whole CPU core, we can either specify this fraction of a CPU as a decimal or as a millicore value.
- For example, a value of 0.5 represents half of a core.
- It is also possible to configure requests or limits with a millicore value. As there are 1,000 millicores to a single core, we could specify half a CPU as 500 m.
- The smallest amount of CPU that can be specified is 1 m or 0.001.

How to limit Kubernetes resources (CPU & Memory)

Defining memory limit

- Limits and requests for memory are measured in bytes.
- You can express memory as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, K.
- You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki

Name	Bytes	Suffix	Name	Bytes	Suffix
kilobyte	1000	K	kibibyte	1024	Ki
megabyte	1000*2	M	mebibyte	1024*2	Mi
gigabyte	1000*3	G	gibibyte	1024*3	Gi
terabyte	1000*4	T	tebibyte	1024*4	Ti
petabyte	1000*5	P	pebibyte	1024*5	Pi
exabyte	1000*6	E	exbibyte	1024*6	Ei

Memory units supported by Kubernetes

How to limit Kubernetes resources (CPU & Memory)

How pods with resource limits are managed

- When the Kubelet starts a container, the CPU and memory limits are passed to the container runtime, which is then responsible for managing the resource usage of that container.
- If you are using **Docker**, the CPU limit (in milicores) is multiplied by 100 to give the amount of CPU time the container will be allowed to use every 100 ms. If the CPU is under load, once a container has used its quota it will have to wait until the next 100 ms period before it can continue to use the CPU.
- The method used to share CPU resources between different processes running in cgroups is called the **Completely Fair Scheduler or CFS**; this works by dividing CPU time between the different cgroups. This typically means assigning a certain number of slices to a cgroup. If the processes in one cgroup are idle and don't use their allocated CPU time, these shares will become available to be used by processes in other cgroups.
- If memory limits are reached, the container runtime will kill the container (and it might be restarted) with OOM.
- If a container is using more memory than the requested amount, it becomes a candidate for eviction if and when the node begins to run low on memory.

cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

When working with Kubernetes, Out of Memory (OOM) errors and CPU throttling are typical issues in handling cloud applications. When a process runs Out Of Memory (OOM), it's killed since it doesn't have the required resources.

In case CPU consumption is higher than the actual limits, the process will start to be throttled.

Example: Define CPU and Memory limit for containers

```
[root@controller ~]# cat pod-resource-limit.yml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  namespace: cpu-limit
spec:
  containers:
    - name: db
      image: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "password"
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```

```
...
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

```
[root@controller ~]# kubectl create -f pod-resource-limit.yml
pod/frontend created
```

Kubernetes resource quotas

- Resource quotas allow you to place limits on how many resources a particular namespace can use. Depending on how you have chosen to use namespaces, they can give you a powerful way to limit the resources that are used by a particular team, application, or group of applications, while still giving developers the freedom to tweak the resource limits of each individual container.
- A resource quota, defined by a `ResourceQuota` object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that namespace.
- Resource quotas are managed by an admission controller.
- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code 403 FORBIDDEN with a message explaining the constraint that would have been violated.

The following resource types are supported:

Resource Name	Description
<code>limits.cpu</code>	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
<code>limits.memory</code>	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
<code>requests.cpu</code>	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
<code>requests.memory</code>	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
<code>hugepages-<size></code>	Across all pods in a non-terminal state, the number of huge page requests of the specified size cannot exceed this value.
<code>cpu</code>	Same as <code>requests.cpu</code>
<code>memory</code>	Same as <code>requests.memory</code>

Example: Define CPU quota for a namespace

- As quotas will affect all the pods within a namespace, we will start by creating a new namespace using kubectl:

```
[root@controller ~]# kubectl create namespace quota-example  
namespace/quota-example created
```

- Next we will create a YAML file with a CPU quota limit and assign this to the newly created namespace, in this example we have only defined quota limit for CPU for 1 core but you can also add limit for other resource types such as memory, pod count etc.

```
[root@controller ~]# cat ns-quota-limit.yml  
apiVersion: v1  
kind: ResourceQuota  
metadata:  
  name: resource-quota  
  namespace: quota-example  
spec:  
  hard:  
    limits.cpu: 1
```

Kubernetes Networking Model

There are 4 distinct networking problems to address:

- Highly-coupled **container-to-container** communications: this is solved by Pods and localhost communications.
- **Pod-to-Pod** communications
- **Pod-to-Service** communications: this is covered by Services.
- **External-to-Service** communications: this is also covered by Services.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- pods on a node can communicate with all pods on all nodes without NAT
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node.
- pods in the host network of a node can communicate with all pods on all nodes without NAT

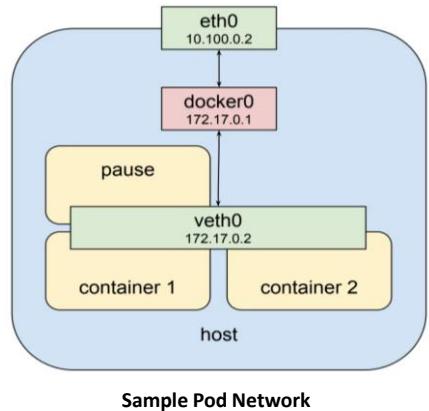
If a pod runs in the host network of the node where the pod is deployed, the pod can use the network namespace and network resources of the node. In this case, the pod can access loopback devices, listen to addresses, and monitor the traffic of other pods on the node.

If the IP address of the pod is the same as that of the node, this indicates that the pod runs in the host network of the node.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  hostNetwork: true
  containers:
    - name: nginx
      image: nginx
```

Kubernetes Networking Model

- In each pod Kubernetes creates a special container for each pod, started with the **pause** command, whose only purpose is to provide a network interface for the other containers.
- **Docker** can start a container and rather than creating a new virtual network interface for it, specifies that it shares an existing interface (*). All other containers created in the pod are created in this way.
- The pause command suspends the current process until a signal is received so these containers do nothing at all except sleep until kubernetes sends them a SIGTERM. Despite this lack of activity the “pause” container is the heart of the pod, providing the virtual network interface that all the other containers will use to communicate with each other and the outside world.



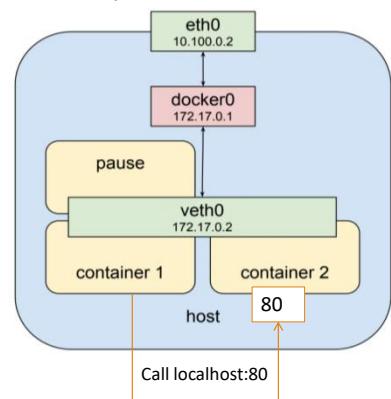
(*) <https://docs.docker.com/engine/reference/run/#network-settings>

Kubernetes deviates from the default Docker networking model (though as of Docker 1.8 their network plugins are getting closer). The goal is for each pod to have an IP in a flat shared networking namespace that has full communication with other physical computers and containers across the network. IP-per-pod creates a clean, backward-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes Networking Model: in node

- Thus Pods can talk to each other via localhost since they connect to the same network.
- Containers within a Pod also have access to shared volumes, which are defined as part of a Pod.
- For external networking, every Pod gets its own IP address.
- No need of explicitly creating links between Pods
- No need to map container ports to host ports.

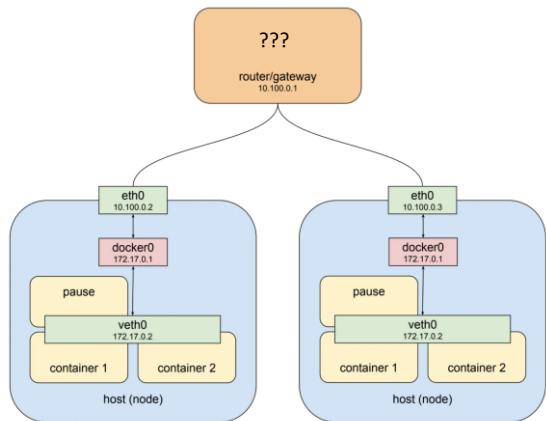
This model is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.



(*) <https://docs.docker.com/engine/reference/run/#network-settings>

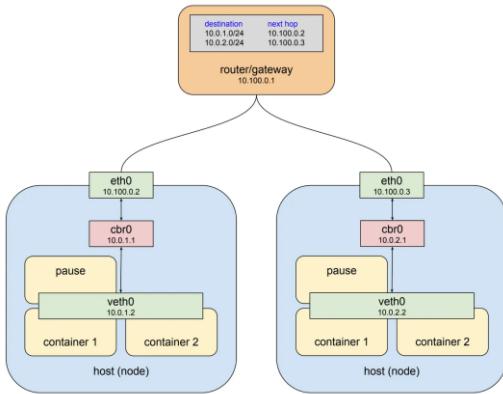
Kubernetes deviates from the default Docker networking model (though as of Docker 1.8 their network plugins are getting closer). The goal is for each pod to have an IP in a flat shared networking namespace that has full communication with other physical computers and containers across the network. IP-per-pod creates a clean, backward-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes Networking Model: out node



- Each pod on a node has its own network namespace. Each pod has its own IP address, and each pod thinks it has a totally normal ethernet device called eth0 to make network requests through.
- Both Pods can legitimately enclose the same 172.17.0.0/24 subnet.
- It is worst case, and it might very well work out that way if you just installed docker and let it do its thing.
- Even if the used network is different this highlights the more fundamental problem which is that one node typically has no idea what private address space was assigned to a bridge on another node, and we need to know that if we're going to send packets to it and have them arrive at the right place.
- Clearly some managing structure, as an **overlay network**, is required

Pod-to-Pod Networking: out node



- Kubernetes has to assigns an overall address space for the bridges on each node, and then assigns the bridges addresses within that space, based on the node the bridge is built on.
- Secondly it adds routing rules to the gateway.
- The changed of name of the bridges from “docker0” to “cbr0,” short for “custom bridge,” indicates that Kubernetes does not use the standard docker bridge.
- Generally it is non necessary to think about how this network functions. When a pod talks with another pod it most often does so through the abstraction of a service.

Since every pod gets a "real" (not machine-private) IP address, pods can communicate without proxies or translations.

The standard networking scenario in Kubernetes is that each pod has a single network interface (eth0). The pod behaves like a single host, which can communicate with every other host on the same node, and usually with all other nodes on the network, depending on the Container Network Interface(CNI) in use.

We set up the Pods to each have their one network namespace so that they believe they have their own Ethernet device and IP address, and they are connected to the root namespace for the Node. Now, we want the Pods to talk to each other through the root namespace, and for this we use a network *bridge*.

When a pod makes a request to the IP address of another node, it makes that request through its own eth0 interface. This tunnels to the node's respective virtual vethX interface.

Every Kubernetes Pod includes an empty pause container, which bootstraps the Pod to establish all of the cgroups, reservations, and namespaces before its individual

containers are created. The pause container image is always present, so the pod resource allocation happens instantaneously as containers are created.

To display the pause container:

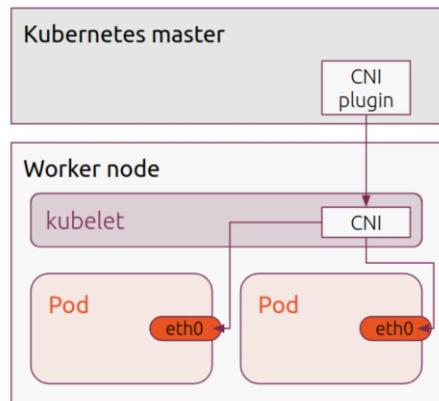
```
docker ps -a | grep -l pause
```

The **veth** devices are virtual Ethernet devices. They can act as tunnels between network namespaces to create a bridge to a physical network device in another namespace, but can also be used as standalone network devices.

From the Pod's perspective, it exists in its own Ethernet namespace that needs to communicate with other network namespaces on the same Node. Thankfully, namespaces can be connected using a Linux [Virtual Ethernet Device](#) or *veth pair* consisting of two virtual interfaces that can be spread over multiple namespaces. To connect Pod namespaces, we can assign one side of the veth pair to the root network namespace, and the other side to the Pod's network namespace. Each veth pair works like a patch cable, connecting the two sides and allowing traffic to flow between them. This setup can be replicated for as many Pods as we have on the machine.

The CNI

- CNI is an initiative of the Cloud-Native Computing Foundation (CNCF), which specifies the configuration of Linux container network interfaces.
- The CNI provides a common API for connecting containers to the outside network.
- **Container Network Interface (CNI)** is a framework for dynamically configuring networking resources. It uses a group of libraries and specifications. The plugin specification defines an interface for configuring the network, provisioning IP addresses, and maintaining connectivity with multiple hosts.



<https://kubernetes.io/docs/concepts/cluster-administration/networking/>

Since every pod gets a "real" (not machine-private) IP address, pods can communicate without proxies or translations.

The standard networking scenario in Kubernetes is that each pod has a single network interface (eth0). The pod behaves like a single host, which can communicate with every other host on the same node, and usually with all other nodes on the network, depending on the Container Network Interface(CNI) in use.

Cluster networking provides communication between different Pods.

The Kubernetes networking model requires that Pod IPs are reachable across the network, but it doesn't specify how that must be done. In practice, this is network specific, but some patterns have been established to make this easier.

Generally, every Node in your cluster is assigned a CIDR block specifying the IP addresses available to Pods running on that Node. Once traffic destined for the CIDR block reaches the Node it is the Node's responsibility to forward traffic to the correct

Pod. The figure illustrates the flow of traffic between two Nodes *assuming that the network can route traffic in a CIDR block to the correct Node.*

The CNI provides a common API for connecting containers to the outside network. As developers, we want to know that a Pod can communicate with the network using IP addresses, and we want the mechanism for this action to be transparent. For example, the CNI plugin developed by AWS tries to meet these needs while providing a secure and manageable environment through the existing VPC, IAM, and Security Group functionality provided by AWS. The solution to this is using elastic network interfaces.

You must use a CNI plugin that is compatible with your cluster and that suits your needs. Different plugins are available (both open- and closed- source) in the wider Kubernetes ecosystem.

The Container Runtime Interface (CRI) manages its own CNI plugins. There are two Kubelet command line parameters to keep in mind when using plugins:

- cni-bin-dir: Kubelet probes this directory for plugins on startup
- network-plugin: The network plugin to use from cni-bin-dir. It must match the name reported by a plugin probed from the plugin directory. For CNI plugins, this is cni.

Kubernetes CNI Plugins

- A CNI plugin must be implemented as an executable that is invoked by the container management system.
- A CNI plugin is responsible for inserting a network interface into the container network namespace (e.g. one end of a *veth* pair) and making any necessary changes on the host (e.g. attaching the other end of the *veth* into a bridge). It should then assign the IP to the interface and setup the routes consistent with the IP Address Management section by invoking appropriate IPAM plugin.
- Kubernetes basic networking provider, **kubenet**, is a simple network plugin that works with various cloud providers but it has many limitations. Advanced features, such as BGP, egress control, and mesh networking, are only available with different CNI plugins.

IPAM: IP address management plugin.

Kubenet is a very basic, simple network plugin, on Linux only. It does not, of itself, implement more advanced features like cross-node networking or network policy. It is typically used together with a cloud provider that sets up routing rules for communication between nodes, or in single-node environments.

Kubenet creates a Linux bridge named `cbr0` and creates a *veth* pair for each pod with the host end of each pair connected to `cbr0`. The pod end of the pair is assigned an IP address allocated from a range assigned to the node either through configuration or by the controller-manager. `cbr0` is assigned an MTU matching the smallest MTU of an enabled normal interface on the host.

Kubernetes CNI Plugins

Common Pod networking add-on plugins:

- **Flannel:** This is a layer 3 IPv4 network between cluster nodes that can use several backend mechanisms such as VXLAN
 - Simple option to configure a layer3 network. It uses Kubernetes API with no external database, and makes use of Kubernetes etcd storage.
 - It implements subnetworks on each host using a **flanneld** agent.
 - It creates one VxLAN for all nodes. Every new node is attached to this it with a veth.
 - Can be layered with Calico.
- **Weave:** A common add-on for CNI enabled Kubernetes cluster
- **Calico:** A layer 3 network solution that uses IP encapsulation and is used in Kubernetes, Docker, OpenStack, OpenShift and others.
 - It uses BGP, BIRD and Felix, an agent installed on each node, supplies end points (external interface), it shares ip table and routes between nodes.
 - confd monitors etcd for BGP configuration changes. It automatically manages BIRD configurations.
 - Calico also provides IP-in-IP encapsulation mode for an overlay network
- **AWS VPC:** A networking plugin that is commonly used for AWS environment

<https://www.golinuxcloud.com/calico-kubernetes/>

The project Calico attempts to solve the speed and efficiency problems that using virtual LANs, bridging, and tunneling can cause. It achieves this by connecting your containers to a vRouter, which then routes traffic directly over the L3 network. This can give huge advantages when you are sending data between multiple data centers as there is no reliance on NAT and the smaller packet sizes reduce CPU utilization.

Basic Objects: Service

Pod to Service Networking

- Pod networking in a cluster is neat stuff, but by itself it is insufficient to enable the creation of **durable** systems.
- Unlike in the non-Kubernetes world, where a sysadmin would configure each client app by specifying the exact IP address or hostname of the server providing the service in the client's configuration files, doing the same in Kubernetes wouldn't work, because pods in kubernetes are ephemeral. They may **come and go at any time** - e.g a pod is removed from a node to make room for other pods, because someone scaled down the number of pods, or because a cluster node has failed. If you use a pod IP address as an endpoint there is no guarantee that the address won't change the next time the pod is recreated, which might happen for any number of reasons.
- Kubernetes assigns an **IP address** to a pod after the pod has been scheduled to a node and **before it is started**. Clients thus can't know the IP address of the server pod up front.
- **Horizontal scaling** means multiple pods may provide the same service. Each of those pods has its own IP address. Clients shouldn't care how many pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of pods. Instead, all those pods should be accessible through a single IP address.

Basic Objects: Service

- This is a quite old problem, and it has a standard solution: run the traffic through a **reverse-proxy/load balancer**. Clients connect to the proxy and the proxy is responsible for maintaining a list of healthy servers to forward requests to. This implies a few requirements for the proxy: it must itself be durable and resistant to failure; it must have a list of servers it can forward to; and it must have some way of knowing if a particular server is healthy and able to respond to requests.
- The kubernetes designers solved this problem in an elegant way that builds on the basic capabilities of the platform to deliver on all three of those requirements, and it starts with a resource type called a **service**.

Basic Objects: Service

In order to have a stable endpoint to communicate with a set of pods, a service is introduced to expose an application. A Service is an abstraction which defines a logical set of Pods and a policy by which to access them.

- The service model relies upon the most basic aspect of services: **discovery**.
- The set of Pods targeted by a Service is usually determined by a **selector**.
- Services ensure that traffic is always **routed** to the appropriate Pod within the cluster, regardless of the node on which it is scheduled.
- Each service exposes an **IP address**, and may also expose a **DNS endpoint**, both of which will never change. Internal or external consumers that need to communicate with a set of pods will use the service's IP address, or its more generally known DNS endpoint.

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a service can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

Basic Objects: Service

Let us go on with the two-Pods example to describe how a kubernetes service enables load balancing across a set of server pods, allowing client pods to operate independently and durably.

To create server pods we can use a Deployment:

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a service can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

- This deployment creates two very simple http server pods that respond on port **8080** with the hostname of the pod they are running in.
- After creating this deployment using **kubectl apply** the pods are running in the cluster.

```

kind: Deployment
apiVersion: apps/v1
metadata:
  name: service-test
spec:
  replicas: 2
  selector:
    matchLabels:
      app: service_test_pod
  template:
    metadata:
      labels:
        app: service_test_pod
    spec:
      containers:
        - name: simple-http
          image: python:2.7
          command: ["/bin/bash"]
          args: ["-c", "echo '<p>Hello from ${hostname}</p>' > index.html; python -m SimpleHTTPServer 8080"]
          ports:
            - name: http
              containerPort: 8080

```

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a service can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

Basic Objects: Service

- A service is a type of kubernetes resource that causes a proxy to be configured to forward requests to a set of pods.
- The set of pods that will receive traffic is determined by the selector, which matches labels assigned to the pods when they were created.
- Once the service is created we can see that it has been assigned an IP address and will accept requests on port 80.

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  selector:
    app: service_test_pod
  ports:
  - port: 80
    targetPort: http
```

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a service can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

In Kubernetes, when creating a service, you have to set the “port” and “targetPort” properties.

A “port” refers to the container port exposed by a pod or deployment, while a “targetPort” refers to the port on the host machine that traffic is directed to.

1. ClusterIP

- ClusterIP is the default and most common service type.
- Kubernetes will assign a cluster-internal IP address to ClusterIP service. This makes the service only reachable within the cluster.
- You cannot make requests to service (pods) from outside the cluster.

- You can optionally set cluster IP in the service definition file.

Use Cases

- Inter service communication within the cluster. For example, communication between the front-end and back-end components of your app.

Basic Objects: Service

DNS for Services and Pods

- Requests can be sent to the service IP directly but it would be better to use a hostname that resolves to the IP address. Kubernetes provides an internal **cluster DNS** that resolves the service name.
- Kubernetes DNS records for Services and Pods. You can contact Services with consistent DNS names instead of IP addresses.
- Kubernetes publishes information about Pods and Services which is used to program DNS. **Kubelet configures Pods' DNS so that running containers can lookup Services by name rather than IP.**
- Services defined in the cluster are assigned DNS names. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain.

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a service can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

In Kubernetes, you can set up a DNS system with two well-supported add-ons: CoreDNS and Kube-DNS. CoreDNS is a newer add-on that became a default DNS server as of Kubernetes v1.12. However, Kube-DNS may still be installed as a default DNS system by certain Kubernetes installer tools.

Both add-ons schedule a DNS pod or pods and a service with a static IP on the cluster and both are named kube-dns in the metadata.name field for interoperability. When the cluster is configured by the administrator or installation tools, the kubelet passes DNS functionality to each container with the --cluster-dns=<dns-service-ip> flag.

When configuring the kubelet, the administrator can also specify the name of a local domain using the flag --cluster-domain=<default-local-domain> .

Kubernetes DNS add-ons currently support forward lookups (A records), port lookups

(SRV records), reverse IP address lookups (PTR records), and some other options. In the following sections, we discuss the Kubernetes naming schema for pods and services within these types of records.

Basic Objects: Service

- An example of a client pod that makes use of the Kubernetes DNS

```
apiVersion: v1
kind: Pod
metadata:
  name: service-test-client
spec:
  restartPolicy: Never
  containers:
    - name: test-client2
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "echo 'GET / HTTP/1.1\r\n\r\n' | nc service-test 80"]
```

You can continue to run the client pod and you'll see responses from both server pods with each getting approximately 50% of the requests.

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a service can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

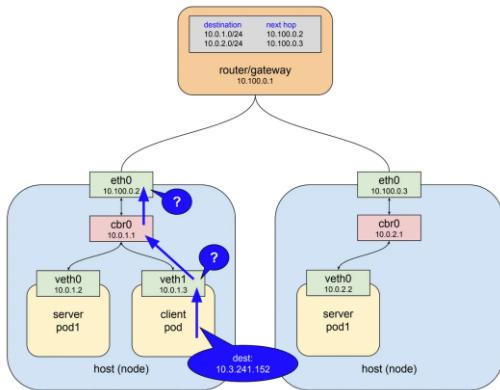
Basic Objects: Service

- The IP that the test service was assigned represents an address on a network that is **not the same as the one the pods are on**. The network specified by this address space is called the “service network.”
- Every service that is of type “**ClusterIP**” will be assigned an IP address on this network.
- There are other types of services, but ClusterIP is the default and it means “the service will be assigned an IP address reachable from any pod in the cluster.” You can see the type of a service by running the **kubectl describe services** command with the service name – e.g. kubectl describe services service-test
- Like the pod network the service network is **virtual**, but it differs from the pod network in some interesting ways. Consider the pod network address range 10.0.0.0/14 in the running example. If you go looking on the hosts that make up the nodes in your cluster, listing bridges and interfaces you’re going to see actual devices configured with addresses on this network. Those are the virtual ethernet interfaces for each pod and the bridges that connect them to each other and the outside world.
- You can examine the routing rules at the gateway that connects all the nodes and you won’t find any routes for this network. The service network **does not exist**, at least not as connected interfaces. However, it work. How did that happen?

Basic Objects: Service

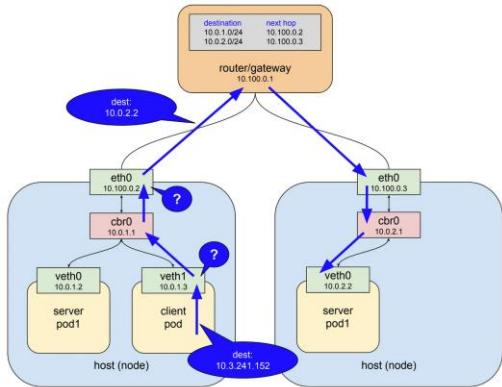
- The IP that the test service was assigned represents an address on a network that is not the same as the one the pods are on. The network specified by this address space is called the “service network.”
- Every service that is of type “ClusterIP” will be assigned an IP address on this network.
- There are other types of services, but ClusterIP is the default and it means “the service will be assigned an IP address reachable from any pod in the cluster.” You can see the type of a service by running the `kubectl describe services` command with the service name – e.g. `kubectl describe services service-test`
- Like the pod network the service network is **virtual**, but it differs from the pod network in some interesting ways. Consider the pod network address range 10.0.0.0/14 in the running example. If you go looking on the hosts that make up the nodes in your cluster, listing bridges and interfaces you’re going to see actual devices configured with addresses on this network. Those are the virtual ethernet interfaces for each pod and the bridges that connect them to each other and the outside world.
- You can examine the routing rules at the gateway that connects all the nodes and you won’t find any routes for this network. The service network **does not exist**, at least not as connected interfaces. However, it work. How did that happen?
- **Like everything in kubernetes a service is just a resource, a record in a central database, that describes how to configure some bit of software to do something.**

Basic Objects: Service



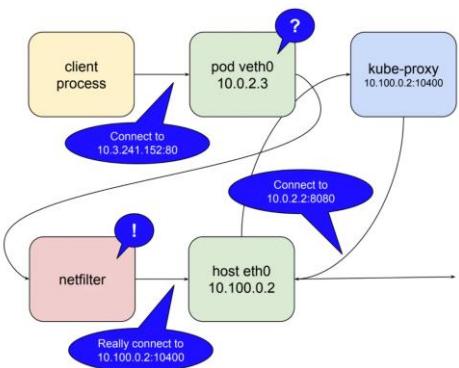
- Assume the client makes an http request to the service using the DNS name service-test. The cluster DNS system resolves that name to the service cluster IP 10.3.241.152, and the client pod ends up creating an http request that results in some packets being sent with that IP in the destination field.
- IP networks are typically configured with routes such that when an interface cannot deliver a packet to its destination because no device with that specified address exists locally it forwards the packet on to its upstream gateway.

Basic Objects: Service



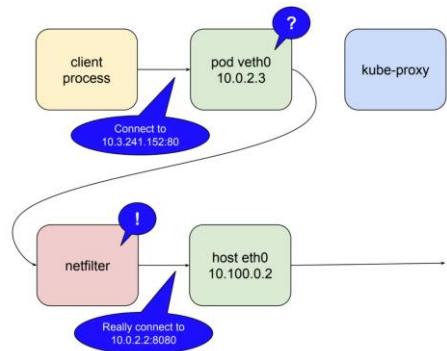
- The host/node ethernet interface in this example is on the network 10.100.0.0/24 and it doesn't know any devices with the address 10.3.241.152 either, so again what would normally happen is the packet would be forwarded out to this interface's gateway, the top level router shown in the drawing. Instead what actually happens is that the packet is snagged in flight and redirected to one of the live server pods.
- The reason of this behavior is on the intervention of the **kube-proxy**.
- Initially kube-proxy was implemented as just such a user-space proxy, but with a twist. A proxy needs an interface, both to listen on for client connections, and to use to connect to backend servers. Kubernetes made use of a linux kernel called **netfilter**, and a user space interface to it called **iptables**.

Basic Objects: Service



- In this mode kube-proxy opens a port (10400 in the example figure) on the local host interface to listen for requests to the test-service, inserts netfilter rules to reroute packets destined for the service IP to its own port, and forwards those requests to a pod on port 8080. That is how a request to 10.3.241.152:80 magically becomes a request to 10.0.2.2:8080.
- Given the capabilities of netfilter all that is required to make this all work for any service is for kube-proxy to open a port and insert the correct netfilter rules for that service, which it does in response to notifications from the master api server of changes in the cluster.
- Unfortunately, user space proxying is expensive due to marshaling packets between user space and kernel space.

Basic Objects: Service



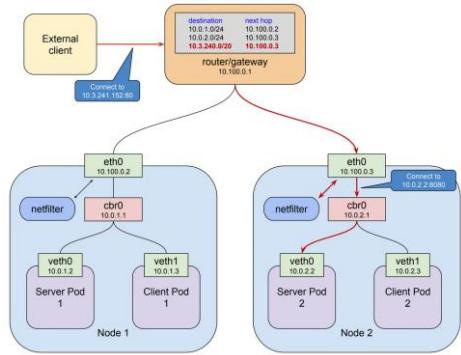
- In kubernetes 1.2 kube-proxy gained the ability to run in iptables mode. In this mode kube-proxy mostly ceases to be a proxy for inter-cluster connections, and instead delegates to netfilter the work of detecting packets bound for service IPs and redirecting them to pods, all of which happens in kernel space.
- In this mode kube-proxy's job is more or less limited to keeping netfilter rules in sync.
- **kube-proxy listens to the master api server for changes in the cluster, which includes changes to services and endpoints.**

Askube-proxy receives updates it uses iptables to keep the netfilter rules in sync. When a new service is created and its endpoints are populated kube-proxy gets the notification and creates the necessary rules. Similarly it removes rules when services are deleted. Health checks against endpoints are performed by the kubelet, another component that runs on every node, and when unhealthy endpoints are found kubelet notifies kube-proxy via the api server and netfilter rules are edited to remove this endpoint until it becomes healthy again.

Basic Objects: Service

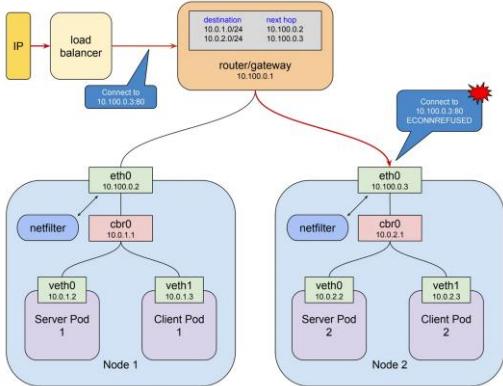
- “ClusterIP” only works for requests that originate inside the cluster, i.e. requests from one pod to another.
- Any mechanism we use to allow external clients to call into our pods has to make use of this same routing infrastructure.
- Those external clients have to end up calling the cluster IP and port, because that is the “front end” for all the machinery that makes it possible not to care where a pod is running at any given time.
- **The cluster IP of a service is only reachable from a node's ethernet interface. Nothing outside the cluster knows what to do with addresses in that range. How can we forward traffic from a publicly visible IP endpoint to an IP that is only reachable once the packet is already on a node?**

Basic Objects: Service



- We could just give clients the cluster IP, maybe assign it a friendly domain name, and then add a route to get those packets to one of the nodes.
- Clients call the cluster IP, the packets follow a route down to a node, and get forwarded to a pod.
- This solution suffers from some serious problems. The first is simply that nodes are ephemeral. They are not *as* ephemeral as pods, but they can be migrated to a new vm, clusters can scale up and down, etc. Routers do not know healthy services from unhealthy. They expect the next hop in the route to be stable and available. If the node becomes unreachable the route will break and stay broken for a significant time in most cases.
- Even if the route were durable if all external traffic passing through a single node is not optimal.

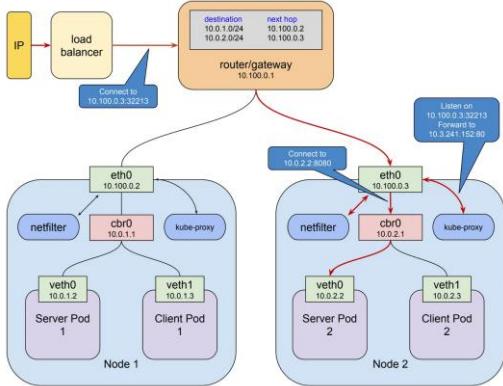
Basic Objects: Service



- To could use a load balancer for distributing client traffic to the nodes in a cluster we need a public IP the clients can connect to, and we need addresses on the nodes themselves to which the load balancer can forward the requests.
- For the reasons discussed we can't easily create a stable static route between the gateway router and the nodes using the service network (cluster IP). The only other addresses available are on the network the nodes' ethernet interfaces are connected to, 10.100.0.0/24 in the example.
- The gateway router already knows how to get packets to these interfaces, and connections sent from the load balancer to the router will get to the right place. But if a client wants to connect to our service on port 80 we can't just send packets to that port on the nodes' interfaces since there is no process listening on 10.100.0.3:80.

The reason why this fails is readily apparent. There is no process listening on 10.100.0.3:80 (or if there is it's the wrong one), and the netfilter rules that we were hoping would intercept our request and direct it to a pod don't match that destination address. They only match the cluster IP on the service network at 10.3.241.152:80. So those packets can't be delivered when they arrive on that interface and the kernel responds with ECONNREFUSED.

Basic Objects: Nodeport Service



- The solution is to create something called a **NodePort**.
- A service of type NodePort is a ClusterIP service with an additional capability: it is reachable at the IP address of the **node** as well as at the assigned cluster IP on the services network.
- The way this is accomplished is pretty straightforward: when Kubernetes creates a **NodePort** service **kube-proxy allocates a port in the range 30000–32767 and opens this port on the eth0 interface of every node** (thus the name “NodePort”). Connections to this port are forwarded to the service’s cluster IP.
- If we create the service above and run `kubectl get svc` service-test we can see the NodePort that has been allocated for it.

2. NodePort

- NodePort service is an extension of ClusterIP service. A ClusterIP Service, to which the NodePort Service routes, is automatically created.
- It exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP.
- NodePort exposes the service on each Node’s IP at a static port (the NodePort). Each node proxies that port into your Service. So, external traffic has access to fixed port on each Node. It means any request to your cluster on that port gets forwarded to the service.
- You can contact the NodePort Service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`.
- Node port must be in the range of 30000–32767. Manually allocating a port to the service is optional. If it is undefined, Kubernetes will automatically assign one.
- If you are going to choose node port explicitly, ensure that the port was not already used by another service.

Use Cases

- When you want to enable external connectivity to your service.
- Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to

expose one or more nodes' IPs directly.

- Prefer to place a load balancer above your nodes to avoid node failure.

The NodePort service exposes a single port on each node in the Kubernetes cluster. Remember, a cluster is just a group of available nodes, so that means when you create this service, you should be able to ping the IP address of the node. NodePorts can be used to make your application accessible on the internet.

<https://www.tkng.io/services/nodeport/>

NodePort builds on top of the ClusterIP Service and provides a way to expose a group of Pods to the outside world. At the API level, the only difference from the ClusterIP is the mandatory service type which has to be set to NodePort, the rest of the values can remain the same.

Whenever a new Kubernetes cluster gets built, one of the available configuration parameters is service-node-port-range which defines a range of ports to use for NodePort allocation and usually defaults to 30000-32767. One interesting thing about NodePort allocation is that it is not managed by a controller.

From the networking point of view, NodePort's implementation is very easy to understand:

- For each port in the NodePort Service, API server allocated a unique port from the service-node-port-range.
- This port is programmed in the dataplane of each Node by the kube-proxy (or its equivalent) – the most common implementations with IPTables, IPVS and eBPF are covered in the Lab section below.
- Any incoming packet matching one of the configured NodePorts will get destination NAT'ed to one of the healthy Endpoints and source NAT'ed (via masquerade/overload) to the address of the incoming interface.
- The reply packet coming from the Pod will get reverse NAT'ed using the connection tracking entry set up by the incoming packet.

Basic Objects: Nodeport Service

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  type: NodePort
  selector:
    app: service_test_pod
  ports:
  - port: 80
    targetPort: http
```

- If we create the service and run `kubectl get svc service-test` we can see the NodePort that has been allocated for it.

```
$ kubectl get svc service-test
NAME      CLUSTER-IP   EXTERNAL-IP PORT(S)      AGE
service-test  10.3.241.152 <none>     80:32213/TCP  1m
```

- In this case our service was allocated the NodePort 32213. This means that we can now connect to the service on either node in the example cluster, at `10.100.0.2:32213` or `10.100.0.3:32213` and traffic will get forwarded to the service. With this piece in place we now have a complete pipeline for load balancing external client requests to all the nodes in the cluster.

Basic Objects: Loadbalancer Service

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  type: LoadBalancer
  selector:
    app: service_test_pod
  ports:
  - port: 80
    targetPort: http
```

- The designers could have stopped there and just let you worry about public IPs and load balancers, and indeed in certain situations such as running on bare metal or in your home lab that is what you are going to have to do. But in environments that support API-driven configuration of networking resources Kubernetes makes it possible to define everything in one place.
- A service of type LoadBalancer has all the capabilities of a NodePort service plus the ability to build out a complete ingress path, assuming you are running in an environment like GCP or AWS that supports API-driven configuration of networking resources.

```
$ kubectl get svc service-test
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
openvpn   10.3.241.52  35.184.97.156  80:32213/TCP  5m
```

- On GCP, for example, this requires the system to create an external IP, a forwarding rule, a target proxy, a backend service, and possibly an instance group. Once the IP has been allocated you can connect to your service through it, assign it a domain name and distribute it to clients.

GCP : Google Cloud Platform

3. LoadBalancer

Currently only implemented in public cloud. So if you're on Kubernetes in Azure or AWS, you will find a load balancer.

LoadBalancer service is an extension of NodePort service. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.

It integrates NodePort with cloud-based load balancers.

It exposes the Service externally using a cloud provider's load balancer.

Each cloud provider (AWS, Azure, GCP, etc) has its own native load balancer implementation. The cloud provider will create a load balancer, which then automatically routes requests to your Kubernetes Service.

Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.

The actual creation of the load balancer happens asynchronously.

Every time you want to expose a service to the outside world, you have to create a new LoadBalancer and get an IP address.

Use Cases

When you are using a cloud provider to host your Kubernetes cluster.

The canonical name record, or CNAME record, is a type of DNS record that maps an alias name to a real or canonical domain name. Typically, CNAME records are used to map a www or mail subdomain to the domain hosting its contents. For example, a CNAME record can map the web address www.example.com to the actual website of the domain example.com.

Service without selector: External Service

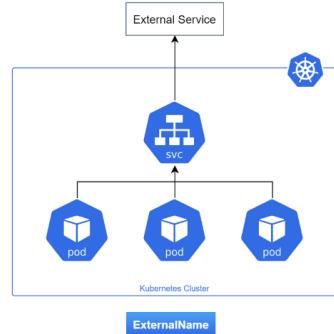
It is used for direct connections based on IP port combinations without an endpoint.

Services most commonly abstract access to Kubernetes Pods thanks to the selector, but when used with a corresponding set of EndpointSlices objects and without a selector, the Service can abstract other kinds of backends, including ones that run outside the cluster.

For example:

- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different Namespace or on another cluster.
- You are migrating a workload to Kubernetes. While evaluating the approach, you run only a portion of your backends in Kubernetes.

In any of these scenarios you can define a Service without specifying a selector to match Pods.



Service without selector: External Service

ExternalName: Up to now, we've talked about services backed by one or more pods running inside the cluster. But cases exist when you have to expose external services through the Kubernetes services feature. Instead of having the service redirect connections to pods in the cluster, you want it to redirect to external IP(s) and port(s). It is based on DNS names and redirection.

```
apiVersion: v1
kind: Service
metadata:
  name: my-database-svc
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

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.

The ExternalName service maps a particular service to a DNS name. The ExternalName Service acts as a proxy, allowing a user to redirect requests to a service sitting outside (or inside) the cluster. Essentially it creates a CNAME record that connects the DNS name to some cluster-local name—that way your pods can leverage that service. This all may feel a little vague at first, so let's look at a real-life example.

Let's say you are migrating all of your applications to Kubernetes. However, that is a hard job, and it would be easier to do it piecemeal. So you retain an external database that your new k8s containers can retrieve data from. However, that database resides *outside* the cluster—so your pods have no idea what it is. We get around this problem by using ExternalName Service.

Basic Objects: Ingress Service

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: "gce"
spec:
  tls:
    - secretName: my-ssl-secret
  rules:
    - host: testhost.com
      http:
        paths:
          - path: /*
            backend:
              serviceName: service-test
              servicePort: 80
```

Internet-to-Service Networking

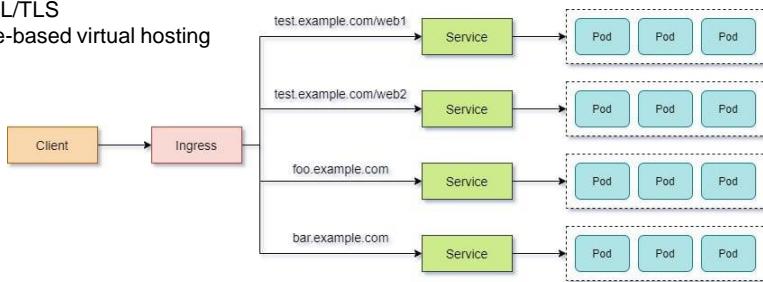
- Services of type LoadBalancer have some limitations. You cannot configure the lb to terminate https traffic. You can't do virtual hosts or path-based routing, so you can't use a single load balancer to proxy to multiple services in any practically useful way. These limitations led to the addition in version 1.2 of a separate kubernetes resource for configuring load balancers, called an Ingress.
- Ingress controllers uses NodePorts to forward traffic to the corresponding service, leveraging the internal load-balancing provided by the iptables rules installed on each Node by kube-proxy.
- When using an Ingress you create your services as type NodePort and let the ingress controller figure out how to get traffic to the nodes. There are ingress controller implementations for GCE load balancers, AWS elastic load balancers, and for popular proxies such as nginx and haproxy.

Google Compute Engine (GCE) is **an infrastructure as a service (IaaS) offering that allows clients to run workloads on Google's physical hardware**. Google Compute Engine provides a scalable number of virtual machines (VMs) to serve as large compute clusters for that purpose.

The Ingress resource

Kubernetes offers an ingress resource and controller that is designed to expose Kubernetes services to the outside world. It can do the following:

- An API object that manages external access to services in a cluster, typically HTTP.
- Traffic routing is controlled by rules defined on the Ingress resource.
- Provisioning of an externally visible URL to your service
- Load balance of traffic
- Terminates SSL/TLS
- Provides name-based virtual hosting



As a customer I just want the hostname/IP address on which I can access the nginx web server and I don't want to remember all these additional Port no. So we use *Kubernetes ingress*.

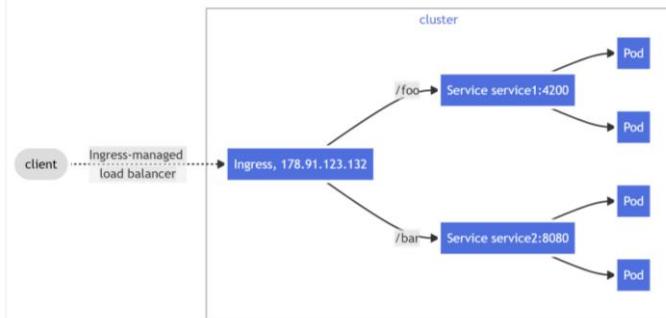
The life of a packet flowing through an Ingress is very similar to that of a LoadBalancer. The key differences are that an Ingress is aware of the URL's path (allowing and can route traffic to services based on their path), and that the initial connection between the Ingress and the Node is through the port exposed on the Node for each service.

With the name based virtual hosting you can host several **domains/websites** on a single machine with a single **IP**. All domains on that server will be sharing a single IP. It's easier to configure than IP based virtual hosting, you only need to configure **DNS** of the domain to map it with its correct IP address and then configure the Ingress to recognize it with the domain names.

A **TLS termination proxy** (or **SSL termination proxy**,^[1] or **SSL offloading**^[2]) is a proxy server that acts as an intermediary point between client and server applications, and is used to terminate and/or establish TLS (or DTLS) tunnels by decrypting and/or

encrypting communications. This is different to **TLS pass-through proxies** that forward encrypted (D)TLS traffic between clients and servers without terminating the tunnel.

The Ingress resource



Ingresses do not work like other Services in Kubernetes. Just creating the Ingress itself will do nothing. You need two additional components:

- **An Ingress controller:** you can choose from many implementations, built on tools such as Nginx or HAProxy.
- **ClusterIP or NodePort Services** for the intended routes.

By using an Ingress it is possible to contact a service by using a URL instead of an IP address.

If you set the Service's type field to NodePort, the Kubernetes master will allocate a port from a range you specify, and each Node will proxy that port (the same port number on every Node) into your Service. That is, any traffic directed to the Node's port will be forwarded on to the service using iptables rules. This Service to Pod routing follows the same internal cluster load-balancing pattern we've already discussed when routing traffic from Services to Pods.

To expose a Node's port to the Internet you use an Ingress object. An Ingress is a higher-level HTTP load balancer that maps HTTP requests to Kubernetes Services. The Ingress method will be different depending on how it is implemented by the Kubernetes cloud provider controller. HTTP load balancers, like Layer 4 network load balancers, only understand Node IPs (not Pod IPs) so traffic routing similarly leverages the internal load-balancing provided by the iptables rules installed on each Node by kube-proxy.

Generally, clusters will not come configured with any pre-existing Ingress controllers. **You must have an Ingress controller to satisfy an Ingress.** Only creating an Ingress

resource has no effect. You'll need to select and deploy one to your cluster. [ingress-nginx](#) is likely the most popular choice, but there are several others, you can get the complete list on [Kubernetes official page](#)

The Ingress resource definition

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - path: /foo
            backend:
              serviceName: service1
              servicePort: 4200
          - path: /bar
            backend:
              serviceName: service2
              servicePort: 8080
```

information
needed to
configure a
load balancer
or proxy server

list of rules
matched against
all incoming
requests. Ingress
resource only
supports rules for
directing HTTP(S)
traffic.

Multi Service Ingress:

- name must be a valid DNS subdomain name
- annotations used to configure some options depending on the Ingress controller
- Each HTTP rule may contain an host (optional), if not specified the rule applies to all inbound HTTP traffic through the Ingress IP address
- Each path has an associated backend defined with a serviceName and servicePort
- A single service can be exposed by specifying an Ingress with no rules and default backend or using a NodePort or LoadBalancer Service.Type

An Ingress needs apiVersion, kind, metadata and spec fields. The name of an Ingress object must be a valid [DNS subdomain name](#). Ingress frequently uses annotations to configure some options depending on the Ingress controller, an example of which is the [rewrite-target annotation](#). Different [Ingress controllers](#) support different annotations. Review the documentation for your choice of Ingress controller to learn which annotations are supported.

The Ingress [spec](#) has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic. If the ingressClassName is omitted, a [default Ingress class](#) should be defined.

Ingress rules

Each HTTP rule contains the following information:

- An optional host. In this example, no host is specified, so the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, foo.bar.com), the rules apply to that host.
- A list of paths (for example, /testpath), each of which has an associated backend defined with a service.name and a service.port.name or service.port.number. Both the host and path must match the content of an incoming request before the load balancer directs traffic to the referenced Service.

- A backend is a combination of Service and port names as described in the [Service doc](#) or a [custom resource backend](#) by way of a [CRD](#). HTTP (and HTTPS) requests to the Ingress that matches the host and path of the rule are sent to the listed backend. A defaultBackend is often configured in an Ingress controller to service any requests that do not match a path in the spec.

Each path in an Ingress is required to have a corresponding path type. Paths that do not include an explicit pathType will fail validation. There are three supported path types:

- ImplementationSpecific: With this path type, matching is up to the IngressClass. Implementations can treat this as a separate pathType or treat it identically to Prefix or Exact path types.
- Exact: Matches the URL path exactly and with case sensitivity.
- Prefix: Matches based on a URL path prefix split by /. Matching is case sensitive and done on a path element by element basis. A path element refers to the list of labels in the path split by the / separator. A request is a match for path p if every p is an element-wise prefix of p of the request path.

Hosts can be precise matches (for example “foo.bar.com”) or a wildcard (for example “*.foo.com”). Precise matches require that the HTTP host header matches the host field. Wildcard matches require the HTTP host header is equal to the suffix of the wildcard rule.

DNS Subdomain Names

Most resource types require a name that can be used as a DNS subdomain name as defined in [RFC 1123](#). This means the name must:

- contain no more than 253 characters
- contain only lowercase alphanumeric characters, '-' or '!
- start with an alphanumeric character
- end with an alphanumeric character

RFC 1123 Label Names

Ingress + Secret resources

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
  - hosts:
    - sslexample.foo.com
    secretName: testsecret-tls
  rules:
  - host: sslexample.foo.com
    http:
      paths:
      - path: /
        backend:
          serviceName: service1
          servicePort: 80
```

- Secure an Ingress by specifying a Secret that contains a TLS private key and certificate.
- Referencing this secret in an Ingress tells the Ingress controller to secure the channel from the client to the load balancer using TLS.

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code.

Example: To create base64 encoded certificate file:

```
apiVersion: v1
data:
  tls.crt:
LS0tLS1CRUdjTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURKakNDQWc2ZOF3SUJBZ0IKQUw2Y
3R2bk9zMzIUTUEwR0NTcUdTSWIzRFFFQkJRVUFNQlI4RkRBU0JnTlYKQkFNVEMyWnZi
eTVpWVhJdVkyOXRNQjRYRFRFNE1USxhOREUxTwpJeU1Gb1hEVEU1TVRJeE5ERTFNa
kl5TUZvdwpGakVV TUJJR0ExVUVBeE1MWm05dkxtSmhjaTVqYjlwd2dnRW INQTBHQ1N
xR1NJYjNEUUVCQVFVQUE0SUJE0F3CmndnRUtBb0lCQVFDbWVsQTNqVy9NZ2REejJNa
zMwbXZ4K2VOSHjkQlIwMEJ4ZUR1VjBjYWVFUGNFa2RmSnk5V28KaTFpSXV1V04vZGV
6UEhyTWMxenBPNGtzbWU5NThRZVFCWjNmVThWeGpRYktmb1JzNnhQUINKZVVSc
VCcWE4SQpUSXpEVVdaUTAwQ2xsa1dOejE4dDYvVjJycWxJd1VvaTVZWHloOVJsaWR4
MjZRaXJBcFFFaXZDY2QzdUExc3AwCkUxRXdIVGxVdzFqSE9Eb3BLZGxaRndmcWhFSHN
```

mYjZvLzJFb1A1MXMwY2JuTId6MHNsUjhhejdzOExVYnhBWnkKQkNQdDY1Z2VhT3hYW
WUxaWhLYzN4SE4wYSsxMXpBYUdDMnpTemdOcEVWeFFJQ3lzdVZld3dNb0FrcHNkdG
EybwpnMnFTaDZQZzRHeFFabzRwejlwN0c2SkFuaflyNENiTEFnTUJBQUdqZHpCMU1C
MEDBMVVkRGdRV0JCU3NBcUzoCkpPS0xZaXNHTkNVRGU4N1VWRkp0UERCR0JnTlZI
U01FUhpBOWdCU3NBcUzoSk9LTFlpc0dOQ1VEZTg3VVZGSnQKUEtFYXBCZ3dGakVVT
UJJR0ExVUVBeE1MWm05dkxtSmhjaTVqYjlyQ0NRQytuTGI1enJOL1V6QU1CZ05WSFJN
RQpCVEFEQVFIL01BMEdu3FHU0liM0RRRUJCUVBQTRJQkFRQU1wcDRLSEtPM2k1N
zR3dzZ3eU1pTExHanpKYXI4Cm8xbHBa3BJR3FMOHVnQWg5d2ZNQWWhsYnhJcWZJRH
lqNWQ3QIZIQlc1UHZweHpKV3pWbmhPOXMrdzdWRTINVHUKWIHSXVRMjdEeExueS
9DVjVQdmJUSTBrcjcwYU9FcGlvTWYyUVUvaTBiN1B2ajJoeEJEMVZTVkd0bHFTSVpqUA
o0VXZQYk1yTWZUWmjka1plbG1SUjJmbW4zK3NTVndrZTrhWXIENVVHNnpBVitjd3BB
bkZWS25VR0d3TkpVMjA4CmQrd3J2UUZ5bi9kcVBKTEdINTkvODY4WjFCcFlxRmJYMit
UVW4yWTEExZ0dkl0J4VmIzeGJ0b29GQkhIVDFLbnIKTTZCVUhEeFnvWVF0VnJWSDRJM
Wh5UGRkdmhPczgwQkQ2K01Dd203OXE2UExacIVKOURGbfI2VTAKLS0tLS1FTkQgQ0V
SVEIGSUNBVEUtLS0tLQo=

tls.key:

LS0tLS1CRUdjTiBSU0EgUFJJVkJURSBLRVktLS0tLQpNSUfB3dJQkFBS0NBuuVBcG5wU
U40MXZ6SUhRODIqSk45SnI4Zm5qUjYzUVVkJkFjWGc3bGRIR25oRDNCskhYCnljdlZxSX
RZaUxybGpmM1hzeng2ekhOYZUdUpMSm52ZWZFSgtBV2QzMVBGY1kwR3luNkViT3
NUMFVpWGxFYXgKQWFtdkNFeU13MUztVU5OQXBaWkZqYzlmtGV2MWRxNnBTTUZL
SXVXRjhvZlVaWW5jZHvrsXF3S1VCSXJ3bhkNwpnTmJLZEJOuk1CMDVWTU5ZeHpnNk
tTbIpXUmNINm9SQjdIMitxUDloS0QrZGJOSeC1elZzOUxKVWZHcys3UEMxCkc4UUDjZ1
FqN2V1WUhtanNWMkh0WW9Tbk44UnpkR3Z0ZGN3R2hndHMwczREYVJGY1VDQXNt
TGxYc01ES0FKS2IKSGJXdhFJTnFrB2VqNE9Cc1VHYU9LyI0T3h1aVFFNFVkdUFteXdJREF
RQUJBb0lCQUMvSitzOEhwZWxCOXJhWgpLNkgvb0ljVTRiNkkwYjA3ZEVOZVpWUnJwS1
ZwWDArTGdqTm1kUTN0K2xzOMXzbmdQWIF4TDFzVHyK0JISzZWCi9kMjJhQ0pheW1
mNmh6cENib21nYWVsT1RpRU13cDZJOEhUMnZjMFhGRzFaSjVMYUlidW0rSTV0MGZI
L3ZYWDEKUzVrY0Mya2JGQ2w3L21lcmZJTVNBQy8vREhpRTUyV1QydElrQk01U2FMV3
p4cDhFa3NwNkxWN3ZwYmR4dGtrTwpkZ1A4QjkWIByck5SdUN5ekRwRukMnhBY2
4yVzNidBqRGpoTjBXdlhTbTERvk9DcXNqOEkrRkxoUzZjemVuCm1MukFZNnpWVGpZV
05TU2J3dTRkbnNmNEIOEdiQkZJajcrdIN5YVNTEZiVGJzY3ZqQ3I1MuSzbWt2bEVMVjg
KaWsvMIJoa0NnWUVBMFpmV2xUTjR2alh2T0Fju1RUU3MwMFhIRWh6QXFjOFpUTE
w2S1d4YkxQVFJNaXBEYklEbQp6b3BiMGNTemxITCtNMVJCY3dqMk5HcUNodXcyczBaN
TQyQVhSZXdtGe1EcWJaWkFQY0UzbERQNW5wNGRpTFRCCIZaMFY4UEExSYjMrd2tUdE
83VThJZlY1aINNdmRDTWtnekI4dU1yQ1VMynhxMXIVUGtLdGpJdThDZ1IFQXkxYWMK
WjEyZC9HWWFpQjDcWpuNONXZE5YdGhFS2dOYUFob21nNIFMzmlKakVLajk3SExKaIF
abFZ0b3kra1RrdTjzAp0Wm1zUi9IU042YmZLbEpckpUWWkzY2E1TGY4a3NxR0Z5Y0x
1MXo3cmN6K1lUaEVWSFLyOVkrVHVWYXRTNkzCkICOGNUQW1ORWIVMIVHR2VKeU
IIIME44Z1VZRXCYzFaMEg2QWlVUNnWUFETDlrgJPeIuxelQvZ1B3Q09GNjQKQjBoel
B3U2VrQXN1WXpueUR6ZURnMIQ2Z2pIc0lEbGh3akNMQzVZL0hPZ0IGNnUyOTImbUR
BaFh6SmM0T2tYMwo4cW5uNGIMa3VPeFhKZ1ZyNnRmUlpNaINaRxpHbXhpbdISVE2
MS9MZGdGVTg3WExYWHdmaTZPdW80cUVhNm9YCjhCRMZxOWRVcXB4bEVLY2Y1N3

```
JsK1FLQmdGbjVSaFc2NS9oU0diVlhFWVZQU0pSOW9FK3IkRjcrd3FvaGRoOVQKekQ0UTZ6THBCQXJITkFYeDZZK0gxM3pFVIUzVEwrTTJUM1E2UGFHZ2Rpa2M5TIRPdkE3aU1nVTRvRXMzMENPWQpoR2x3bUhEc1B6YzNsWXIsU0NvYVVPeDJ2UFFwN2VJSndoU25PVVBwTVdKWi80Z2pZZTFjZmNseTFrQTJBR0x3CIJ1STIBb0dCQU14aGFJSUdwTGdmcHk0K24rai9BSWhJUUhLZFRCNVBqaGx0WWhqZittK011UURwk21OeTVMbzEKT0FRc0Q0enZ1b3VxeHlmQIFQZIIYT hvcm4vTDE3WIjyc3ISNHlhS1M3cDVQYmJKQINlcTc5Z0g5ZUNIQkxMbQo0aThCUFh0K0NmWktMQzg3NTNHSHVpOG91V25scUZ0NGxMQUlWaGJZQmtUbURZSWo4Q0NaCi0tLS0tRU5EIFJTQSBQUkIWQVRFIetFWS0tLS0tCg==
```

kind: Secret

metadata:

 name: test-tls

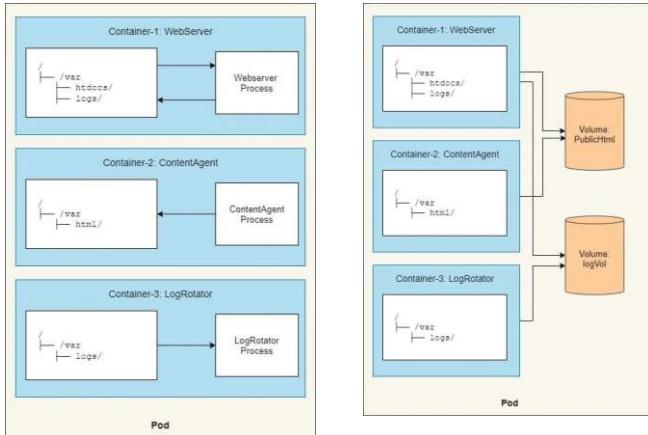
 namespace: default

type: kubernetes.io/tls

Kubernetes Volumes

- Assume that a container within a Pod gets restarted (either because the process died or because the liveness probe signaled to Kubernetes that the container wasn't healthy anymore). You'll realize that the new container **will not see anything** that was written to the filesystem by the previous container, even though the newly started container runs in the same pod.
- In certain scenarios you want the new container to continue where the last one finished, such as when restarting a process on a physical machine. You may not need (or want) the whole filesystem to be persisted, but you do want to **preserve certain directories** that hold actual data.
- Kubernetes provides this by defining **storage volumes**. They are not top-level resources like pods, but are instead defined as a part of a pod and share the same lifecycle as the pod. This means a volume is created when the pod is started and is destroyed when the pod is deleted. Because of this, a volume's contents will persist across container restarts. After a container is restarted, the new container can see all the files that were written to the volume by the previous container. Also, if a pod contains multiple containers, the volume can be used by all of them at once.

Kubernetes Volumes



- Volumes are not a standalone Kubernetes object and cannot be created or deleted on their own.
- A volume is available to all containers in the pod, but it must be mounted in each container that needs to access it.
- In each container, you can mount the volume in any location of its filesystem

Let us take this example to get a more clear understanding:

One container runs a web server that serves HTML pages from the /var/htdocs directory and stores the access log to /var/logs. The second container runs an agent that creates HTML files and stores them in /var/html. The third container processes the logs it finds in the /var/logs directory (rotates them, compresses them, analyzes them, or whatever).

These 3 containers within the Pod are working just fine but they all are performing read and write operation to their own file system, even though all of them are using /var directory. So it doesn't make sense to have such architecture, we need our logRotator container to process the logs from other containers and similarly contentAgent will write new content inside /var/html.

But by mounting the same volume into two containers, they can operate on the same files. In your case, you're mounting two volumes in three containers. By doing this, your three containers can work together and do something useful.

First, the pod has a volume called publicHtml. This volume is mounted in the WebServer container at /var/htdocs, because that's the directory the web server

serves files from. The same volume is also mounted in the ContentAgent container, but at /var/html, because that's where the agent writes the files to. By mounting this single volume like that, the web server will now serve the content generated by the content agent. Similarly, the pod also has a volume called logVol for storing logs. This volume is mounted at /var/logs in both the WebServer and the LogRotator containers. Note that it isn't mounted in the ContentAgent container. The container cannot access its files, even though the container and the volume are part of the same pod.

Kubernetes Volumes

Different volume types in Kubernetes

Volume Type	Storage Provider
emptyDir	Localhost
hostPath	Localhost
glusterfs	GlusterFS cluster
downwardAPI	Kubernetes Pod information
nfs	NFS server
awsElasticBlockStore	Amazon Web Service Amazon Elastic Block Store
gcePersistentDisk	Google Compute Engine persistent disk
azureDisk	Azure disk storage
projected	Kubernetes resources; currently supports secret, downwardAPI, and configMap
secret	Kubernetes Secret resource
vSphereVolume	vSphere VMDK volume
gitRepo	Git repository

Now we know that Kubernetes introduces volume, **which lives with a Pod across a container life cycle**. It supports various types of volumes, including popular network disk solutions and storage services in different public clouds.

Storage providers are required when you start to use volume in Kubernetes, except for emptyDir, which will be erased when the Pod is removed. For other storage providers, folders, servers, or clusters have to be built before using them in the Pod definition.

Volumes are defined in the volumes section of the pod definition with unique names. Each type of volume has a different configuration to be set. Once you define the volumes, you can mount them in the volumeMounts section in the container specs.volumeMounts.name and volumeMounts.mountPath are required, which indicate the name of the volumes you defined and the mount path inside the container, respectively.

Kubernetes Volumes

```
[root@controller ~]# cat shared-volume-emptyDir.yml
apiVersion: v1
kind: Pod
metadata:
  name: shared-volume-emptyDir
spec:
  containers:
    - name: alpine1
      image: alpine
      command: ["/bin/sleep", "999999"]
      volumeMounts:
        - mountPath: /alpine1
          name: data
    - name: alpine2
      image: alpine
      command: ["/bin/sleep", "999999"]
      volumeMounts:
        - mountPath: /alpine2
          name: data
  volumes:
    - name: data
      emptyDir: {}
```

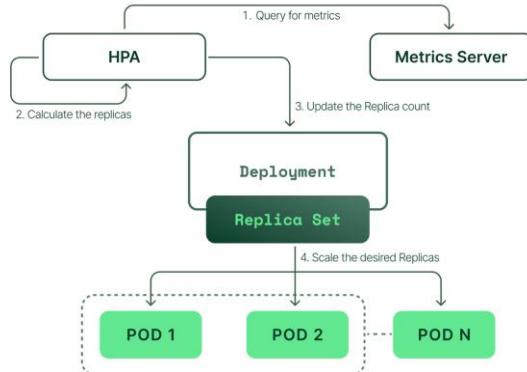
- By this configuration file, we can run a Pod with two containers, alpine1 and alpine2. Both containers will sleep for 999999 seconds. The configuration defines a volume named 'data' in the 'volumes' section. This volume is mounted under the path '/alpine1' in the alpine1 container and '/alpine2' in the alpine2 container.

Kubernetes Horizontal Pod Autoscaler (HPA)

- An HPA object watches the resource consumption of pods that are managed by a controller (Deployment, ReplicaSet, or StatefulSet) at a given interval and controls the replicas by **comparing the desired target of certain metrics with their real usage**.
- For instance, suppose that we have a Deployment controller with two pods initially, and they are currently using **1,000 m** of CPU on average while we want the CPU percentage to be **200 m** per pod. The associated HPA would calculate how many pods are needed for the desired target with **$2 * (1000 \text{ m} / 200 \text{ m}) = 10$** , so it will adjust the replicas of the controller to 10 pods accordingly. Kubernetes will take care of the rest to schedule the eight new pods.

Kubernetes Horizontal Pod Autoscaler (HPA)

- HPA continuously monitors the metrics server for resource usage.
- Based on the collected resource usage, HPA will calculate the desired number of replicas required.
- Then, HPA decides to scale up the application to the desired number of replicas.
- Finally, HPA changes the desired number of replicas.
- Since HPA is continuously monitoring, the process repeats from Step 1.



In Kubernetes, a *HorizontalPodAutoscaler* automatically updates a workload resource (such as a [Deployment](#) or [StatefulSet](#)), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more [Pods](#). This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

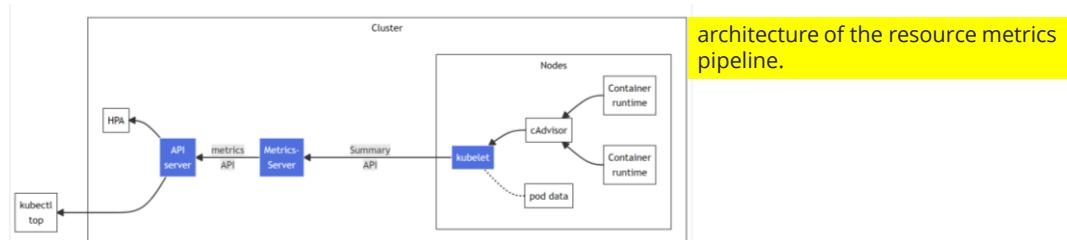
Horizontal pod autoscaling does not apply to objects that can't be scaled (for example: a [DaemonSet](#).)

The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a [controller](#). The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes [control plane](#), periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other

custom metric you specify.

Kubernetes Horizontal Pod Autoscaler (HPA)

- It is **mandatory** that you have a metrics server installed and running on your Kubernetes Cluster. The metrics server will provide the metrics through the [Metrics API](#). Horizontal Pod Autoscaler uses this API to collect metrics.
- For Kubernetes, the Metrics API offers a basic set of metrics to support automatic scaling and similar use cases. **This API makes information available about resource usage for node and pod**, including metrics for CPU and memory. If you deploy the Metrics API into your cluster, clients of the Kubernetes API can then query for this information, and you can use Kubernetes' access control mechanisms to manage permissions to do so.



The architecture components, from right to left in the figure, consist of the following:

- cAdvisor: Daemon for collecting, aggregating and exposing container metrics included in Kubelet.
- kubelet: Node agent for managing container resources. Resource metrics are accessible using the /metrics/resource and /stats kubelet API endpoints.
- Summary API: API provided by the kubelet for discovering and retrieving per-node summarized stats available through the /stats endpoint.
- metrics-server: Cluster addon component that collects and aggregates resource metrics pulled from each kubelet. The API server serves Metrics API for use by HPA, VPA, and by the kubectl top command. Metrics Server is a reference implementation of the Metrics API.
- Metrics API: Kubernetes API supporting access to CPU and memory used for workload autoscaling. To make this work in your cluster, you need an API extension server that provides the Metrics API.

Kubernetes Horizontal Pod Autoscaler (HPA)

- Metrics Server can be installed either directly from YAML manifest or via the official Helm chart. To install the latest Metrics Server release from the components.yaml manifest, run the following command.

```
# kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

- You could also download the file, change it and deploy through the `kubectl apply` command.

Kubernetes Horizontal Pod Autoscaler (HPA)

- HPA runs intermittently (it is not a continuous process). The interval is set by the `--horizontal-pod-autoscaler-sync-period` parameter to the `kube-controller-manager` (and the default interval is 15 seconds).
- Once during each period, the controller manager queries the resource utilization against the metrics specified in each `HorizontalPodAutoscaler` definition. The controller manager finds the target resource defined by the `scaleTargetRef`, then selects the pods based on the target resource's `.spec.selector` labels, and obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).
 - For per-pod resource metrics (like CPU), **the controller fetches the metrics from the resource metrics API** for each Pod targeted by the `HorizontalPodAutoscaler`. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each Pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted Pods, and produces a ratio used to scale the number of desired replicas.

<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

Kubernetes Horizontal Pod Autoscaler (HPA)

- From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

- For example, if the current metric value is 200m, and the desired value is 100m, the number of replicas will be doubled, since $200.0 / 100.0 == 2.0$. If the current value is instead 50m, you'll halve the number of replicas, since $50.0 / 100.0 == 0.5$. The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a globally-configurable tolerance, 0.1 by default).

<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

Kubernetes Horizontal Pod Autoscaler (HPA)

- The Horizontal Pod Autoscaler is an **API resource** in the Kubernetes **autoscaling** API group. The current stable version can be found in the **autoscaling/v2** API version which includes support for scaling on memory and custom metrics.
- When defining the **pod specification** the resource requests like **cpu** and **memory** should be specified. This is used to determine the resource utilization and used by the HPA controller to scale the target up or down. To use resource utilization based scaling specify a metric source like this:

```
type: Resource  
resource:  
  name: cpu  
target:  
  type: Utilization  
  averageUtilization: 60
```

With this metric the HPA controller will keep the average utilization of the pods in the scaling target at 60%.

<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

Kubernetes Horizontal Pod Autoscaler (HPA)

- If you use the v2 HorizontalPodAutoscaler API, you can use the `behavior` field (see the API reference) to configure separate scale-up and scale-down behaviors. You specify these behaviours by setting `scaleUp` and / or `scaleDown` under the `behavior` field.
- You can specify a *stabilization window* that prevents [flapping](#) the replica count for a scaling target.
- One or more scaling policies can be specified in the `behavior` section of the spec. When multiple policies are specified the policy which allows the highest amount of change is the policy which is selected by default.

```
behavior:  
scaleDown:  
stabilizationWindowSeconds: 300  
policies:  
- type: Percent  
value: 100  
periodSeconds: 15  
scaleUp:  
stabilizationWindowSeconds: 0  
policies:  
- type: Percent  
value: 100  
periodSeconds: 15  
- type: Pods  
value: 4  
periodSeconds: 15  
selectPolicy: Max
```

Default behavior

<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

For scaling down the stabilization window is 300 seconds (or the value of the `--horizontal-pod-autoscaler-downscale-stabilization` flag if provided). There is only a single policy for scaling down which allows a 100% of the currently running replicas to be removed which means the scaling target can be scaled down to the minimum allowed replicas. For scaling up there is no stabilization window. When the metrics indicate that the target should be scaled up the target is scaled up immediately. There are 2 policies where 4 pods or a 100% of the currently running replicas will be added every 15 seconds till the HPA reaches its steady state.

Kubernetes Horizontal Pod Autoscaler (HPA)

- HorizontalPodAutoscaler, like every API resource, is supported in a standard way by `kubectl`. You can create a new autoscaler using `kubectl create` command. You can list autoscalers by `kubectl get hpa` or get detailed description by `kubectl describe hpa`. Finally, you can delete an autoscaler using `kubectl delete hpa`.
- In addition, there is a special `kubectl autoscale` command for creating a HorizontalPodAutoscaler object. For instance, executing `kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for ReplicaSet foo, with target CPU utilization set to 80% and the number of replicas between 2 and 5.

Example: Autoscaling applications using HPA for CPU Usage

- We could add some more configuration to this components.yaml file of the Metric server, as the new entries as highlighted:

```
...  
template:  
metadata:  
  labels:  
    k8s-app: metrics-server  
spec:  
  hostNetwork: true ## add this line  
  containers:  
    - args:  
        - --cert-dir=/tmp  
        - --secure-port=4443  
        - --kubelet-insecure-tls ## add this line  
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname  
        - --kubelet-use-node-status-port  
      image: k8s.gcr.io/metrics-server/metrics-server:v0.4.2  
...  
  
  --kubelet-preferred-address-types - The priority of node address types used when determining an address for connecting to a particular node  
  --kubelet-insecure-tls - Do not verify the CA of serving certificates presented by Kubelets.  
  hostNetwork - Enable hostNetwork mode
```

Next we will deploy this manifest using kubectl apply command:

```
[root@controller ~]# kubectl apply -f components.yaml
```

Example: Autoscaling applications using HPA for CPU Usage

Create deployment

```
[root@controller ~]# cat nginx-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
labels:
  type: dev
name: nginx-deploy
```

```
[root@controller ~]# kubectl create -f nginx-deploy.yaml
deployment.apps/nginx-deploy created
```

```
spec:
replicas: 1
selector:
matchLabels:
  type: dev
template:
  metadata:
    labels:
      type: dev
  spec:
    containers:
      - image: nginx
        name: nginx
      ports:
        - containerPort: 80
    resources:
      limits:
        cpu: 500m
      requests:
        cpu: 200m
```

Example: Autoscaling applications using HPA for CPU Usage

Create deployment

- Create a HorizontalPodAutoscaler resource for this Deployment that will be able to auto-scale the Deployment from one to five replicas, with a CPU utilization of 10%, in imperative form:

```
[root@controller ~]# kubectl autoscale deployment nginx-deploy --cpu-percent=10 --min=1 --max=5
horizontalpodautoscaler.autoscaling/nginx-deploy autoscaled
```

- Alternatively we could have also used following YAMLfile to create this HPA resource:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-deploy
spec:
  minReplicas: 1
  maxReplicas: 5
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deploy ## Name of the deployment
  targetCPUUtilizationPercentage: 10
```

Kubeadm

- Kubeadm is a tool built to provide kubeadm init and kubeadm join as best-practice "fast paths" for creating Kubernetes clusters.
- kubeadm performs the actions necessary to get a minimum viable cluster up and running. By design, it cares only about bootstrapping, not about provisioning machines. Likewise, installing various nice-to-have addons, like the Kubernetes Dashboard, monitoring solutions, and cloud-specific addons, is not in scope.
- Instead, higher-level and more tailored tooling are expected to be built on top of kubeadm, and ideally, using kubeadm as the basis of all deployments will make it easier to create conformant clusters.



kubeadm init

Run this command in order to set up the Kubernetes control plane

Sources

- <https://kubernetes.io/docs/home/>
- <https://www.golinuxcloud.com/kubernetes-tutorial/>
- https://linuxacademy.com/site-content/uploads/2019/04/Kubernetes-CheatSheet_07182019.pdf •
- <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>
- <https://vimeo.com/245778144/4d1d597c5e>
- <https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/>
- <https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727>
- <https://medium.com/google-cloud/understanding-kubernetes-networking-services-f0cb48e4cc82>
- <https://github.com/containernetworking/cni>
- <https://github.com/containernetworking/cni/blob/master/SPEC.md>
- <https://www.objectif-libre.com/en/blog/2018/07/05/k8s-network-solutions-comparison/>
- <https://chrislovecnm.com/kubernetes/cni/choosing-a-cni-provider/>
- <https://www.contino.io/insights/kubernetes-is-hard-why-eks-makes-it-easier-for-network-and-security-architects>
- <https://medium.com/flant-com/calico-for-kubernetes-networking-792b41e19d69>
- <https://newrelic.com/blog/how-to-relic/how-to-use-kubernetes-volumes>

Virtual Extensible LAN

GIANLUCA REALI



Overview

Virtualization of servers causes challenges for Datacenter networks with traditional three-layer architecture

VXLAN can respond to these challenges.

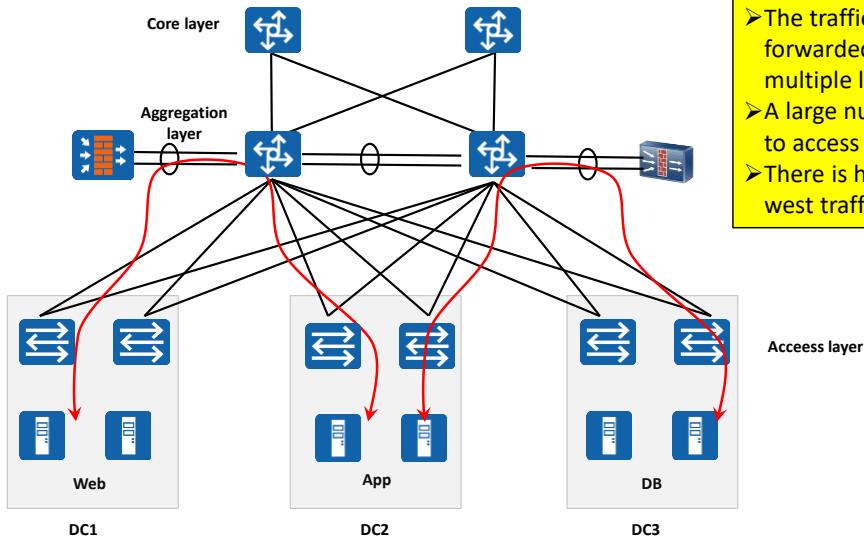
Server virtualization greatly reduces IT construction and operations and maintenance (O&M) costs and improves service deployment flexibility.

Virtual machines (VMs) on a traditional data center network can only seamlessly migrate on Layer 2. If VMs migrate across a Layer 3 network, services will be interrupted.

The Virtual eXtensible Local Area Network (VXLAN) technology is introduced to improve VM migration flexibility, so that the large number of tenants are not limited by IP address changes and broadcast domains.



Challenge: Low Latency Requirements of Compute Nodes



A large number of VMs are deployed on a physical server, causing huge traffic concurrency.

The data traffic model is converted from the traditional north-south traffic to east-west traffic.

A large amount of many-to-one and many-to-many east-west traffic exists on the network.

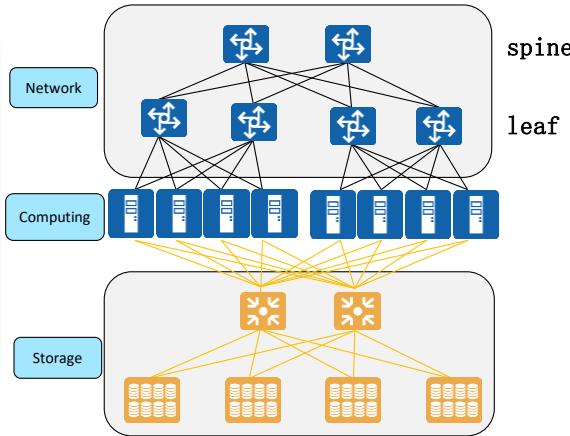
The devices on the access and aggregation layers need to provide high processing capabilities.



Data Center Basic Concepts and Features

What Is a Data Center

- Data Center (DC)
- Core of the enterprise IT system
- Massive data computing, switching, and storage center
- Computing environment for key information, services, and applications
- Environment for centralized management and control of various data, applications, physical devices, and virtual devices



The best example of a switching fabric topology is the Spine-and-Leaf, which is commonly used as an underlay network.

A data center has four features: reliable, flexible, environmentally-friendly, and efficient.

A Data Center (DC) is a collection of complete and complex systems consisting of the computing system, auxiliary devices (such as the communications and storage system), data communication system, environment control devices, monitoring devices, and various security devices.

A data center stores, processes, transmits, switches, and manages information in a centralized mode within a physical space.

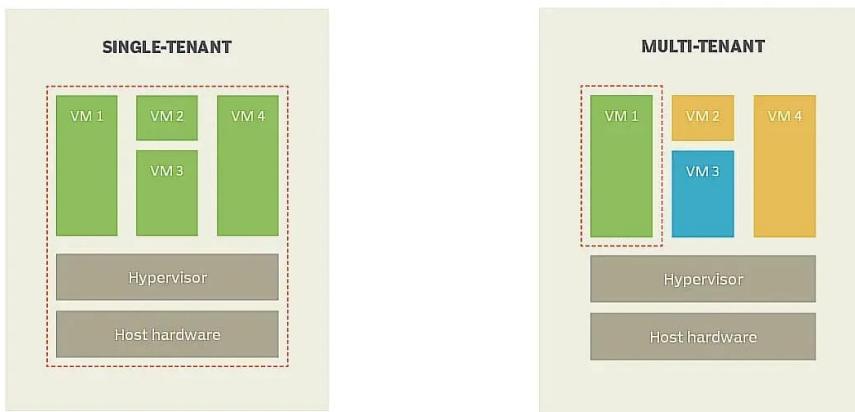
Key devices in a data center include servers, network devices, and storage devices.

Power supply system, air conditioning system, cabinets, fire protection system, monitoring system, and other systems that affect operating environment of the key devices are key physical facilities.

An Internet Data Center (IDC) is a center for data storage and processing on the Internet that involves the most intensive data exchange.



Architectural Alternatives



Tenancy in cloud computing refers to the sharing of computing resources in a private or public environment that is isolated from other users and kept secret. Tenancy in SaaS is divided into two types: single-tenant SaaS and multi-tenant SaaS.

<https://www.cloudzero.com/blog/single-tenant-vs-multi-tenant/>

In SaaS (Software-as-a-Service), a single-tenant architecture is where a single instance of the software application, and its supporting infrastructure, serves a single customer (tenant).

The single instance of the software runs on [a dedicated cloud server](#) for just this one customer. So there is no sharing it with any other customer of the SaaS provider.

A multi-tenant architecture is where a single software instance and its supporting infrastructure are shared among two or more customers (tenants) at the same time. While each tenant's data is isolated from the other tenants', each customer shares a single database, software application, and SaaS server with the others.



meh leggi se vuoi capire ma meh

Possible challenge in a small/medium datacenter

Assume a datacenter with **20 racks** including **48 physical servers** connected to each access switch (TOR). Each of these servers includes **five different tenants** (dedicated virtualized environments) with their own virtual routing (VRF). **One tenant consists of three broadcast domains (VLAN)**, e.g. Presentation, Application and Database, each with two virtual machines backing up each other. The customer manages his own tenant and can define the VLANs IDs, the mac addresses of virtual machines and the IP address architecture.

The mobility of virtual machines is unlimited.

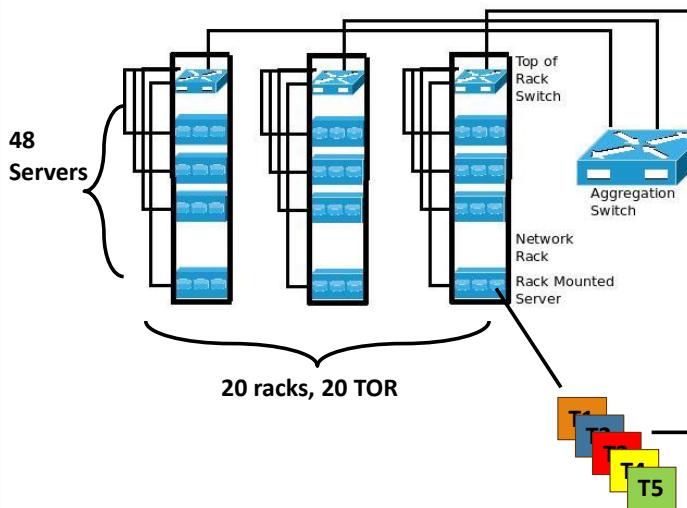
Virtual Routing and Forwarding (VRF) IP Technology allows users to configure multiple routing table instances to simultaneously co-exist within the same router.



meh ancora, è un esempio

Be quantitative!

Top-Of-Rack (ToR) - Network Connectivity Architecture



- # Physical servers: $20 \text{ (ToR)} \times 48 \text{ (port per ToR)} = 960$
- # VMs - Mac addresses - ARP entries:
 $960 \text{ (servers)} \times 30 \text{ (VM per server)} = 28800$
- # Tenants / VRF: $960 \text{ hosts} \times 5 \text{ tenants} = 4800$
- # Broadcast Domains: $5 \text{ (tenant per server)} \times 3 \text{ (VLANs per tenant)} \times 960 \text{ (servers)} = 14400$

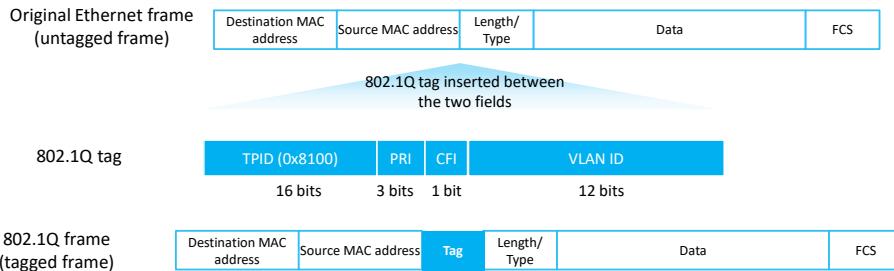
UNLIMITED MOBILITY REQUIRED !!!

Virtual Routing and Forwarding (VRF) IP Technology allows users to configure multiple routing table instances to simultaneously co-exist within the same router.



Summary of identified issues:

- **The isolation capability of traditional networks is limited.** VLAN is a mainstream network isolation technology. However, the VLAN tag field defined in IEEE 802.1Q has **only 12 bits** and can identify only a **maximum of 4096 VLANs**, which cannot meet the isolation requirements of a large number of tenants.





Summary of identified issues:

- **The VM migration scope is limited.** To ensure services continuity during VM migration, the IP addresses and MAC addresses of the VMs must remain unchanged before and after the migration. This means that **VM migration must occur in one Layer 2 domain**. However, VM migration in Layer 2 domains of traditional data center networks is limited to a small scope.



VXLAN solutions

- Limitation of network isolation capabilities: **24 bits isolation identifier**, similar to VLAN ID, which is called VXLAN Network ID (VNI).
- Limitation of the VM migration scope: VXLAN encapsulates Ethernet packets in IP packets and transmits them over routes on a network to construct a large Layer 2 network. Therefore, **VM migration is not restricted** by the network architecture.

A VXLAN network is a virtual Layer 2 network constructed on a Layer 3 network to enable communication of hosts at Layer 2. Compared with VLAN, VXLAN has higher flexibility and scalability, and addresses the following issues:

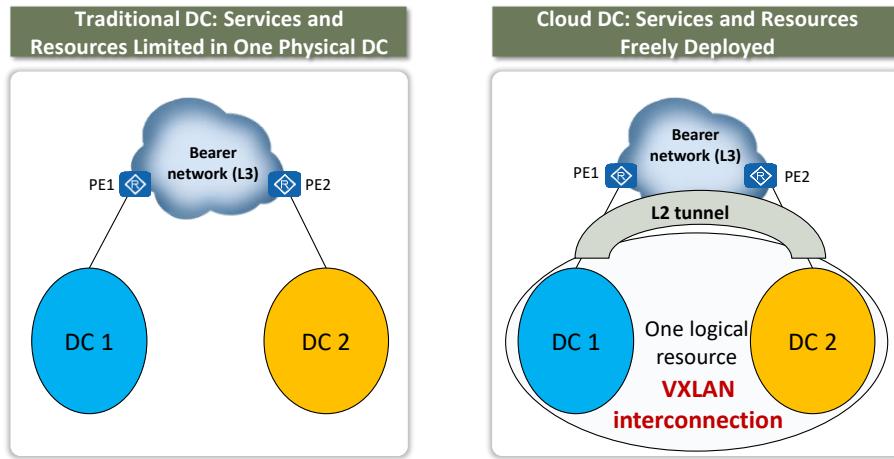
Limitation of the VM scale imposed by network specifications; Data packets sent by VMs are encapsulated in IP data packets. The network is only aware of the encapsulated network parameters. This greatly reduces the number of MAC address entries required by large Layer 2 networks.

Limitation of network isolation capabilities VXLAN technology extends the number of isolation identifier bits to 24 bits, which greatly increases the number of tenants that can be isolated. Theoretically, up to 16 million tenants can be isolated. VXLAN introduces a network identifier similar to VLAN ID, which is called VXLAN Network ID (VNI). Each VNI has 24 bits and can identify up to 16 million tenants, meeting the isolation requirements of a large number of tenants.

Limitation of the VM migration scope imposed by the network architectureVXLAN encapsulates Ethernet packets in IP packets and transmits them over routes on a network to construct a large Layer 2 network. Therefore, VM migration is not restricted by the network architecture. In addition, a routed network has good scalability, self-healing capability, and load balancing capability.



Large Layer 2 Interconnection Among Multiple DCs - VXLAN



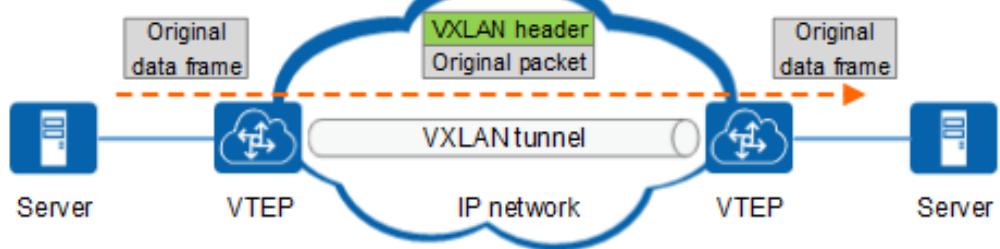
In this early stage, VM management and migration are completed on physical networks. Therefore, the east-west traffic in a DC is mainly Layer 2 traffic. To extend the scale of a Layer 2 physical network and improve link utilization, large Layer 2 technologies, such as Transparent Interconnection of Lots of Links (TRILL) and Shortest Path Bridging (SPB) are developed.

As the virtualized DC scale expands and cloud-based management becomes popular, VM management and migration on physical networks can no longer meet virtualization requirements. As a result, overlay technologies such as VXLAN and Network Virtualization using Generic Routing Encapsulation (NVGRE) are developed.

In the overlay solution, east-west traffic on a physical network is gradually changed from Layer 2 traffic to Layer 3 traffic. In addition, the overlay solution changes the network topology from physical Layer 2 to logical Layer 2 and provides the logical Layer 2 division and management functions, better matching requirements of multiple tenants. Overlay technologies such as VXLAN and NVGRE use MAC-in-IP encapsulation to solve limitations of physical networks, including the limit on the number of VLANs and MAC address entries supported by access switches. These technologies also provide a unified logical network management tool to enable policy migration during VM migration, greatly reducing network dependency during virtualization. These technologies attract major concern in network virtualization.



VXLAN network model



- Overlay network: VXLAN
- Underlay network: IP

Below IP, Spine-and-Leaf uses two layers:

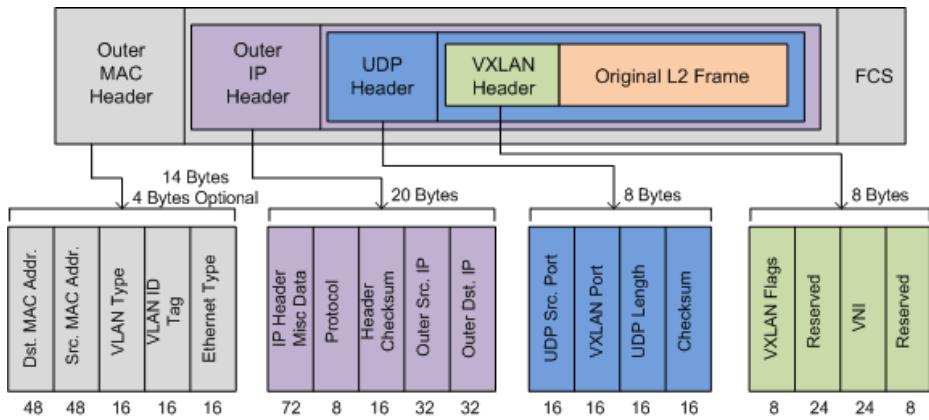
- **Spine:** The spine layer switches are only used to pass traffic through leaf switches. They are not aware of VxLAN.
- **Leaf:** The Leaf layer of switches interconnect the spine and the end points. The leaf layer switches create the VxLAN tunnels, encapsulation, and maps VLANs to VNI. The leaf switches that perform VxLAN functions are known as VTEPs (VxLAN Tunnel Endpoints)

To address the preceding problems, overlay network technologies are gradually evolved to meet the network capability requirements of cloud computing. There are multiple overlay technologies, such as Virtual eXtensible Local Area Network (VXLAN), Network Virtualization using Generic Routing Encapsulation (NVGRE), and Stateless Transport Tunneling (STT). This document describes VXLAN that is the most widely used overlay technology.

VXLAN is one of the Network Virtualization over Layer 3 (NVO3) technologies defined by the Internet Engineering Task Force (IETF) and is essentially a tunneling technology. VXLAN adds the VXLAN header to an original data frame, encapsulates the frame into a UDP packet, and forwards the UDP packet in traditional IP network transmission mode. After the UDP packet arrives at the end point, the end point removes the outer header and sends the original data frame to the target terminal. The end point of a VXLAN tunnel as shown in the Figure is called **VXLAN Tunnel Endpoint (VTEP)**, which encapsulates and decapsulates VXLAN packets. A VXLAN tunnel is defined by a pair of VTEPs. The source VTEP encapsulates packets and sends them to the destination VTEP through the VXLAN tunnel. The destination VTEP decapsulates the received packets.



VXLAN packet format



VXLAN Header: contains the 24-bit VNI field and 8-bit VXLAN flag bit. The other fields are reserved.

UDP Header: contains the destination port number fixed at 4789. The VXLAN header and the original Ethernet frame are used as UDP data.

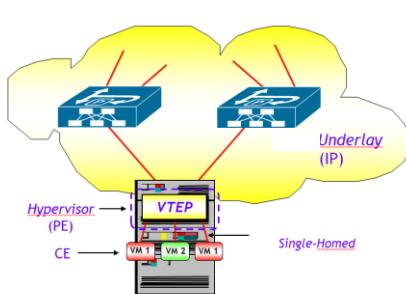
Outer IP Header: The source IP address is the IP address of the source VTEP and the destination IP address is the IP address of the destination VTEP.

Outer MAC Header: The source MAC address is the MAC address of the source VTEP, and the destination MAC address is the MAC address of the next-hop device on the route to the destination VTEP.

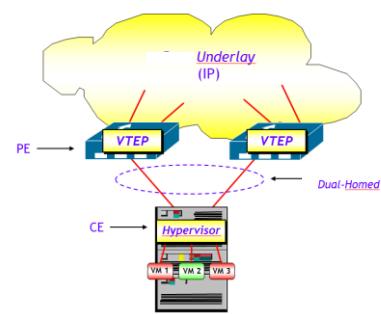


Where is VTEP implemented?

Any endpoint like a host, switch, or router that supports VxLAN can be referred to as a VTPE (VxLAN Tunnel Endpoint).



The VTEP function is implemented by software and located in a Hypervisor (e.g. KVM, Hyper-V, VMware NSX, etc.).



The VTEP function is hardware implemented and located in a TOR switch

Any endpoint like a host, switch, or router that supports VxLAN can be referred to as a VTPE (VxLAN Tunnel Endpoint).

As the name implies, the job of VTEPs is to create and terminate tunnels between each other. In other words, they encapsulate and decapsulate VxLAN traffic.

a. How does a VTEP work?

The VTPE is connected to the underlay network using a layer 3 IP address. VTPEs may have one or more VNIs associated with it.

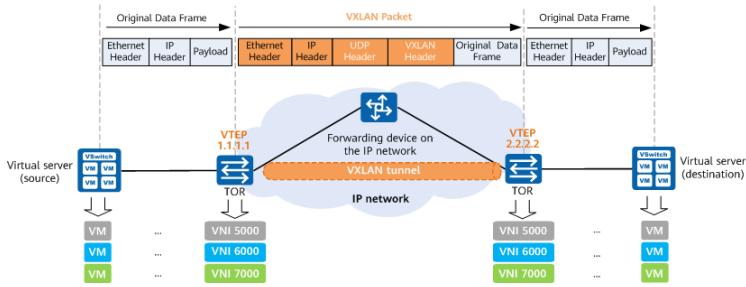
When a layer 2 frame with the same VNI arrives at the ingress VTEP, it encapsulates the frame with a VxLAN and UDP/IP headers.

Then sends it over using the underlay IP network transport towards the egress VTPE for decapsulation.

The egress VTPE removes the IP and UDP headers and delivers the original layer 2 frame.



VXLAN network model



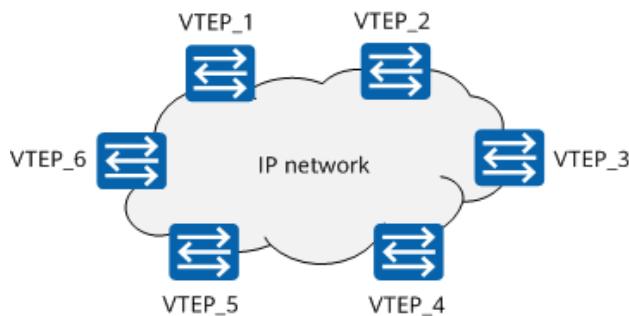
The VTEP is connected to the underlay network using a layer 3 IP address.
VTEPs may have one or more VNIs associated with it.

When a layer 2 frame with the same VNI arrives at the ingress VTEP, it encapsulates the frame with a VxLAN and UDP/IP headers.

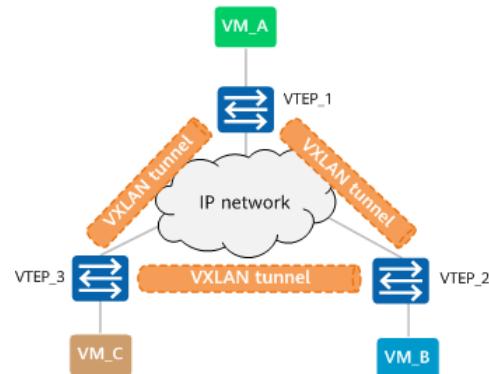
<https://support.huawei.com/enterprise/en/doc/EDOC1100086966>



VXLAN network model



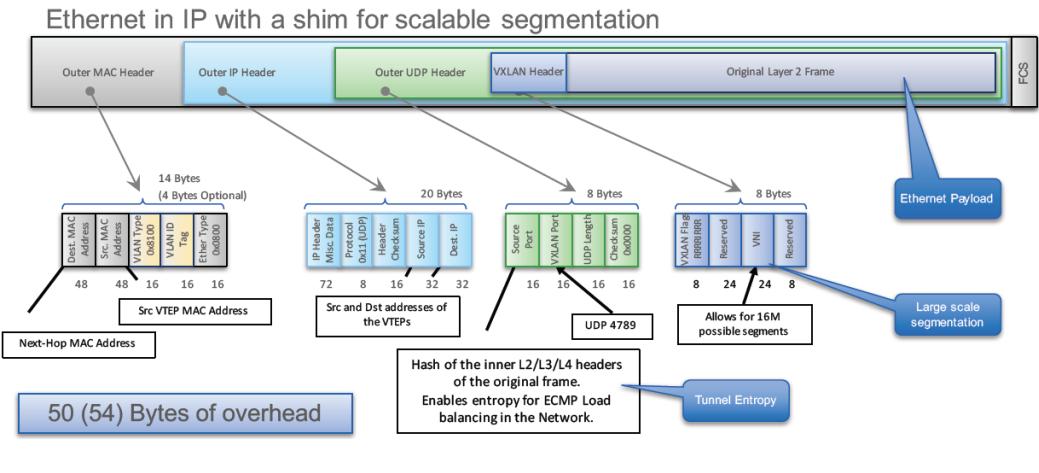
Assume that VMs each connected to VTEP_1, VTEP_2, and VTEP_3 respectively require large Layer 2 interconnection. Every two of VTEP_1, VTEP_2, and VTEP_3 then need to establish VXLAN tunnels between them,



The main problem to be addressed is L2-L3 address management and distribution over the VXLAN.



VXLAN packet format



RFC7348: IANA has assigned the value **4789** for the VXLAN UDP port, and this value **SHOULD** be used by default as the destination UDP port

17

RFC7348:

VXLAN Header: This is an 8-byte field that has:

- **Flags (8 bits):** where the I flag MUST be set to 1 for a valid VXLAN Network ID (VNI). The other 7 bits (designated "R") are reserved fields and MUST be set to zero on transmission and ignored on receipt.
- **VXLAN Segment ID/VXLAN Network Identifier (VNI):** this is a 24-bit value used to designate the individual VXLAN overlay network on which the communicating VMs are situated. VMs in different VXLAN overlay networks cannot communicate with each other. - Reserved fields (24 bits and 8 bits): MUST be set to zero on transmission and ignored on receipt.

- **Destination Port:** IANA has assigned the value 4789 for the VXLAN UDP port, and this value **SHOULD** be used by default as the destination UDP port. Some early implementations of VXLAN have used other values for the destination port. To enable interoperability with these implementations, the destination port **SHOULD** be configurable.

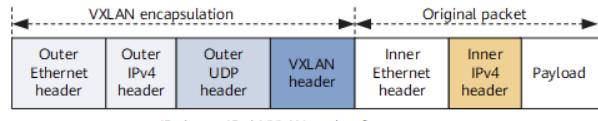
- **Source Port:** It is recommended that the UDP source port number be calculated using a hash of fields from the inner packet -- one example being a hash of the inner Ethernet frame's headers. This is to enable a level of entropy for the Equal-Cost Multipath (ECMP)/load- balancing of the VM-to-VM traffic across the VXLAN overlay. When calculating the UDP source port number in this manner, it is RECOMMENDED that the value be in the dynamic/private port range 49152-65535 [RFC6335]. Mahalingam, et al. Informational [Page 10] [RFC 7348](#) VXLAN August 2014

- **UDP Checksum:** It **SHOULD** be transmitted as zero. When a packet is received with a UDP checksum of zero, it **MUST** be accepted for decapsulation. Optionally, if the encapsulating end point includes a non-zero UDP checksum, it **MUST** be correctly calculated across the entire packet including the IP header, UDP header, VXLAN header, and encapsulated MAC frame. When a decapsulating end point receives a packet with a non-zero checksum, it **MAY** choose to verify the checksum value. If it chooses to perform such verification, and the verification fails, the packet **MUST** be dropped. If the decapsulating destination chooses not to perform the verification, or performs it successfully, the packet **MUST** be accepted for decapsulation.

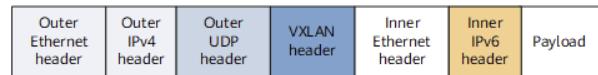


VXLAN packet format

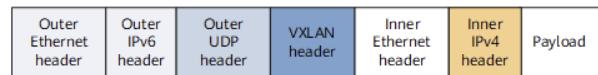
The infrastructure network on which VXLAN tunnels are established is called the underlay network, and the service network carried over VXLAN tunnels are called the overlay network. The following combinations of underlay and overlay networks exist in VXLAN scenarios.



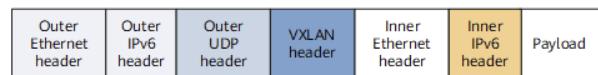
IPv4 over IPv4 VXLAN packet format



IPv6 over IPv4 VXLAN packet format



IPv4 over IPv6 VXLAN packet format



IPv6 over IPv6 VXLAN packet format



VXLAN Packet Forwarding Mechanism

Packets forwarded on a VXLAN network are classified into two types by forwarding mode:

- broadcast, unknown unicast, and multicast (**BUM**) packets
- known unicast packets.

Unknown-unicast traffic happens when a switch receives unicast traffic intended to be delivered to a destination that is not in its forwarding information base.



BUM packet forwarding

BUM packets can be forwarded in

- ingress replication
- multicast replication modes
- centralized replication

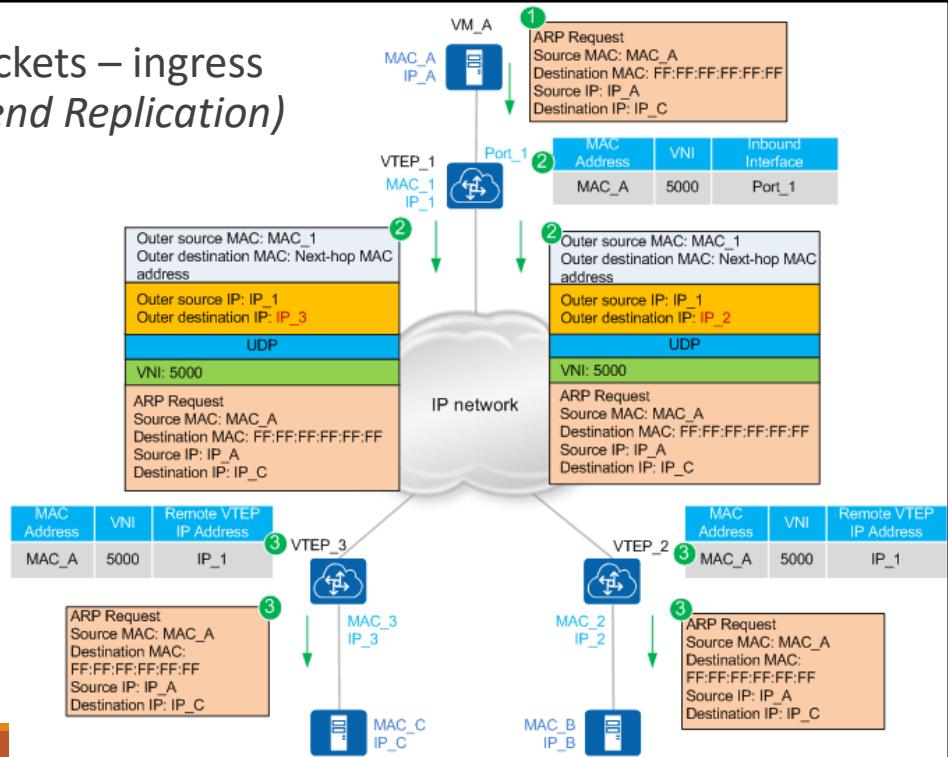
The most important problem in VXLAN implementation is the determination of MAC-to-VTEP tables

As with all switches, each leaf node creates a table. The table contains the MAC addresses of the devices connected to the leaf node's ports. What is different in VXLANs is the sharing of these tables. At an elementary level, the leaf nodes share their tables and associated VXLAN tunnel endpoint (VTEP) information with other leaf node VTEPs.

When a user device, for example, sends an egress packet to the leaf node to which it is connected, the leaf node checks its MAC-to-VTEP table.



Forwarding of BUM Packets – ingress replication (aka *Head-end Replication*)



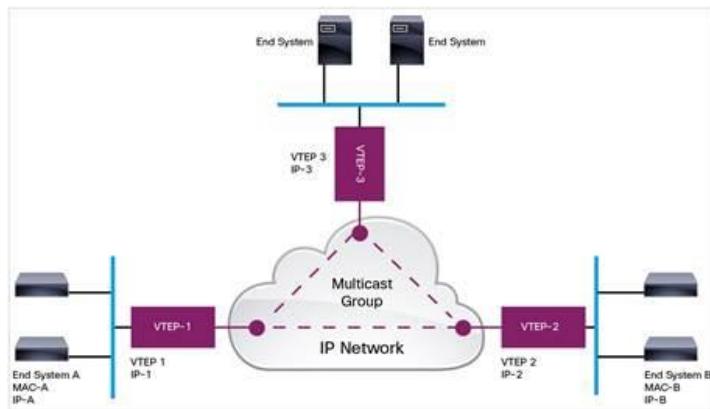
VM_A broadcasts an ARP Request packet, requesting the MAC address of VM_C. In the ARP Request packet, the source MAC address is MAC_A and the destination MAC address is all Fs.

After receiving the ARP Request packet, VTEP_1 determines the VNI and **ingress replication** list of the VXLAN tunnel. VTEP_1 replicates the ARP Request packet based on the ingress replication list, performs VXLAN encapsulation, and sends the encapsulated packet to each tunnel. In the encapsulated packet, the outer destination IP addresses are the IP addresses of the peer VTEPs (VTEP_2 and VTEP_3) respectively. The encapsulated packet is transmitted over the IP network based on the outer MAC address and IP address until it reaches the peer VTEPs.

After the packet reaches VTEP_2 and VTEP_3, VTEP_2 and VTEP_3 decapsulate it to obtain the original packet sent by VM_A. At the same time, VTEP_2 and VTEP_3 learn the mapping among MAC address of VM_A, VNI, and IP address of VTEP_1, and save the mapping in the local MAC address tables. Then, VTEP_2 and VTEP_3 send the original packet to the hosts in the corresponding Layer 2 domain. After receiving the ARP Request packet, VM_C sends an ARP Reply packet (while VM_B discards the ARP Request packet). Because VM_C has learned the MAC address of VM_A, the ARP Reply Packet is a known unicast packet.



Forwarding of BUM Packets – multicast replication





no

Forwarding of BUM Packets – multicast replication

In multicast replication mode, all VTEPs with the same VNI join the same multicast group.

- **Local MAC learning**
- **Remote MAC learning:** according to the specifications of RFC 7348 it is based on a multicast routing protocol (PIM-SM / SSM or more often PIM-BiDir) on the IP underlay network
 - problems both in terms of scalability and management of the underlay network.

- In multicast replication mode, all VTEPs with the same VNI join the same multicast group. A multicast routing protocol, such as PIM, is used to create a multicast forwarding entry for the multicast group. When the source VTEP receives a BUM packet, it adds a multicast destination IP address, such as 225.0.0.1, to the BUM packet before sending the packet to the remote VTEPs based on the created multicast forwarding entry, reducing flooded packets.

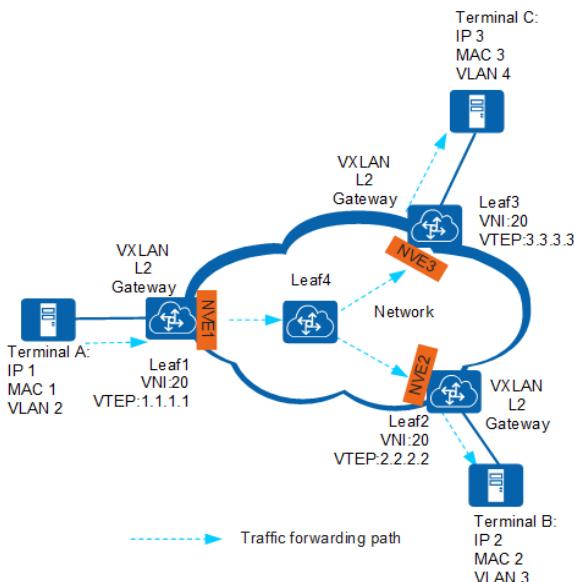
Protocol-Independent Multicast (PIM) Sparse Mode (SM)

Bidirectional (BiDir PIM) Protocol-Independent Multicast (PIM)

source-specific multicast (SSM) PIM Protocol-Independent Multicast (PIM)



Forwarding of BUM Packets – multicast replication



Layer 2 packet initiated by Terminal A		Leaf 1-encapsulated VxLAN packet Leaf1 -> Leaf4	Packet forwarded by Leaf 4 in multicast mode Leaf4 -> Leaf2/Leaf3	Leaf 2/Leaf 3-decapsulated VxLAN packet	
DMAC	All F	DMAC	Net MAC	DMAC	All F
SMAC	NVE1 MAC	SMAC	NVE1 MAC	SMAC	MAC1
SIP	1.1.1.1	SIP	1.1.1.1	VLAN Tag	3
DIP	225.0.0.1	DIP	225.0.0.1	DMAC	All F
UDP S_P	HASH	UDP S_P	HASH	SMAC	MAC1
UDP D_P	4789	UDP D_P	4789	VLAN Tag	4
VNI	20	VNI	20	DMAC	All F
DMAC	All F	DMAC	All F	SMAC	MAC1
SMAC	MAC1	SMAC	MAC1		

NVE (Network Virtual Interface): Logical interface where the encapsulation and de-encapsulation occur. The point where the VTEP function is implemented.

A bridge domain is a set of logical ports that share the same flooding or broadcast characteristics. Like a virtual LAN (VLAN), a bridge domain spans one or more ports of multiple devices.

In simple terms, a bridge domain is something that makes it possible to define a broadcast domain that is contained within a bridging device. It is a substitute for 802.1D bridge groups as well as 802.1Q VLAN bridging. The purpose of a bridge domain is to specify the broadcast domain number.

After receiving a packet from Terminal A, Leaf 1 determines the Layer 2 BD (Bridge Domain) of the packet **based on the access interface and VLAN information**.

Leaf 1's VTEP obtains the multicast replication address for the VNI based on the Layer 2 BD and performs VxLAN encapsulation. The encapsulated VxLAN packet is displayed as a multicast packet. The VTEP forwards it to Leaf 4 based on the matching multicast forwarding entry.

After receiving the multicast packet, Leaf 4 directly forwards it to Leaf 2 and Leaf 3 based on the matching multicast forwarding entry. NOTE: Leaf 4 acts as a non-gateway node and directly forwards multicast packets. Leaf 4 can be configured as a gateway node. In this case, Leaf 4 needs to forward multicast packets and decapsulate VxLAN packets. In this way, Leaf 4 is called a Bud node.

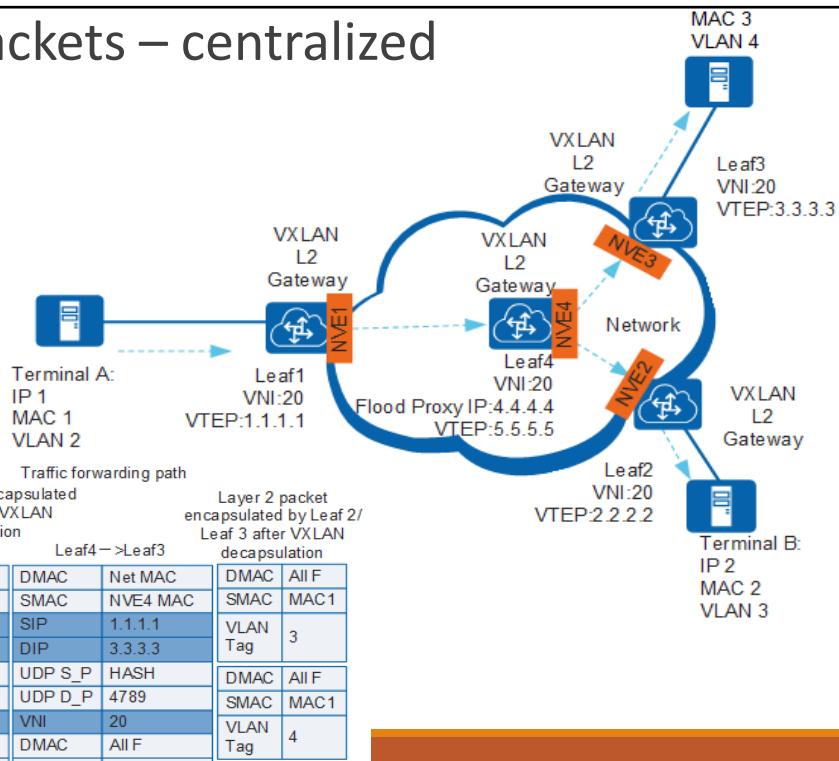
After the VTEP on Leaf 2/Leaf 3 receives the packet, it finds that the packet is a VxLAN packet after searching for the outbound interface (NVE interface) in a matching multicast forwarding entry. It checks the UDP destination port number, source and destination IP addresses, and VNI of the packet to determine the packet validity. After confirming that the packet is valid, the VTEP obtains the BD based on the VNI and decapsulates the VxLAN packet to obtain the inner Layer 2 packet.

Leaf 2/Leaf 3 checks the destination MAC address of the inner Layer 2 packet and finds it a BUM MAC address. Therefore, Leaf 2/Leaf 3 broadcasts the packet onto the network connected to the terminals (not the VxLAN tunnel side) in the Layer 2 BD. Specifically, Leaf 2/Leaf 3 finds the outbound interfaces and encapsulation information not related to the VxLAN tunnel, adds VLAN tags to the packet, and forwards the packet to Terminal B/Terminal C.

Note that it is not necessary to have a multicast tree for VxLAN, an aspect that could lead to serious scalability problems, being the VxLAN of the order of hundreds of thousands (in theory, as mentioned above, up to more than 16 million). Multicast trees shared by multiple VxLANs can be used safely, however the segregation of traffic between VxLANs is not affected since this depends exclusively on the value of the VNI. The downside to using shared multicast trees is that BUM traffic can also go to VTEPs that do not have hosts of a particular VNI, with consequent bandwidth waste.



Forwarding of BUM Packets – centralized replication



The centralized replication mode can prevent multicast problems.

In centralized replication mode, the centralized replication function is configured on the ingress VTEP and the flood proxy IP address is configured on the centralized replicator. When a BUM packet enters a VXLAN tunnel, the ingress VTEP only needs to send one copy of the packet to the centralized replicator, reducing flooded traffic on the network. The centralized replicator is also called flood gateway. The centralized replicator decapsulates and encapsulates the BUM packet and sends it to each egress VTEP. When the BUM packet leaves the VXLAN tunnel, the egress VTEPs decapsulate the BUM packet. The figure shows the forwarding process of a BUM packet in centralized replication mode.

After Leaf 1 receives a packet from Terminal A, Leaf 1 determines the Layer 2 BD of the packet based on the access interface and VLAN information.

Leaf 1's VTEP obtains the centralized replication tunnel for the VNI based on the Layer 2 BD and performs VXLAN encapsulation. Leaf 1 then forwards the VXLAN packet through the outbound interface.

After Leaf 4 used as the centralized replicator receives the VXLAN packet, it checks the UDP destination port number, source and destination IP addresses, and VNI of the packet to determine the packet validity. After confirming that the packet is valid, the VTEP obtains the BD based on the VNI, decapsulates the VXLAN packet to obtain the inner Layer 2 packet, and then performs VXLAN encapsulation based on the matching ingress replication list. After VXLAN encapsulation, the outer source IP address is the VTEP address of Leaf 1. Therefore, MAC address learning among the VTEPs is not affected.

After the VTEP on Leaf 2/Leaf 3 receives the VXLAN packet, it checks the UDP destination port number, source and destination IP addresses, and VNI of the packet to determine the packet validity. After confirming that the packet is valid, the VTEP obtains the BD based on the VNI and decapsulates the VXLAN packet to obtain the inner Layer 2 packet.

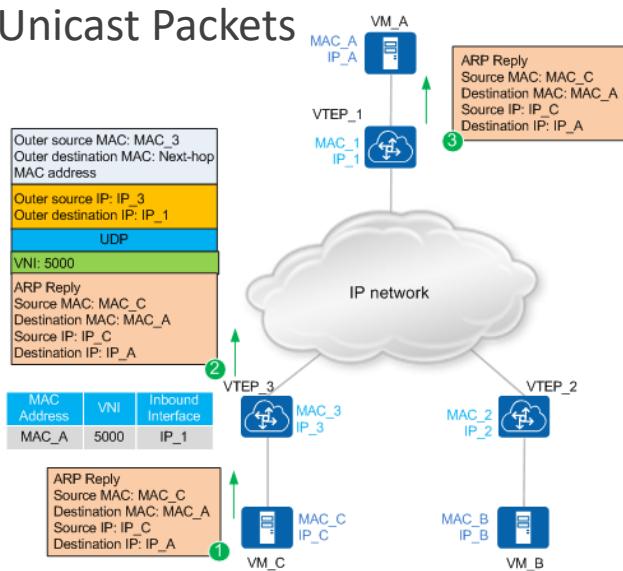
Leaf 2/Leaf 3 checks the destination MAC address of the inner Layer 2 packet and finds it a BUM MAC address.

Therefore, Leaf 2/Leaf 3 broadcasts the packet onto the network connected to the terminals (not the VXLAN tunnel side) in the Layer 2 BD. Specifically, Leaf 2/Leaf 3 finds the outbound interfaces and encapsulation information not related to the VXLAN tunnel, adds VLAN tags to the packet, and forwards the packet to Terminal B/Terminal C.



Forwarding of Known Unicast Packets

VM_C sends an ARP Reply packet to VM_A with the source MAC address being MAC_C and the destination MAC address being MAC_A.



VM_C sends an ARP Reply packet to VM_A with the source MAC address being MAC_C and the destination MAC address being MAC_A.

After receiving the ARP Reply packet sent by VM_C, VTEP_3 determines the VNI and performs VXLAN encapsulation on the packet. Since VTEP_3 has learned the MAC address of VM_C, the outer destination IP address in the encapsulated packet is the IP address of the peer VTEP (VTEP_1). The encapsulated packet is transmitted over the IP network based on the outer MAC address and IP address until it reaches VTEP_1.

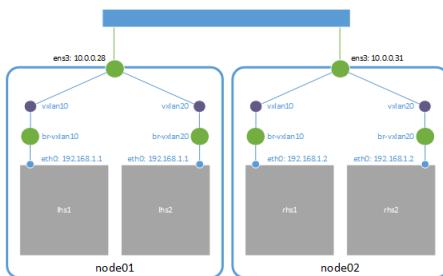
After the packet reaches VTEP_1, VTEP_1 decapsulates it to obtain the original packet sent by VM_C. Then, VTEP_1 sends the decapsulated packet to VM_A.



nah

Enjoy with Linux...

In each of the nodes



```
ip link add vxlan10 type vxlan id 10 group 239.1.1.1 dstport 0 dev ens3  
ip link add br-vxlan10 type bridge  
ip link set vxlan10 master br-vxlan10  
ip link set vxlan10 up  
ip link set br-vxlan10 up
```

```
ip link add vxlan20 type vxlan id 20 group 239.1.1.1 dstport 0 dev ens3  
ip link add br-vxlan20 type bridge  
ip link set vxlan20 master br-vxlan20  
ip link set vxlan20 up  
ip link set br-vxlan20 up
```

<https://ilearnedhowto.wordpress.com/2017/02/16/how-to-create-overlay-networks-using-linux-bridges-and-vxlans/>

First we create a vxlan port with VNI 10 that will use the device ens3 to multicast the UDP traffic using group 239.1.1.1 (using dstport 0 makes use of the default port). Then we will create a bridge named br-vxlan10 to which we will bridge the previously created vxlan port.
Finally we will set both ports up



Flood and Learn (AKA Bridging)

- Flood and Learn was the original learning method for VxLAN (all VTEPs with this VNI). It is also known as bridging, as it's used to create virtual bridges (VNIs) between hosts.
- The other reason this is called bridging is that this is a layer-2 only solution. There is no built-in way to route between VNI's. If you need this, you must connect an external router, and let traffic pass through it.
- The flooding nature of this method limits its scalability. Also, as there is no control plane learning of VTEPs, its possible for a rogue VTEP to be added to the network. It could intercept and inject traffic.
- BUM traffic must be handled by multicast. There is no option for Head End replication in this case.
- While this method is still worth understanding, it is recommended to use control plane learning in production.



EVPN in VXLAN

- On VXLAN networks, VXLAN tunnels can be manually created by configuring VXLAN network identifiers (VNIs) and peer lists of VNIs. The configuration is difficult for users and requires heavy manual workload.
- VXLAN does not provide a control plane**, and VTEP discovery and MAC addresses learning are implemented by traffic flooding on the data plane, resulting in high traffic volumes on DC networks.
- To address this problem, BGP operates in the control plane. A normal BGP deployment will share IP reachability information (routes). When integrated with VxLAN, it can also share MAC and VTEP reachability information.
- It is referred to as **Ethernet virtual private network – EVPN**.
- EVPN allows devices acting as VTEPs to exchange reachability information about endpoints as Layer 2 MAC addresses and Layer 3 IP addresses. As all the addresses are learned proactively, there's no need for flooding.
- All switches in the VxLAN topology need to run BGP EVPN. They don't all need to be running VTEPs. An example is the spine switches in the spine/leaf topology.

VXLAN does not provide the control plane, and VTEP discovery and host information (IP and MAC addresses, VNIs, and gateway VTEP IP address) learning are implemented by traffic flooding on the data plane, resulting in high traffic volumes on VXLAN networks. To address this problem, VXLAN uses EVPN as the control plane. EVPN allows VTEPs to exchange BGP EVPN routes to implement automatic VTEP discovery and host information advertisement, preventing unnecessary traffic flooding.



meh

EVPN in VXLAN

- Traditional BGP-4 peers use Update messages to exchange routing information. An Update message can advertise reachable routes with the same path attribute. These routes are carried in the **Network Layer Reachability Information (NLRI)** field.
- BGP-4 can manage only IPv4 unicast routing information, so **MP-BGP was developed to support multiple network layer protocols**, such as IPv6 and multicast. MP-BGP extends NLRI based on BGP-4. After extension, the description of the address family is added to NLRI to differentiate network layer protocols, such as the IPv6 unicast address family and VPN instance address family.
- Similarly, EVPN uses the MP-BGP mechanism and defines a **new sub-address family, EVPN address family**, in the L2VPN address family. In the EVPN address family, a new type of NLRI is added, that is, **EVPN NLRI**. EVPN NLRI defines several types of BGP EVPN routes, which can carry information such as the host IP address, MAC address, VNI, and VRF.
- After a VTEP learns the IP address and MAC address of a connected host, the VTEP can send the information to other VTEPs through MP-BGP routes. This way, learning of host IP address and MAC address information can be implemented on the control plane, suppressing traffic flooding on the data plane.

The Network Layer Reachability Information (NLRI) is exchanged between BGP routers using UPDATE messages. An NLRI is composed of a LENGTH and a PREFIX. The length is a network mask in CIDR notation (eg. /25) specifying the number of network bits, and the prefix is the Network address for that subnet

The NLRI is unique to BGP version 4 and allows BGP to carry supernetting information, as well as perform aggregation.

The NLRI would look something like one of these:

/25, 204.149.16.128
/23, 206.134.32
/8, 10



EVPN in VXLAN

Using EVPN as the control plane of VXLAN has the following advantages:

- VTEPs can be automatically discovered and VXLAN tunnels can be automatically established, simplifying network deployment and expansion.
- EVPN can advertise both Layer 2 MAC address information and Layer 3 routing information.
- Flooding traffic is reduced on the network. When a host comes online, it announces its MAC address. This can also happen at other times with Gratuitous ARP messages. The local switch will add the MAC into the local BGP database. This is then sent to its peers as a BGP update.
- Five Types of EVPN Routes (RFC 7432):

Type 1 Ethernet auto-discovery (A-D) route

Type 2 MAC/IP advertisement route

Type 3 Inclusive multicast Ethernet tag route

Type 4 Ethernet segment route

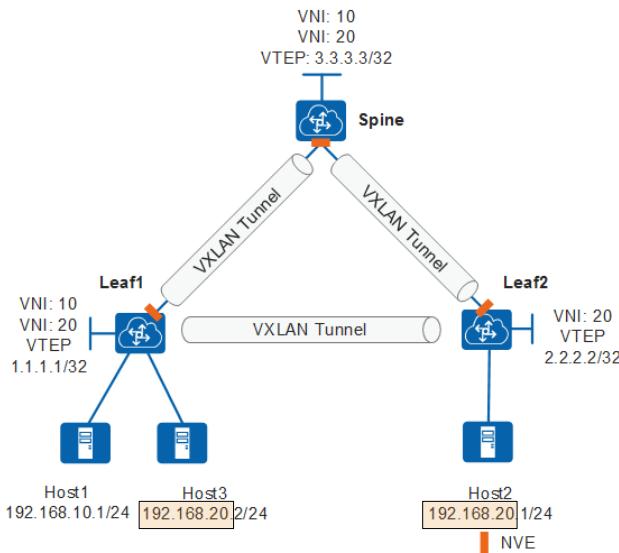
Type 5 IP prefix route

<https://support.huawei.com/enterprise/en/doc/EDOC1100168670#:~:text=In%20the%20EVPN%20address%20family,address%2C%20VNI%2C%20and%20VRF>.



ripetitivo, recap breve

VXLAN Tunnel Establishment



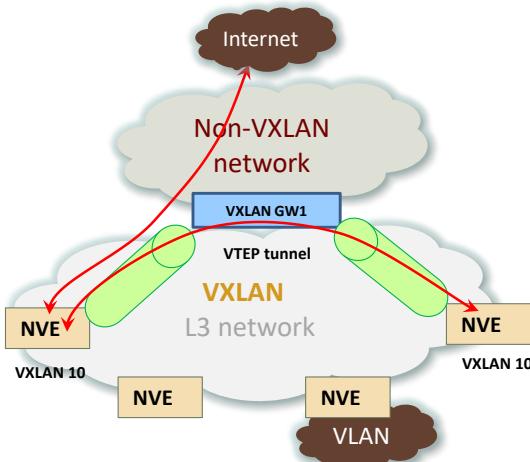
When BGP EVPN is used to dynamically establish a VXLAN tunnel, the local and remote VTEPs first establish a BGP EVPN peer relationship and then exchange BGP EVPN routes to transmit VNIs and VTEPs' IP addresses

A VXLAN tunnel is identified by a pair of VTEP IP addresses. During VXLAN tunnel establishment, the local and remote VTEPs attempt to obtain the IP addresses of each other. A VXLAN tunnel can be established if the IP addresses obtained are reachable at Layer 3. When BGP EVPN is used to dynamically establish a VXLAN tunnel, the local and remote VTEPs first establish a BGP EVPN peer relationship and then exchange BGP EVPN routes to transmit VNIs and VTEPs' IP addresses.

On the network shown in the figure, Leaf 1 connects to Host 1 and Host 3; Leaf 2 connects to Host 2; Spine functions as a Layer 3 gateway. To allow Host 3 and Host 2 to communicate, establish a VXLAN tunnel between Leaf 1 and Leaf 2. To allow Host 1 and Host 2 to communicate, establish a VXLAN tunnel between Leaf 1 and Spine and between Spine and Leaf 2. Although Host 1 and Host 3 both connect to Leaf 1, they belong to different subnets and must communicate through the Layer 3 gateway (Spine). Therefore, a VXLAN tunnel is also required between Leaf 1 and Spine.



VXLAN Gateways



To implement Layer 3 interworking, a Layer 3 gateway must be deployed on a VXLAN.

Similar to a VXLAN NVE, a Layer 3 VXLAN gateway provides mappings between the VXLAN packet header and IP packet header.

Different VLANs need to communicate with each other through Layer 3 gateways.

Similarly, Layer 3 gateways are also required for communication between VXLANs with different VNIs.

In the typical spine-leaf VXLAN networking, Layer 3 VXLAN gateways can be classified into centralized gateways and distributed gateways based on their deployment locations.

The point where the VTEP function is implemented is referred to as *NVE, Network Virtualization Endpoint*. We can consider it in the VTEP interface.

An NVE is a functional module at the server virtualization layer, enabling VMs to use virtualization software to establish VTEP tunnels.

An NVE can also be a VXLAN-capable access switch that provides the VXLAN gateway service to multiple tenants in a centralized manner.

A VXLAN gateway can implement communication between tenants on different VXLANs, as well as between VXLAN users and non-VXLAN users. This function is similar to that of a VLANIF interface.



VXLAN Gateways

- Both the Layer 2 VXLAN gateway and Layer 3 VXLAN gateway are used to implement connection between a VXLAN networks. **Integrated Routing and Bridging (IRB)** is supported in VTEPs when BGP is used. This means that each switch with a VTEP can also behave as a router.
- VXLAN gateways can be deployed in centralized or distributed mode.
- Centralized VXLAN Gateway Mode**
 - In this mode, Layer 3 gateways are configured on one device. Traffic across network segments is forwarded through Layer 3 gateways to implement centralized traffic management
- Distributed VXLAN Gateway Mode**
 - In the distributed VXLAN gateway networking, each leaf node functions as a VTEP and as a Layer 3 VXLAN gateway.



VNI EVPN Types

Two types of VNI's are used which is one for L2 operations and one for L3 operations. They are referred to as layer-2 VNI's (L2VNI) or layer-3 VNI's (L3VNI). BGP distributes both of these to all peers.

- **L2VNI** is the bridge domain. This is for bridging hosts on the same layer-2 segment. Essentially, it is the VxLAN equivalent of a VLAN. It is recommended to keep this one-to-one relation between L2VNI and VLAN's
- **L3VNI** can be used to route between L2VNI's, so this will have an IP associated which is used for routing purposes. The ingress or egress VTEP can perform routing. This is called **Symmetric IRB**. Another form of routing called **Asymmetric IRB**, uses the ingress VTEP for routing and bridging, while the egress VTEP can only do bridging.

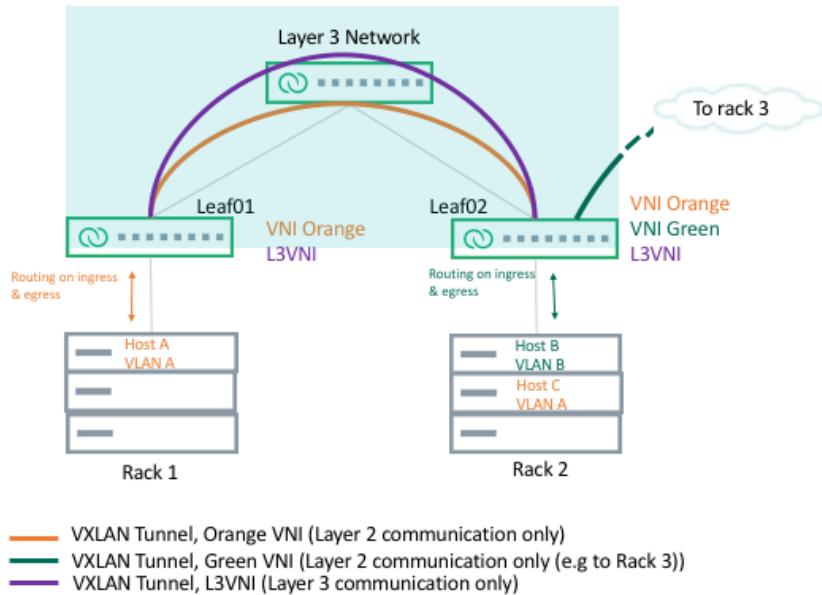
MAC-VRF: A Virtual Routing and Forwarding table for storing Media Access Control

(MAC) addresses on a VTEP for a specific tenant.

The L2VNI is in charge of forwarding traffic within the same VNI ("VLAN") between two switches. This is part of the overlay, and from a user's perspective the network is behaving as one big switch.



VNI Types - symmetric EVPN IRB



<https://cumulusnetworks.com/blog/asymmetric-vs-symmetric-model/>

The symmetric model routes and bridges on both the ingress and the egress leafs. This results in bi-directional traffic being able to travel on the same VNI, hence the symmetric name.

However, a new specialty transit VNI is used for all routed VXLAN traffic, called the L3VNI. All traffic that needs to be routed will be routed onto the L3VNI, tunneled across the layer 3 Infrastructure, routed off the L3VNI to the appropriate VLAN and ultimately bridged to the destination.

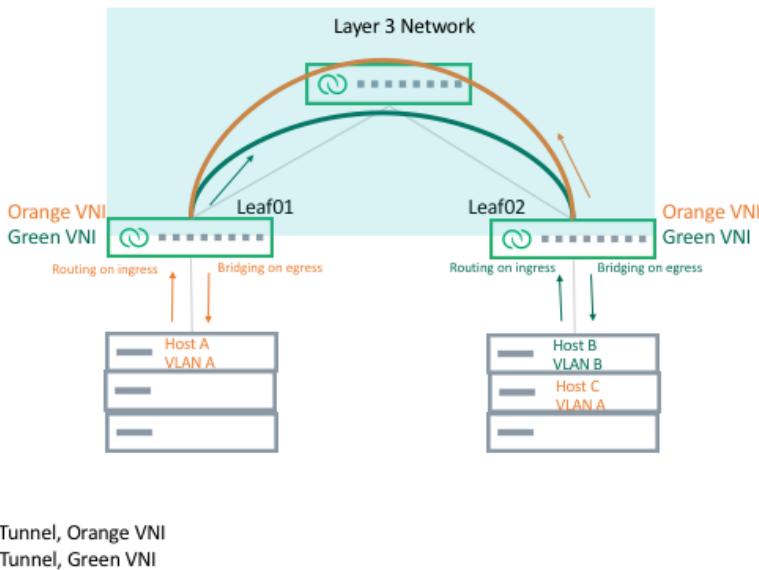
Now consider the scenario with a symmetric model, as shown above. Let's say Host A on VLAN A needs to communicate with Host B on VLAN B. Since the destination is a different subnet from Host A, Host A sends the frame to its default gateway, which is Leaf01.

Leaf01 recognizes that the destination MAC address is itself and will use the routing table to route the packet to the L3VNI and nexthop Leaf02. The VXLAN-encapsulated packet will have the egress leaf's MAC as the destination MAC address and this L3VNI as the VNI. Leaf02 performs VXLAN decapsulation and recognizes that the destination MAC address is itself and routes the packet on to the destination VLAN, to reach the destination host. The return traffic will be routed similarly over the same L3VNI.

With symmetric model, the leaf switches only need to host the VLANs and the corresponding VNIs that are located on its rack, as well as the L3VNI and its associated VLAN, since the ingress leaf switch doesn't need to know the destination VNI. The ability to host only the local VNIs (plus one extra) helps with scale. However, **the configuration is more complex as an extra VXLAN tunnel and VLAN in your network are required**. The data plane traffic is also more complex as an **extra routing hop occurs and could cause extra latency**. Multitenancy requires one L3VNI per VRF, and all switches participating in that VRF must be configured with the same L3VNI. The L3VNI is used by the egress leaf to identify the VRF in which to route the packet.



VNI Types - asymmetric EVPN IRB



When a switch performs the VTEP functions, it is referred to as a VxLAN Gateway.

The switches can perform VxLAN encapsulation/decapsulation and can also translate the VLAN ID to VNI.

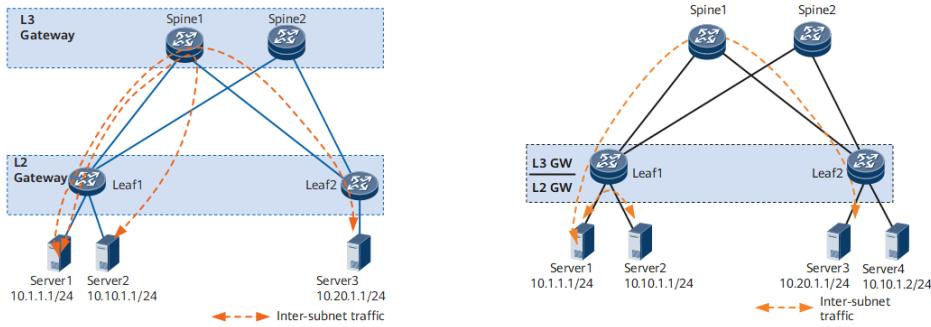
The VxLAN gateway creates the tunnel to the destination VTEP (either host or gateway), so the hosts and IP infrastructure are not aware of the existence of VxLAN.

<https://developer.nvidia.com/blog/using-vxlan-routing-with-evpn-through-asymmetric-or-symmetric-models/>

The asymmetric model allows **routing and bridging on the VXLAN tunnel ingress**, but **only bridging on the egress**. This results in bi-directional VXLAN traffic traveling on different VNIs in each direction (always the destination VNI) across the routed. Host A wants to communicate with Host B, which is located on a different VLAN and a different rack, thus reachable via a different VNI. Since Host B is on a different subnet from Host A, Host A sends the frame to its default gateway, which is Leaf01 (this is generally an Anycast Gateway, but we can cover that in a later post). Leaf01 recognizes that the destination MAC address is itself, looks up the routing table and routes the packet to the Green VNI while still on Leaf01. Leaf01 then tunnels the frame in the Green VNI to Leaf02. Leaf02 removes the VXLAN header from the frame, and bridges the frame to Host B. Likewise, the return traffic would behave similarly. Host B sends a frame to Leaf02. Leaf02 recognizes its own destination MAC address and routes the packet to the Orange VNI on Leaf02. The packet is tunneled within the Orange VNI to Leaf01. Leaf01 removes the VXLAN header from the frame and bridges it to Host A. With the asymmetric model, **all the required source and destination VNIs (e.g. orange and green) must be present on each leaf, even if that leaf doesn't have a host in that VLAN in its rack**. This may increase the number of IP/MAC addresses the leaf must hold, which results in somewhat limited scale. However, in many instances, all VNIs in the network are configured on all leaves anyway to allow VM mobility and to simplify configuration of the network as a whole, in which case asymmetric model is desirable.



VXLAN Gateways

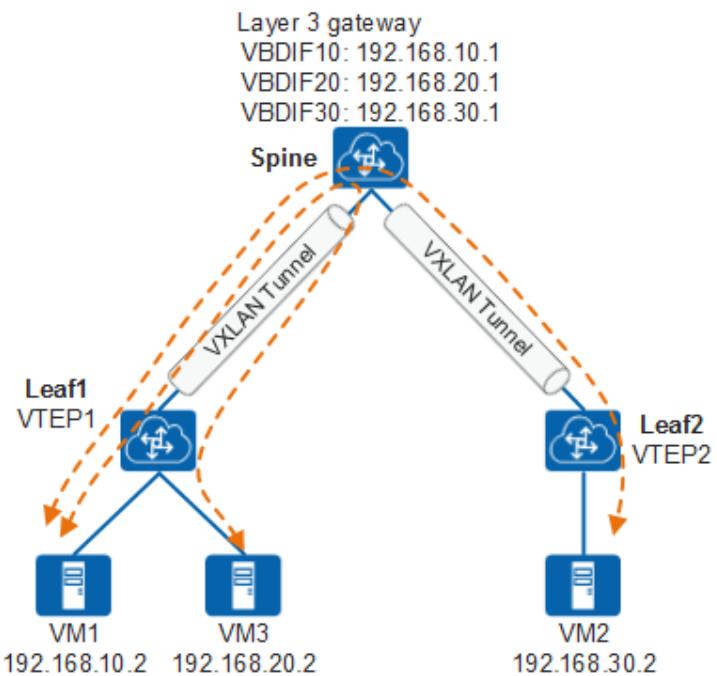




Centralized Gateway

Centralized VXLAN gateway deployment has its advantages and disadvantages.

- Advantage: Inter-segment traffic can be centrally managed, and gateway deployment and management is easy.
- Disadvantages:
 - Forwarding paths are not optimal. Inter-segment Layer 3 traffic of data centers connected to the same Layer 2 gateway must be transmitted to the centralized Layer 3 gateway for forwarding.
 - The ARP entry specification is a bottleneck. ARP entries must be generated for tenants on the Layer 3 gateway. However, only a limited number of ARP entries are allowed by the Layer 3 gateway, impeding data center network expansion.



VBDIF interfaces are Layer 3 logical interfaces. A VBDIF interface is a virtual interface based on a bridge domain and supporting Layer 3 features. VBDIF interfaces implement communication between BDs, between BD and non-BD networks, and between BD and Layer 3 networks. After creating VBDIF interfaces, you can configure Layer 3 features on these interfaces.

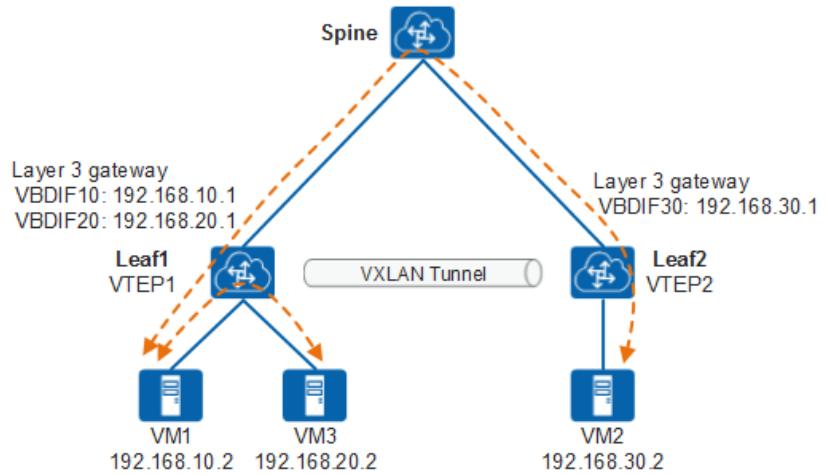
In the centralized gateway networking, Layer 3 gateways are centrally deployed on one spine node, as shown in the following figure. Inter-subnet traffic is forwarded through the Layer 3 gateways.

In the centralized gateway networking, inter-subnet traffic can be centrally managed. Gateway deployment and management are simple, but inter-subnet traffic of VMs on the same leaf node needs to be forwarded by the spine node. Therefore, the traffic forwarding path is not optimal. In addition, all entries of terminals whose traffic is forwarded at Layer 3 need to be generated on the spine node. However, the spine node supports only a limited number of entries. When the number of tenants increases, it may become a network bottleneck.



Distributed Gateway

In the distributed gateway scenario, a control plane is required to transmit host routes between Layer 3 gateways to ensure communication between hosts. To meet this requirement, Ethernet VPN (EVPN) is introduced as the VXLAN control plane.



Distributed VXLAN gateways use the spine-leaf network. In this networking, leaf nodes, which can function as Layer 3 VXLAN gateways, are used as VTEPs to establish VXLAN tunnels. Spine nodes are unaware of the VXLAN tunnels and only forward VXLAN packets between different leaf nodes.

Layer 3 VXLAN gateways are deployed on leaf nodes to enable inter-subnet communication of VMs on the same leaf node. In this way, traffic is directly forwarded by the leaf nodes without passing through the spine node. This conserves bandwidth resources. Unlike centralized Layer 3 gateways that need to learn ARP entries of all hosts, a leaf node in the distributed VXLAN gateway scenario only needs to learn the ARP entries of hosts connected to itself. This eliminates the bottleneck caused by limited ARP entry specifications in the centralized Layer 3 gateway scenario and improves network expansion capabilities.

In the distributed gateway scenario, a control plane is required to transmit host routes between Layer 3 gateways to ensure communication between hosts. To meet this requirement, Ethernet VPN (EVPN) is introduced as the VXLAN control plane. By referring to the BGP/MPLS IP VPN mechanism, EVPN defines several types of BGP EVPN routes by extending BGP. It advertises routes on the network to implement automatic VTEP discovery and host address learning.

Sources

<https://reissromoli.com/it/formazione/catalogo/38-reissblog/vxlan-pc.html>

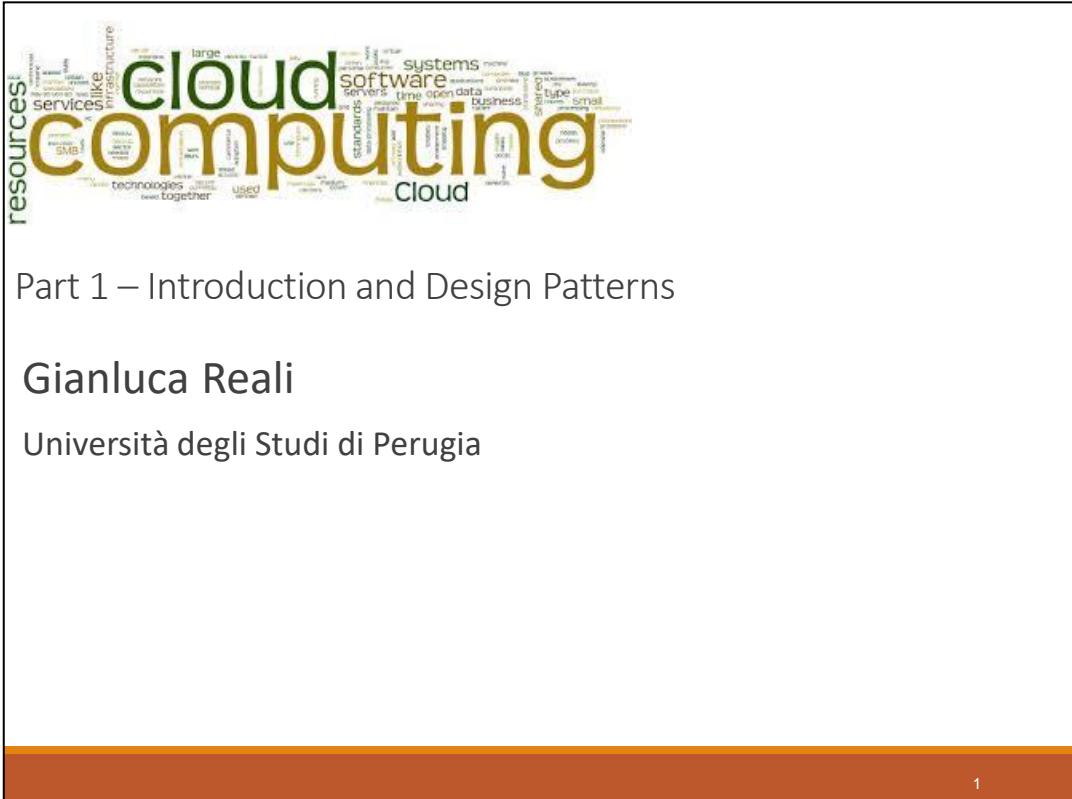
<https://support.huawei.com/enterprise/en/doc/EDOC1100086966>

<https://support.huawei.com/enterprise/en/doc/EDOC1100092936/53f4c6d2/evpn-vxlan-fundamentals>

<https://support.huawei.com/enterprise/en/doc/EDOC1000180270/4d97d2db/example-for-configuring-vxlan-using-bgp-evpn-to-enable-communication-among-users-on-the-same-network-segment>

<https://support.huawei.com/enterprise/en/doc/EDOC1100004365/3e18cbdb/centralized-vxlan-gateway-deployment-using-bgp-evpn>

<https://networkdirection.net/articles/routingandswitching/vxlanoverview/vxlanaddresslearning/>



What do we mean by “cloud?”



Made up of massive datacenters of concrete and steel

Filled with thousands of rows of server racks housing customer data



Slide objectives

When people talk about “the cloud” they are in fact referring to data stored in massive datacenters around the world, which are each filled with many thousands of servers.

Main messaging

The cloud is not an abstract concept, it is a physical space with physical boundaries, requirements and limitations

These physical resources dedicated to the Azure cloud are commonly referred to as the “Azure Fabric” on which all the Azure cloud services run

Characteristics of cloud computing

- Network access to cloud services
- Pay only what you need from a measured service
- Multi-tenancy – many customers in same space
- On-demand self-service to scalable resources
- High bandwidth links to and between datacenters

Why the Cloud? Why not the Cloud?

If you use a (**public**) cloud:

- If you don't have a huge quantity of users, data, or processing, then your **costs** are probably lower:
 - you won't have to pay for as many system administrators
 - you won't have to buy big servers
 - you don't have to manage big networks
- You can have **scalability**, that is, if you have a sudden increase in users then the cloud infrastructure can handle it (of course, at a price).
 - to be able to do this yourself you would have to pay for and maintain extra infrastructure in terms of servers, networks, and system administrators
- Your data may be backed up at a different location, such that it can **survive** a large natural disaster. Or possibly even a war.

Why the Cloud? Why not the Cloud? (cont'd)

There are some security advantages:

- The cloud provider has a big incentive to keep all its **software up to date** with security patches
 - whereas you would have to have a good staff of system administrators, as well as regular update procedures, to make sure this is done in your own company
- The cloud most likely provides some sort of **physical security** for its servers
 - if you do the same in your company, this requires locked rooms with access control, and additional procedures that must be followed. For example, who gets a key? If someone is fired, do you change all the locks? Etc.
- The cloud provider has a big incentive to hire **good system administrators** who have been background-checked

Why the Cloud? Why not the Cloud? (cont'd)

What are some **negatives** for having your company's software on a cloud?

- Your particular application may not benefit from added scalability, particularly if it is a traditional **monolithic application** architecture for which a stable demand is expected
- It may be **difficult to organize** or re-write your application such that it would run well when located on a cloud.
 - There is some literature that says that an application must have a Service Oriented Architecture in order to be able to be hosted on a cloud
- Even if it is possible to move your application to the cloud such that its performance is good, there will probably be some cost involved in making this happen. Perhaps significant cost.

Why the Cloud? Why not the Cloud? (cont'd)

There may be security disadvantages associated with being on a cloud:

- The cloud provider has a big incentive to have good security, however, if you onboard your software to a cloud, **you give up control** of its security

There can be **data policy or privacy disadvantages** associated with being on a cloud:

- your data would be managed by the cloud provider, not by you.

7

there are cases where by law or by privacy agreement, an individual wishes to delete his/her private information.

How can you ensure the cloud provider actually does this?

What is your cloud provider's policy on notifying you if there has been a data breach? How long will this take?—see Winkler (2012)

What is your responsibility for notifying the cloud provider if your data has been breached? Do you have any liability?

Why the Cloud? Why not the Cloud? (cont'd)

- in large cloud companies with data centers located in other countries (maybe not that friendly to your own country), then **physical location of the data** can be a big issue
- how do you know the cloud provider is not mining your private data for its own purposes and perhaps selling this information to a third party?
 - What is your company's **liability** if the cloud provider does this wrong?

General Features

This cloud model includes some essential **Characteristics**, **Service Models**, and **Deployment models**.

Characteristics

- On-demand, self-service
- Broad network access
- Resource pooling
 - Location independence
- Rapid elasticity
- Measured service

Service Models

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)
- Function as a Service (FaaS) - Serverless Computing
- ... a lot of further proposals

Deployment Models

- *Private Cloud*
- *Public Cloud*
- *Community Cloud*
- *Hybrid Cloud*

9

At one end of the spectrum is infrastructure as a service (IaaS). With IaaS, you run the provisioning of the required virtual machines, as well as associated network and storage components. Next, you deploy the desired software and applications to virtual machines. This model is most similar to a traditional on-premises environment, except that the infrastructure is managed from the cloud providers. You can still manage individual virtual machines. The Platform-as-a-Service (PaaS) model offers a hosting environment managed, where you can deploy your application without needing to manage VMs or resources net. For example, instead of creating individual virtual machines, you specify a number of instances and the service will provision, configure and manage the necessary resources. The Functions-as-a-Service (FaaS) model goes even further than that eliminating the need to manage your hosting environment. Instead create compute instances and deploy code to your instances, you simply deploy the code and the service will run it automatically. It is not need to manage computing resources. These services use serverless architecture and augment or automatically reduce performance to the level needed to handle the traffic. IaaS offers the ultimate in control, flexibility, and portability. FaaS offers simplicity, elastic scalability and potentially cost savings, as you only pay based on code execution times. PaaS is a intermediate model. In general, the more flexibility a service offers, the more accountable you are at first resource management and configuration person. FaaS services automatically handle almost all of them aspects of running an application, while IaaS solutions involve provisioning, the autonomous configuration and management of created virtual machines and network components.

Software as a Service (SaaS)

Cloud-native applications are those software developed as software-as-a-service (SaaS), or that application distribution model where a software manufacturer develops and makes available, therefore also managing the service, a Web App that can be granted free of charge or through a service subscription.

The **capability provided to the consumer is to use the provider's applications** running on a cloud infrastructure and accessible from various client devices through a thin client interface such as a Web browser (e.g., web-based email). The consumer does not manage or control the underlying cloud infrastructure, network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

Platform as a Service (PaaS)

The **capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created applications using programming languages and tools supported by the provider** (e.g., Java, Python, .Net). The consumer does not manage or control the underlying cloud infrastructure, network, servers, operating systems, or storage, but the consumer has control over the deployed applications and possibly application hosting environment configurations.

Infrastructure as a Service (IaaS)

The **capability provided to the consumer is to rent processing, storage, networks, and other fundamental computing resources** where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly select networking components (e.g., firewalls, load balancers).

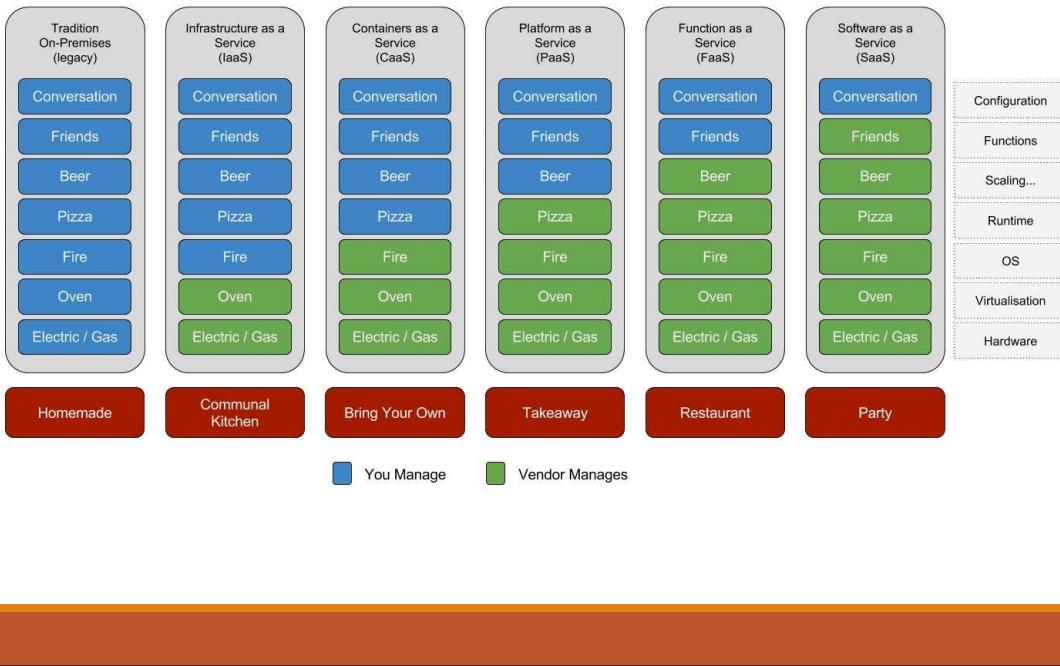
Function as a Service (FaaS)

FaaS is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities **without the complexity of building and maintaining the infrastructure** typically associated with developing and launching an app. Building an application following this model is one way of achieving a "serverless" architecture, and is typically used when building microservices applications.

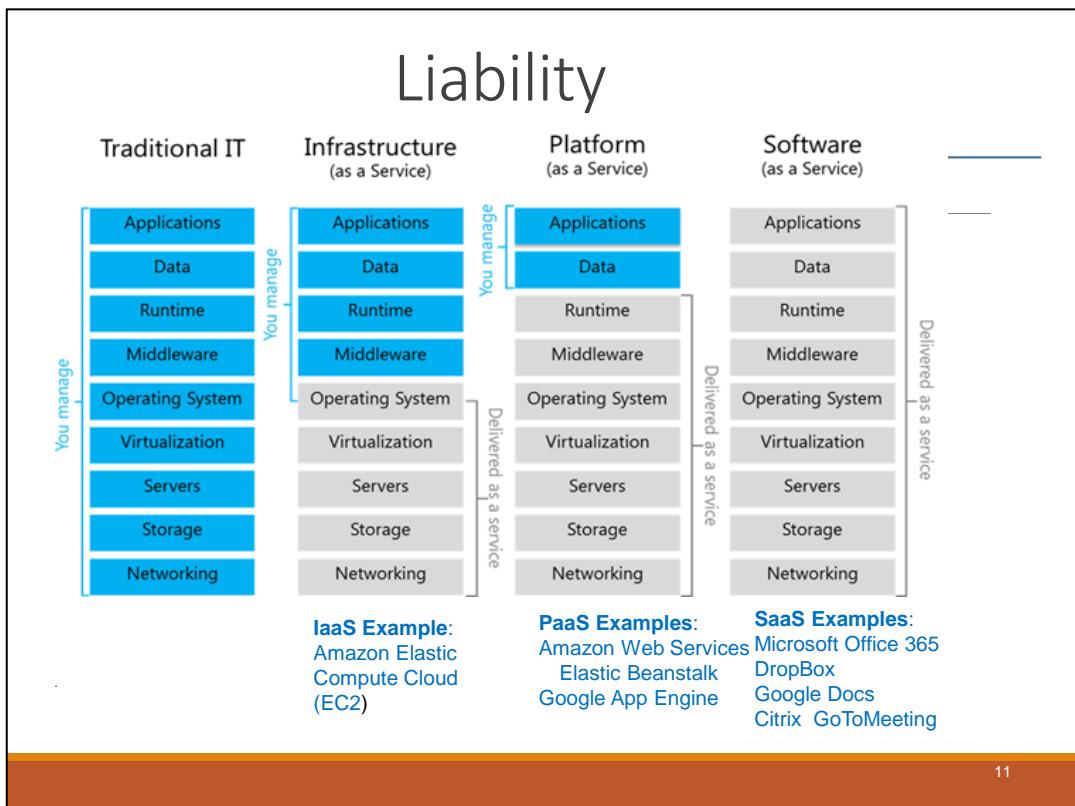


Pizza as a Service 2.0

<http://www.paulkerrison.co.uk>



Liability



Runtime describes software/instructions that are executed *while* your program is running, especially those instructions that you did not write explicitly, but are necessary for the proper execution of your code.

Low-level languages like C have very small (if any) runtime. More complex languages like Objective-C, which allows for dynamic message passing, have a much more extensive runtime.

You are correct that runtime code is library code, but library code is a more general term, describing the code produced by *any* library. Runtime code is specifically the code required to implement the features of the language itself.

Most languages have some form of runtime system that provides an environment in which programs run. This environment may address a number of issues including the layout of application memory, how the program accesses variables, mechanisms for passing parameters between procedures, interfacing with the operating system, and otherwise. The compiler makes assumptions depending on the specific runtime system to generate correct code.

IaaS delivers computer infrastructure

PaaS deliver a computing platform where the developers can develop their own applications.

SaaS is a model of software deployment where the software applications are provided to the customers as a service.

Sometimes a SaaS offering can be run on top of some other cloud provider's IaaS or PaaS offering

For example, DropBox originally ran its offerings on top of the Amazon cloud –see Metz (2016)

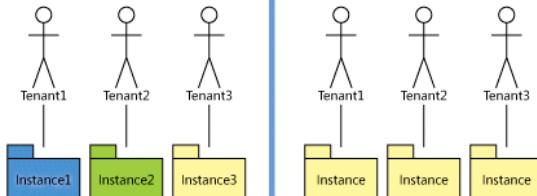
Microsoft Azure allows both IaaS and PaaS operation

Salesforce.com offers Force.com, which allows external developers to create add on applications that integrate with the main salesforce.com offerings

SaaS Maturity Levels

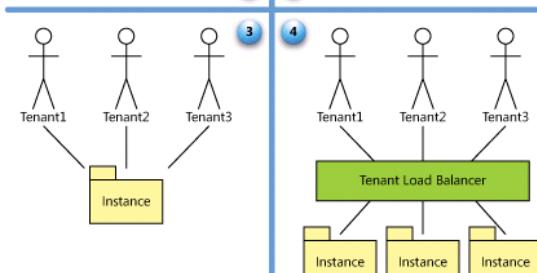
Level 1: Ad-Hoc/Custom

- single tenant model
- a **customized code base** for that particular tenant
- no sharing of anything (code, DB, etc.)



Level 2: Configurable

- single tenant model
- no sharing of anything (code, DB, etc.)
- Instead of customizing the application for a customer (requiring code changes), one allows the user to configure the application through metadata



12

Single Tenant

In a single tenant model each of the tenants will get their own respective instances. There is absolutely no sharing of anything (code, DB, etc.). Each time a new customer (tenant) is added a new (logical) hardware is provisioned and new instance of the product is setup in the allocated environment. This could also be a logical separation of the hardware in the form of separate web sites within the same IIS server. Single tenant model is also commonly referred as Hosted model.

Maturity Level 1 and 2 falls under single tenant model. In both the levels the tenant gets their own instance of the software. However, the primary difference between these levels is that in Level 1 you may even have a customized code base for that particular tenant. Therefore, you may even end up having different code bases for each tenant (as depicted with multiple colors in the above picture). However, from a Tenant's perspective it's immaterial whether it's Level 1 or 2 as long as the software performs the expected functions.

Advantages

Takes less time to roll-out in to SaaS model, since the product does not require going through any changes to support SaaS model.

Overall SaaS transition complexity and cost is going to be less.

Does not require any SaaS Architecture/Engineering expertise. All that is required would be expertise on the hardware front, which is any addressed by IaaS vendors.

Level 1 offers a unique advantage of the ability to provide customized versions to your client. However, this could also be viewed as a disadvantage as you will have to keep maintaining several versions of your product.

Supports non-web native applications. For example, you can use windows citrix to deliver a desktop or client/server application (ex: applications developed in VB) in a SaaS model.

Certain market/customer base does not like the idea of multi-tenancy. For example: a banking customer will not like the idea of sharing their data/environment with other banking organizations. In some cases the security compliance also does not allow sharing. In these instances Single Tenant wins over multi-tenant.

Disadvantages

Maintenance efforts are going to be huge as you will have to maintain multiple code base/environments. For example, if you are going to make a fix then you will have to roll-it out 'n' number of times, where n is the number of tenants supported.

Operational costs are going to be extremely high, particularly over the long run.

Multi-Tenant

Multi-Tenancy is an architecture capability that allows an application/product to recognize tenants (tenants could be users, group of users or organizations) and exhibit functionalities as per the configuration set for the tenants. An on-premise application is typically designed to work for a single organization. Therefore, the very concept of making an on-premise application realize/operate in the context of a particular tenant is a big change. Every single functionality in the product has to be tweaked in order to achieve multi-tenancy. Segregation at the data layer is another big challenge. There are various degrees of multi-tenancy that can be supported. The complexity/effort also depends on the multi-tenancy degree that you are choosing to go with.

Maturity Level 3 and 4 fall under the multi-tenant model. Level 3 and 4 differ only in the level of scalability they can offer. Level 3 is more of a scale-up solution where you can upgrade the hardware to increase the number of tenants that can be supported. Level 4 supports scale-out solution where multiple hardwares can be added so that a load balanced environment can be created. As you can imagine, the decision between Level 3 and Level 4 lies with the amount of tenants that you are expecting to have on the system.

Advantages

Facilitates a cost effective way of delivering SaaS solution.

Huge savings in operational cost, particularly over the long run. This in turn will allow the ISVs to offer an attractive pricing to their customers, therefore creating a snow ball effect.

Enables to achieve higher levels of customer satisfaction.

Roll-out of upgrades/fixes is much easier.

Easier to adopt/implement best practices in the product features as you will be concentrating on only one version of the product.

The design principles of a multi-tenant system offer a high level of maintainability. For example, if a customer requests for few additional fields in one of the pages you can easily add them through custom field's module.

Scalability to expand the number of tenants. For example: Salesforce.com has more than 100,000 tenants running on their multi-tenant system. Can you even imagine how this will work in a single tenant model?

Disadvantages

Due to the amount of changes that has to happen in the product it takes a while to roll-out the solution.

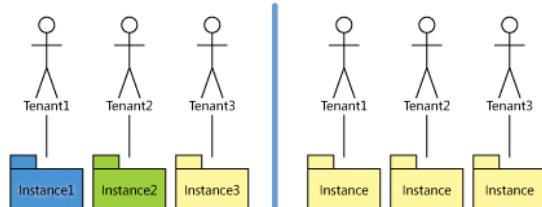
Initial investment to enable multi-tenancy is high. While you will recover this cost in the form of savings in operational expenses it still requires that initial investment and as well the risk involved in achieving the expected business growth.

Requires SaaS architecture expertise, which may not be within the company. Will have to find a strong SaaS partner to guide you in the transition process.

SaaS Maturity Levels

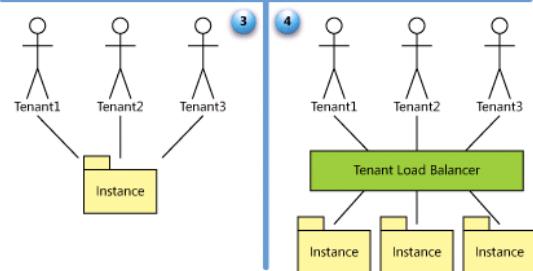
Level 3: Configurable,

- Multi-Tenant-Efficient
- architecture capability that allows an application/product to recognize tenants



Level 4: Scalable,

- Multi-Tenant-Efficient
- architecture capability that allows an application/product to recognize tenants
- Elasticity
- Scalability



13

Single Tenant

In a single tenant model each of the tenants will get their own respective instances. There is absolutely no sharing of anything (code, DB, etc.). Each time a new customer (tenant) is added a new (logical) hardware is provisioned and new instance of the product is setup in the allocated environment. This could also be a logical separation of the hardware in the form of separate web sites within the same IIS server. Single tenant model is also commonly referred as Hosted model.

Maturity Level 1 and 2 falls under single tenant model. In both the levels the tenant gets their own instance of the software. However, the primary difference between these levels is that in Level 1 you may even have a customized code base for that particular tenant. Therefore, you may even end up having different code bases for each tenant (as depicted with multiple colors in the above picture). However, from a Tenant's perspective it's immaterial whether it's Level 1 or 2 as long as the software performs the expected functions.

Advantages

Takes less time to roll-out in to SaaS model, since the product does not require going through any changes to support SaaS model.

Overall SaaS transition complexity and cost is going to be less.

Does not require any SaaS Architecture/Engineering expertise. All that is required would be expertise on the hardware front, which is any addressed by IaaS vendors.

Level 1 offers a unique advantage of the ability to provide customized versions to your client. However, this could also be viewed as a disadvantage as you will have to keep maintaining several versions of your product.

Supports non-web native applications. For example, you can use windows citrix to deliver a desktop or client/server application (ex: applications developed in VB) in a SaaS model.

Certain market/customer base does not like the idea of multi-tenancy. For example: a banking customer will not like the idea of sharing their data/environment with other banking organizations. In some cases the security compliance also does not allow sharing. In these instances Single Tenant wins over multi-tenant.

Disadvantages

Maintenance efforts are going to be huge as you will have to maintain multiple code base/environments. For example, if you are going to make a fix then you will have to roll-it out 'n' number of times, where n is the number of tenants supported.

Operational costs are going to be extremely high, particularly over the long run.

Multi-Tenant

Multi-Tenancy is an architecture capability that allows an application/product to recognize tenants (tenants could be users, group of users or organizations) and exhibit functionalities as per the configuration set for the tenants. An on-premise application is typically designed to work for a single organization. Therefore, the very concept of making an on-premise application realize/operate in the context of a particular tenant is a big change. Every single functionality in the product has to be tweaked in order to achieve multi-tenancy. Segregation at the data layer is another big challenge. There are various degrees of multi-tenancy that can be supported. The complexity/effort also depends on the multi-tenancy degree that you are choosing to go with.

Maturity Level 3 and 4 fall under the multi-tenant model. Level 3 and 4 differ only in the level of scalability they can offer. Level 3 is more of a scale-up solution where you can upgrade the hardware to increase the number of tenants that can be supported. Level 4 supports scale-out solution where multiple hardwares can be added so that a load balanced environment can be created. As you can imagine, the decision between Level 3 and Level 4 lies with the amount of tenants that you are expecting to have on the system.

Advantages

Facilitates a cost effective way of delivering SaaS solution.

Huge savings in operational cost, particularly over the long run. This in turn will allow the ISVs to offer an attractive pricing to their customers, therefore creating a snow ball effect.

Enables to achieve higher levels of customer satisfaction.

Roll-out of upgrades/fixes is much easier.

Easier to adopt/implement best practices in the product features as you will be concentrating on only one version of the product.

The design principles of a multi-tenant system offer a high level of maintainability. For example, if a customer requests for few additional fields in one of the pages you can easily add them through custom field's module.

Scalability to expand the number of tenants. For example: Salesforce.com has more than 100,000 tenants running on their multi-tenant system. Can you even imagine how this will work in a single tenant model?

Disadvantages

Due to the amount of changes that has to happen in the product it takes a while to roll-out the solution.

Initial investment to enable multi-tenancy is high. While you will recover this cost in the form of savings in operational expenses it still requires that initial investment and as well the risk involved in achieving the expected business growth.

Requires SaaS architecture expertise, which may not be within the company. Will have to find a strong SaaS partner to guide you in the transition process.

Deployment Models

There are four primary cloud deployment models:

- Public Cloud
- Private Cloud
- Community Cloud
- Hybrid Cloud

Their differences lie primarily in the scope and access of published cloud services, as they are made available to service consumers.

Private cloud: The cloud infrastructure is operated solely for an organization. It may exist on premise or off premise.

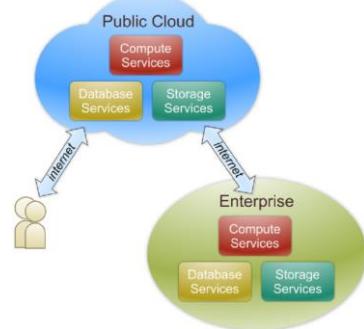
Public cloud: Mega-scale cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

Hybrid cloud: Composition of two or more clouds (private or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability.

Public Cloud

Public cloud definition

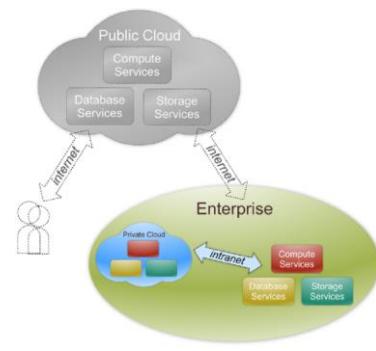
- The cloud infrastructure is made available to the **general public** or a **large industry group** and is owned by an organization selling cloud services.
- Also known as external cloud or multi-tenant cloud, this model essentially represents a cloud environment that is openly accessible.
- Basic characteristics:
 - Homogeneous infrastructure
 - Common policies
 - Shared resources and multi-tenant
 - Leased or rented infrastructure
 - Economies of scale



Private Cloud

Private cloud definition

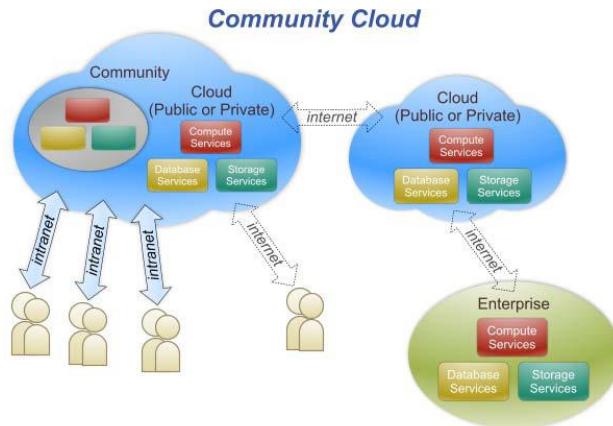
- The cloud infrastructure is **operated solely for single organization**. It may be managed by the organization or a third party and may exist on premise or off premise.
- Also referred to as internal cloud or on-premise cloud, a private cloud intentionally limits access to its resources to service consumers that belong to the same organization that owns the cloud.
- Basic characteristics :
 - Heterogeneous infrastructure
 - Customized and tailored policies
 - Dedicated resources
 - In-house infrastructure
 - End-to-end control



Community Cloud

Community cloud definition

- The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations).



18

They are multi-tenant platforms that enable different organizations to work on a shared platform.

A community cloud usually involves similar industries and complimentary businesses with shared goals all using the same hardware. By sharing the infrastructure between multiple companies, community cloud installations are able to save their members money. Data is still segmented and kept private, except in areas where shared access is agreed upon and configured.

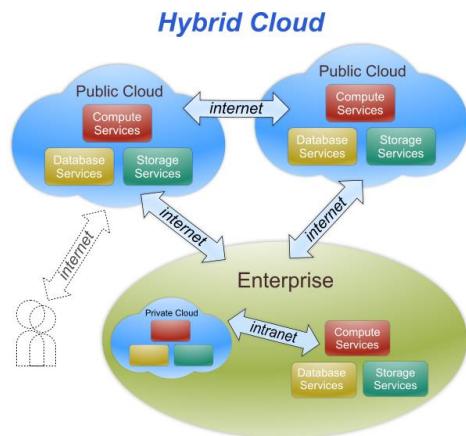
The main benefits are the shared costs and the increase in opportunities to collaborate in real-time across the same infrastructure. Uniformity of [best practices](#) will help to increase the overall security and efficiency of these setups, so they rely quite heavily on effective cooperation between tenants.

A good example is the **U.S.-based dedicated IBM SoftLayer cloud for federal agencies**.

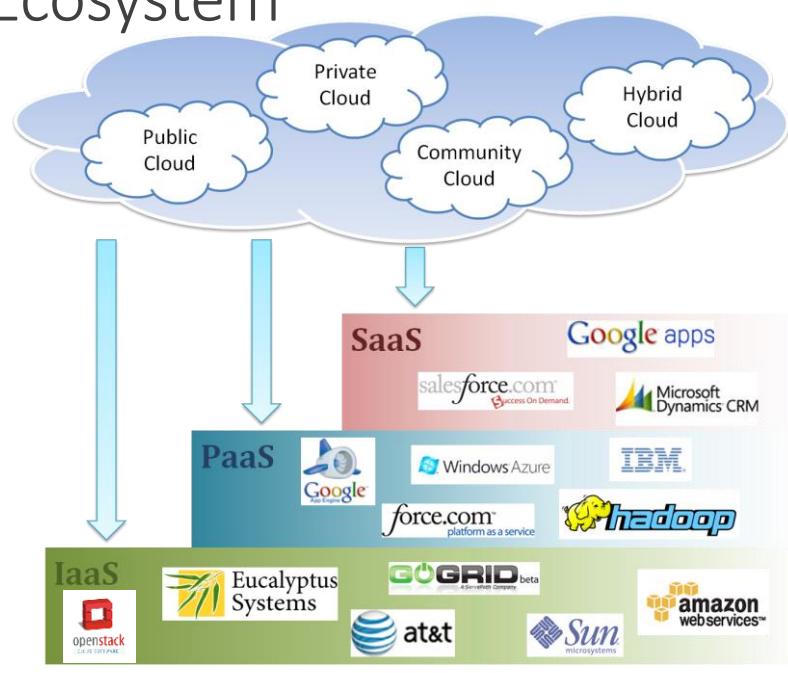
Hybrid Cloud

Hybrid cloud definition

- The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds).



Cloud Ecosystem



20

Open Source Platforms Supporting IaaS

Cloud	History	License	Language Written in	Installation Difficulty
OpenStack	Founded by Rackspace and NASA in 2010	Apache v2.0	Python	difficult
CloudStack	Released by Citrix into Apache in 2011	Apache v2.0	Java,C	Medium
Eucalyptus	Began at Rice University and UCSB, now sponsored by HP. Has formal compatibility agreement with AWS	GPL v 3.0	Java,C	Medium
OpenNebula	Sponsored by OpenNebula Systems	Apache v2.0	C++, C, Ruby, Java, shell script, lex, yacc	Medium

Security Issues

Evergreen...

- Data Loss
- Downtimes
- Phishing
- Password Cracking
- Botnets and Other Malware

Data Loss

"Regrettably, based on Microsoft/Danger's latest recovery assessment of their systems, we must now inform you that personal information stored on your device—such as contacts, calendar entries, to-do lists or photos—that is no longer on your Sidekick almost certainly has been lost as a result of a server failure at Microsoft/Danger."

<http://arstechnica.com/business/news/2009/10/t-mobile-microsoftdanger-data-loss-is-bad-for-the-cloud.ars>

23

<http://arstechnica.com/business/news/2009/10/t-mobile-microsoftdanger-data-loss-is-bad-for-the-cloud.ars>

Downtimes

The screenshot shows a news article from NetworkWorld.com. The top navigation bar includes links for News, Blogs & Columns, Subscriptions, Videos, Events, and More. Below the navigation is a secondary menu for Security, LAN & WAN, UC / VoIP, Infrastructure Mgmt, Wireless, Software, Data Center, and Cloud Computing. The main content area features a large heading "RESOLVED: Windows Azure Outage". The article discusses a power failure at Rackspace's Dallas facility that took customer servers offline last week. It includes a comment from Steve Marx and several updates from the Windows Azure blog. Another section covers an outage at Amazon S3, mentioning multiple data centers and system load reduction. A sidebar titled "Related Content" lists five items. The bottom right corner of the page has a small "24" icon.

24

Phishing

"hey! check out this funny blog about you..."

The screenshot shows a Mozilla Firefox window with the title "Twitter: What are you doing? - Mozilla Firefox". The address bar displays the URL <http://twitter.access-logins.com/login/>. The main content area shows the Twitter login interface with fields for "user name or email" and "password". A red oval highlights the URL in the address bar. To the right of the browser window, there is a sidebar with a list of compromised Hotmail passwords, including:

- 440profitemailers@Hotmail.com
- HERBALVIAGRA@Barry_s...
- 426profitmasters@20h...
- 072Homesbiz@mweb.com
- PERMANENTGROW@Lee_th...
- 254PHARMACYonline@ez...
- ...VIAGRA@Lee_the...

Below this list, a caption reads: "Attack: Spam emails may have been responsible. Photograph: Roger Tooth".

At the bottom of the sidebar, it says: "Microsoft has confirmed that the publication of thousands of Hotmail passwords was the result of a phishing attack against users of the popular email service." and "Precise details of the strike, which was first uncovered on Monday, remain unclear. But in a statement, the American software company said".

At the very bottom of the page, there is a small number "25".

<http://news.cnet.com/twitter-phishing-scam-may-be-spreading/>

Password Cracking

PCWorld News Reviews How-To's Downloads Shop & Compare Apps Business Center

PCWorld Business Center Discover news, issues, and products for your business

Security Alert Practical security advice More Security Alert RSS All Blogs Enter your email to get t

69 2 Like 40 4 Comments +9 Recommendations Email Print BUSINESS CENTER Jan 10, 2011 8:31 pm

Cloud Computing Used to Hack Wireless Passwords

German security researcher Thomas Roth has found an innovative use for cloud computing: cracking wireless networks that rely on pre-shared key passphrases, such as those found in homes and smaller businesses.

SIMILAR ARTICLES:

- Gække Hack Exposes Ridiculous Password Habits
- What Cloud Computing Means For the Real World
- Can Encrypted Blobs Help With Computing Clouds?
- Virtualization is Key to Cloud Security
- 7 Ways to Avoid Getting Hacked by Anonymous
- Password Reuse Is All Too Common, Research Shows

In other words, this isn't a clever or elegant hack, and it doesn't rely on a raw in wireless networking technology. Roth's software merely generates millions of passphrases, encrypts them, and sees if they allow access to the network.

However, employing the theoretically infinite resources of cloud computing to brute force a password is the clever part.

Purchasing the computers to run such a crack would cost tens of thousands of dollars, but Roth claims that a typical wireless password can be guessed by EC2 and his software in about six minutes. He proved this by hacking networks in the area where he lives. The type of EC2 computers used in the attack costs 28 cents per minute, so \$1.68 is all it could take to lay open a wireless network.

stacksmashing.net

Home Imprint

← Small Patch For The GET 'Java Applet' Payload

Cracking Passwords In The Cloud: Getting The Facts Straight →

Cracking Passwords In The Cloud: Amazon's New EC2 GPU Instances

Posted on 15 November 2010 by Thomas Roth

Update: Great article about this at Threatpost! This also got slashdotted, featured on Tech News Today, and there's a ZDNet article about this.

Update: Because of the huge impact I have clarified some things here.

As of today, Amazon EC2 is providing what they call "Cluster GPU Instances": An instance in the Amazon cloud that provides you with the power of two NVIDIA Tesla "Fermi" M2050 GPUs. The exact specifications look like this:

22 GB of memory
33.5 EC2 Compute Units (2 x Intel Xeon X5570, quad-core "Nehalem" architecture)
2 x NVIDIA Tesla "Fermi" M2050 GPUs
1690 GB of instance storage
64-bit platform
I/O Performance: Very High (10 Gigabit Ethernet)
API name: cgt.4large

GPUs are known to be the best hardware accelerator for cracking passwords, so I decided to give it a try: How fast can this instance type be used to crack SHA1 hashes?

Using the CUDA-Multiforce, I was able to crack all hashes from this file with a password length from 1-6 in only 49 Minutes (1 hour costs 2.10\$ by the way.)

Compute done: Reference time 2950.1 seconds
Stepping rate: 249.2M MD4/s
Search rate: 3488.4M NTLM/s

This just shows one more time that SHA1 for password hashing is deprecated – You really don't want to use it anymore! Instead, use something like crypt or PBKDF2! Just imagine

Botnets and Malware

CA Security Advisor Research Blog

Get information on the latest threats

The screenshot shows a news article from GSN: Government Security News. The headline is "Treasury Dept. has cloud hacked". The article discusses how the Treasury Department's website was hacked last week, causing issues for the Bureau of Engineering and Printing - the agency responsible for printing U.S. dollars - down from May 3 to May 7. The Treasury had moved to a cloud platform last year, and the department blamed its cloud computing provider (the Treasury's Web site is hosted by Network Solutions) for the incident. A statement released May 4 from the Treasury Department said: "The Bureau of Engineering and Printing (BEP) entered the cloud computing arena last year. The hosting company used by BEP had an intrusion and as a result of that intrusion, numerous websites (BEP and non-BEP) were affected. On May 3, the Treasury Government Security Operations Center was made aware of the problem and subsequently notified BEP. BEP has four Internet address URLs all pointing to one public website. Those URLs are BEP.gov; BEP.treas.gov; Moneyfactory.gov and Moneyfactory.com. BEP has since suspended the website. Through discussions with the provider, BEP is aware of the remediation steps required to restore the site and is currently working toward resolution." Roger Thompson, chief research officer at security software vendor ArcSight, based in Chelmsford, Mass., first noticed the hack, and reported it to the FBI. Thompson revealed that the hackers had added a tiny snippet of a virtually undetectable frame HTML code that redirected visitors to a Ukrainian Web site. From there, a variety of Web-based attacks were launched using an easy-to-purchase malicious toolkit, called the Eleemosyne exploit. Once users were affected, redirecting them to legitimate sites became a common tactic for attacks, making it difficult for law enforcement to track the perpetrators. For instance, "the Eleemosyne exploit costs only \$5 - even the most minimally talented hacker can exploit flaws in Microsoft Internet Explorer, Firefox and Adobe Acrobat Reader. The widespread problem of low cost hacking that takes advantage of this commonly used software was highlighted in the 2010 Symantec report."

Despite the inherent risks involved in cloud platforms, IT experts tend to agree that the government would [see more benefits](#) from using them, rather than not, and have encouraged government agencies to move towards the cloud in recent months.

"I am not going to say this will scare users away from cloud computing," says Thomas Kraft. "But it definitely brings into clear focus the issues surrounding security in the cloud."

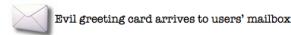
Zeus "in-the-cloud"

Published: December 09 2009, 04:39 AM
by Methusela Cebrian Ferrer

A new wave of a Zeus bot (Zbot) variant was spotted taking advantage of Amazon EC2's cloud-based services for its C&C (command and control) functionalities.



This notable scheme is a highlight from the latest spammed executable "xmas2.exe" (63,488 bytes), for which we have recently published blog titled "Christmas is knocking on the door, so does the malware".



Entices users to click a malicious URL which links to a hacked legitimate website perpetrated for criminal activity such as serving Zeus bot variant.

Once executed, the Zeus bot variant will communicate to its C&C server, which in this case is controlled using "in-the-cloud" based services.

[Figure 01 - Zeus displays cyber-criminal activities]

Action	URL	Details
GET	http://ec2-170-170-170-170.compute-1.amazonaws.com/zeu/config.bn	svchost.exe [s]
POST	http://ec2-170-170-170-170.compute-1.amazonaws.com/zeu/gate.php	svchost.exe [s]
POST	http://ec2-170-170-170-170.compute-1.amazonaws.com/zeu/gate.php	svchost.exe [s]
POST	http://ec2-170-170-170-170.compute-1.amazonaws.com/zeu/gate.php	svchost.exe [s]

As shown in Figure 03, the Zeus bot variant injects code into the system processes (such as svchost.exe) and connects to its cloud-server [Figure 02] for configuration (config.bn) of the master for its criminal activity.

<http://community.ca.com/blogs/securityadvisor/archive/2009/12/09/zeus-in-the-cloud.aspx>

Virtualization Security

Features

- Isolation
- Snapshots

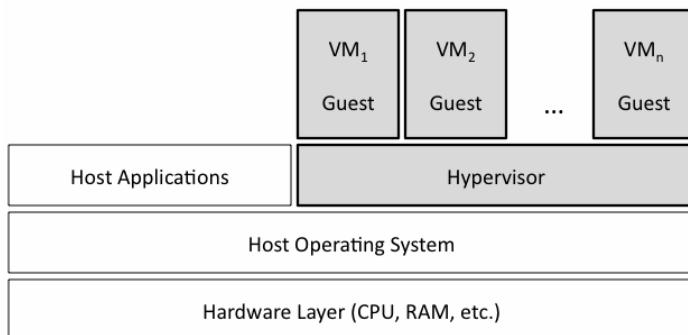
Issues

- State Restore
- Hypervisor vulnerabilities
- Scaling

Features: Isolation

Using a VM for each application provides isolation

- Better isolation than running 2 apps on same server.
- Worse isolation than running on 2 physical servers



A Survey on the Security of Virtual Machines

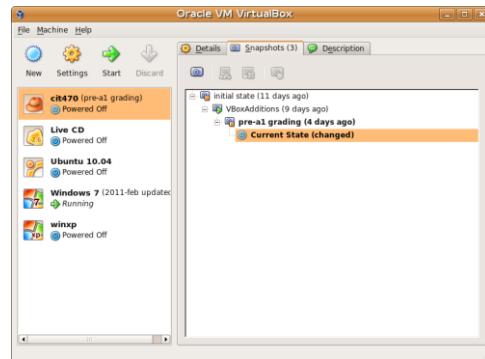
<http://www1.cse.wustl.edu/~jain/cse571-09/ftp/vmsec/index.html>

<http://www1.cse.wustl.edu/~jain/cse571-09/ftp/vmsec/index.html>

Remember Spectre and Meltdown

Features: Snapshot

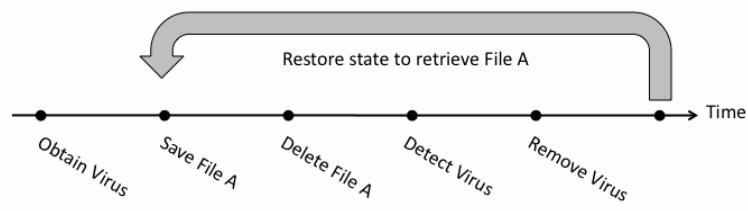
- VMs can record state.
- In event of security incident, revert VM back to an uncompromised state.
- Must be sure to patch VM to avoid recurrence of compromise.



30

State Restore

- VMs can be restored to an infected or vulnerable state using snapshots.
- Patching becomes undone.
- Worms persist at low level forever due to reappearance of infected and vulnerable VMs.



31

<http://www1.cse.wustl.edu/~jain/cse571-09/ftp/vmsec/index.html>

Hypervisor Vulnerabilities

Vulnerability consequences

- Guest code execution with privilege
- VM Escape (Host code execution)

Home > Support & Downloads > Support Resources > Security Advisories

VMware Security Advisories (VMSAs)

VMSA-2009-0006

VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability

VMware Security Advisory

Advisory ID: VMSA-2009-0006
Synopsis: VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability
Issue date: 2009-04-10
Updated on: 2009-04-10 (initial release of advisory)
CVE numbers: CVE-2009-1244

1. Summary
Updated VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability.
2. Relevant releases
VMware Workstation 6.5.1 and earlier,
VMware Player 2.5.1 and earlier,
VMware Server 2.0.1 and earlier,
VMware Server 2.0,
VMware Server 1.0.8 and earlier,
VMware Fusion 2.0.3 and earlier,
VMware ESXi 3.5 without patch ESXi350-200904201-0-SG,
VMware ESX 3.5 without patch ESX350-200904201-SG,
VMware ESX 3.0.3 without patch ESX303_2009040403-SG,
VMware ESX 3.0.2 without patch ESX-1009421.
NOTE: General Support for Workstation version 5.x ended on 2009-03-19.
Users should plan to upgrade to the latest Workstation version 6.x release.
Extended support for ESX 3.0.2 Update 1 ends on 2009-08-08.
Users should plan to upgrade to ESX 3.0.3 and preferably to the newest release available.
3. Problem Description
a. Host code execution vulnerability from a guest operating system
A critical vulnerability in the virtual machine display function might allow a guest operating system to run code on the host.
This issue is different from the vulnerability in a guest virtual display driver described in security advisory VMSA-2009-0005 on 2009-04-03. That vulnerability can cause a potential denial of service and is identified by CVE-2008-4916.
The Common Vulnerabilities and Exposures project (cve.mitre.org) has assigned the name CVE-2009-1244 to this issue.

Xen CVE-2008-1943
VBox CVE-2010-3583

CVE: Common Vulnerabilities and Exposures

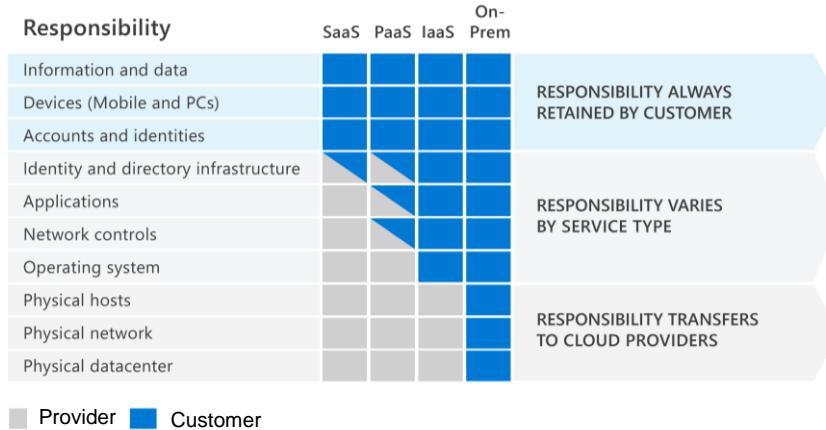
Scaling of VMs

- Growth in physical machines limited by budget and setup time.
- Adding a VM is easy as copying a file, leading to explosive growth in VMs.
- Rapid scaling can exceed capacity of organization's security systems.

New Security Issues

- Accountability
- No Security Perimeter
- Larger Attack Surface
- New Side Channels
- Lack of Auditability
- Regulatory Compliance

Shared responsibility model



The *shared responsibility model* identifies which security tasks are handled by the cloud provider, and which security tasks are handled by you, the customer.

In organizations running only on-premises hardware and software, the organization is 100 percent responsible for implementing security and compliance. With cloud-based services, that responsibility is shared between the customer and the cloud provider.

The responsibilities vary depending on where the workload is hosted:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)
- On-premises datacenter (On-prem)

The shared responsibility model makes responsibilities clear. When organizations move data to the cloud, some responsibilities transfer to the cloud provider and some to the customer organization.

The following diagram illustrates the areas of responsibility between the customer and the cloud provider, according to where data is held.

On-premises datacenters

In an on-premises datacenter, you have responsibility for everything from physical security to encrypting sensitive data.

Infrastructure as a Service (IaaS)

Of all cloud services, IaaS requires the most management by the cloud customer.

With IaaS, you're using the cloud provider's computing infrastructure. The cloud customer isn't responsible for the physical components, such as computers and the network, or the physical security of the datacenter. However, the cloud customer still has responsibility for software components such as operating systems, network controls, applications, and protecting data.

Platform as a Service (PaaS)

PaaS provides an environment for building, testing, and deploying software applications. The goal of PaaS is to help you create an application quickly without managing the underlying infrastructure. With PaaS, the cloud provider manages the hardware and operating systems, and the customer is responsible for applications and data.

Software as a Service (SaaS)

SaaS is hosted and managed by the cloud provider, for the customer. It's usually licensed through a monthly or annual subscription. Microsoft 365, Skype, and Dynamics CRM Online are all examples of SaaS software. SaaS requires the least amount of management by the cloud customer. The cloud provider is responsible for managing everything except data, devices, accounts, and identities.

For all cloud deployment types you, the cloud customer, own your data and identities. You're responsible for protecting the security of your data and identities, and on-premises resources.

In summary, responsibilities always retained by the customer organization include:

- Information and data
- Devices (mobile and PCs)
- Accounts and identities

The benefit of the shared responsibility model is that organizations are clear about their responsibilities, and those of the cloud provider.

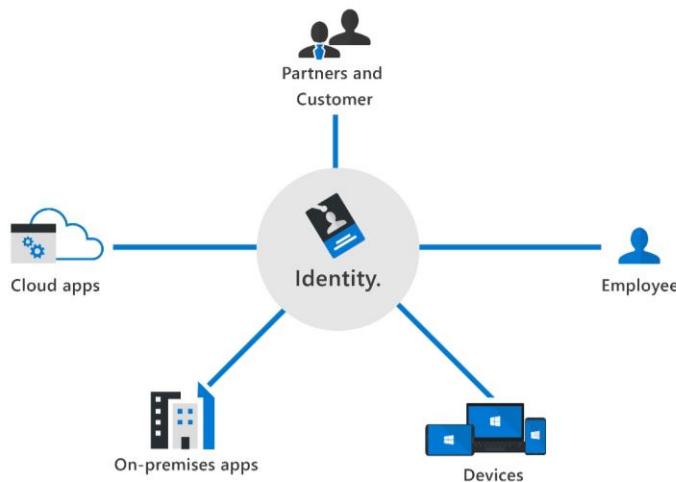
No Security Perimeter

- Little control over physical or network location of cloud instance VMs
- Network access must be controlled on a host by host basis.



3

Identity as the primary security perimeter



Digital collaboration has changed. Your employees and partners now need to collaborate and access organizational resources from anywhere, on any device, and without affecting their productivity. There has also been an acceleration in the number of people working from home.

Enterprise security needs to adapt to this new reality. The security perimeter can no longer be viewed as the on-premises network. It now extends to:

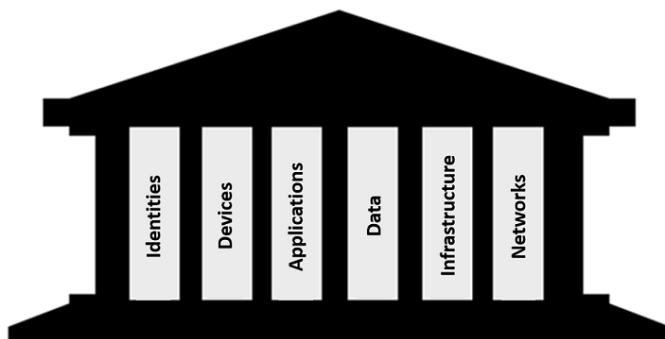
- SaaS applications for business-critical workloads that might be hosted outside the corporate network.
- The personal devices that employees are using to access corporate resources (BYOD, or bring your own device) while working from home.
- The unmanaged devices used by partners or customers when interacting with corporate data or collaborating with employees
- IoT devices installed throughout your corporate network and inside customer locations.

The traditional perimeter-based security model is no longer enough. Identity has become the new security perimeter that enables organizations to secure their assets.

But what do we mean by an identity? An identity is how someone or something can be verified and authenticated to be who they say they are. An identity may be associated with a user, an application, a device, or something else.

Zero Trust guiding principles

Zero Trust Methodology
“Trust no one, verify everything”



<https://docs.microsoft.com/it-it/learn/modules/describe-security-concepts-methodologies/2-describe-zero-trust-methodology>

Zero Trust assumes everything is on an open and untrusted network, even resources behind the firewalls of the corporate network. The Zero Trust model operates on the principle of **“trust no one, verify everything.”**

Attackers' ability to bypass conventional access controls is ending any illusion that traditional security strategies are sufficient. By no longer trusting the integrity of the corporate network, security is strengthened.

In practice, this means that we no longer assume that a password is sufficient to validate a user but add multi-factor authentication to provide additional checks. Instead of granting access to all devices on the corporate network, users are allowed access only to the specific applications or data that they need.

The Zero Trust model has three principles which guide and underpin how security is implemented. These are: verify explicitly, least privilege access, and assume breach.

- Verify explicitly.** Always authenticate and authorize based on the available data points, including user identity, location, device, service or workload, data classification, and anomalies.

- Least privileged access.** Limit user access with just-in-time and just-enough access (JIT/JEA), risk-based adaptive policies, and data protection to protect

both data and productivity.

•**Assume breach.** Segment access by network, user, devices, and application. Use encryption to protect data, and use analytics to get visibility, detect threats, and improve your security.

Six foundational pillars

In the Zero Trust model, all elements work together to provide end-to-end security. These six elements are the foundational pillars of the Zero Trust model:

•**Identities** may be users, services, or devices. When an identity attempts to access a resource, it must be verified with strong authentication, and follow least privilege access principles.

•**Devices** create a large attack surface as data flows from devices to on-premises workloads and the cloud. Monitoring devices for health and compliance is an important aspect of security.

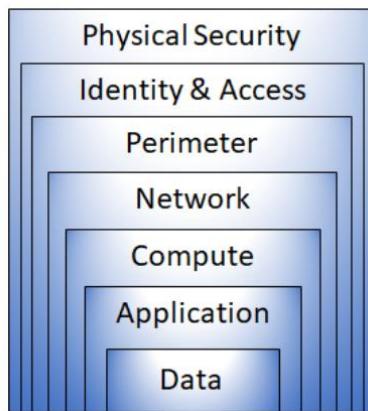
•**Applications** are the way that data is consumed. This includes discovering all applications being used, sometimes called Shadow IT because not all applications are managed centrally. This pillar also includes managing permissions and access.

•**Data** should be classified, labeled, and encrypted based on its attributes. Security efforts are ultimately about protecting data, and ensuring it remains safe when it leaves devices, applications, infrastructure, and networks that the organization controls.

•**Infrastructure**, whether on-premises or cloud based, represents a threat vector. To improve security, you assess for version, configuration, and JIT access, and use telemetry to detect attacks and anomalies. This allows you to automatically block or flag risky behavior and take protective actions.

•**Networks** should be segmented, including deeper in-network micro segmentation. Also, real-time threat protection, end-to-end encryption, monitoring, and analytics should be employed.

Defense in depth



Defense in depth uses a layered approach to security, rather than relying on a single perimeter. A defense in-depth strategy uses a series of mechanisms to slow the advance of an attack. Each layer provides protection so that, if one layer is breached, a subsequent layer will prevent an attacker getting unauthorized access to data.

Example layers of security might include:

- **Physical** security such as limiting access to a datacenter to only authorized personnel.
- **Identity and access** security controls, such as multi-factor authentication or condition-based access, to control access to infrastructure and change control.
- **Perimeter** security including distributed denial of service (DDoS) protection to filter large-scale attacks before they can cause a denial of service for users.
- **Network** security, such as network segmentation and network access controls, to limit communication between resources.
- **Compute** layer security such as securing access to virtual machines either on-premises or in the cloud by closing certain ports.
- **Application** layer security to ensure applications are secure and free of security vulnerabilities.
- **Data** layer security including controls to manage access to business and customer data and encryption to protect data.

Examples of Cloud Services

- Amazon EC2
- Amazon S3
- Amazon RDS
- MS Azure
- OpenStack

Amazon Cloud: EC2

Amazon Elastic Compute Cloud (Amazon EC2) is a **web service** that provides **resizeable computing capacity**—literally, servers in Amazon's data centers—that you use to build and host your software systems. You can access the components and features that EC2 provides using a web-based GUI, command line tools, and APIs.

With EC2, you use and **pay for only the capacity that you need**. This eliminates the need to make large and expensive hardware purchases, reduces the need to forecast traffic, and enables you to automatically **scale your IT resources** to deal with changes in requirements or spikes in popularity related to your application or service.

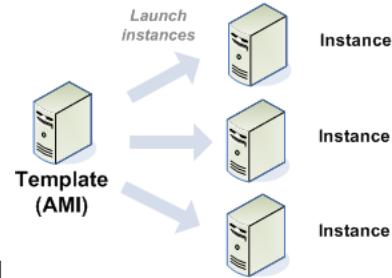
Components of EC2: **Amazon Machine Images** and **Instances**, **Regions and Availability Zones**, **Storage**, **Databases**, **Networking and Security**, **Monitoring**, **Auto-Scaling** and **Load Balancing**, **AWS Identity and Access Management**.

Amazon Cloud EC2: AMI

An *Amazon Machine Image (AMI)* is a template that contains a software configuration (operating system, application server, and applications). From an AMI, you launch *instances*, which are running copies of the AMI. You can launch multiple instances of an AMI, as shown in the figure.

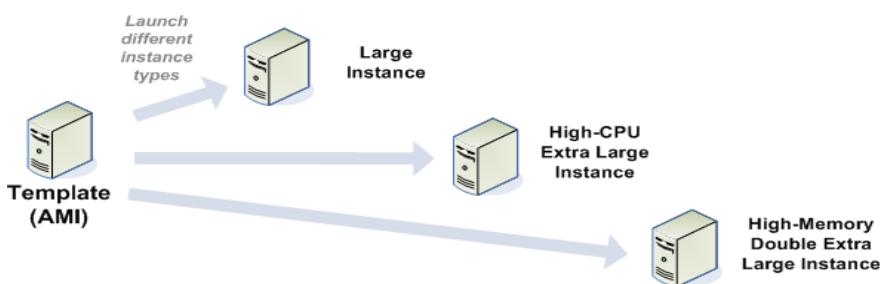
Your instances keep running until you stop or you terminate them, or until they fail. If an instance fails, you can launch a new one from the AMI.

You can use a single AMI or multiple AMIs depending on your needs. From a single AMI, you can launch different *types of instances*.



Amazon Cloud EC2: AMI

- An *instance type* is essentially a hardware archetype. As illustrated in the following figure, you select a particular instance type based on the amount of memory and computing power you need for the application or software that you plan to run on the instance.
- Amazon publishes many AMIs that contain common software configurations for public use. In addition, members of the AWS developer community have published their own custom AMIs.



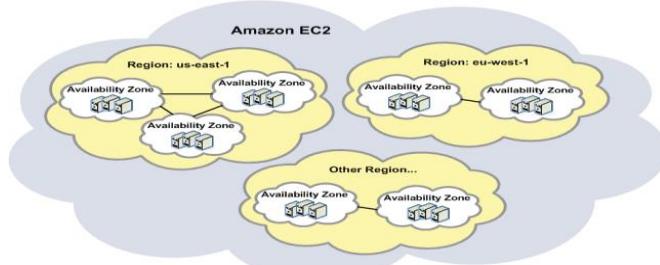
43

example, if your application is a web site or web service, your AMI could be preconfigured with a web server, the associated static content, and the code for all dynamic pages. Alternatively, you could configure your AMI to install all required software components and content itself by running a bootstrap script as soon as the instance starts. As a result, after launching the AMI, your web server will start and your application can begin accepting requests.

Amazon EC2: Regions and Availability Zones

Amazon has data centers in different areas of the world (for example, North America, Europe, and Asia). Correspondingly, Amazon EC2 is available to use in different **Regions**. By launching instances in separate Regions, you can design your application to be closer to specific customers or to meet legal or other requirements.

Each Region contains multiple distinct locations called **Availability Zones**. Each Availability Zone is **engineered to be isolated from failures** in other Availability zones and to provide **inexpensive, low-latency network connectivity to other zones in the same Region**. By launching instances in separate Availability Zones, you can protect your applications from the failure of a single location.



Amazon Storage Services

- Instance Store
- Amazon EC2 Storage
- Amazon S3 Storage

Instance Store

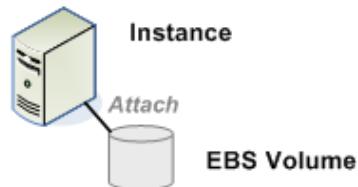
All instance types, with the exception of Micro instances, offer instance store. This is storage that doesn't persist if the instance is stopped or terminated. Instance store is an option for inexpensive temporary storage. You can use instance store volumes if you don't require data persistence.

Amazon Cloud EC2: Storage

Amazon EBS

Amazon EBS volumes are the recommended storage option for the majority of use cases. Amazon EBS provides the instances with persistent, **block-level storage**. Amazon EBS volumes are essentially hard disks that you can attach to a running instance.

Amazon EBS is particularly suited for applications that require a database, file system, or access to raw block-level storage.

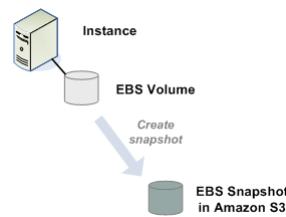


46

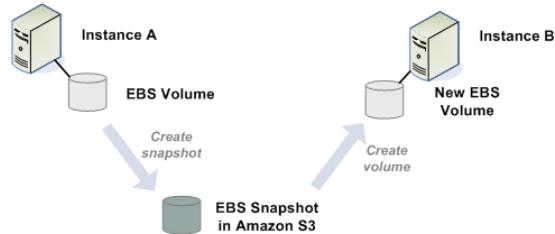
EBS: Elastic Block Store. You can attach multiple volumes to an instance.

Amazon Cloud EC2: Storage

To keep a back-up copy, you can create a snapshot of the volume. As illustrated in the following figure, snapshots are stored in Amazon S3.



You can create a new Amazon EBS volume from a snapshot, and attach it to another instance, as illustrated in the following figure.



Amazon Cloud EC2: Storage

You can also detach a volume from an instance and attach it to a different one, as illustrated in the following figure.



Amazon Cloud S3

Amazon S3

Amazon S3 is storage for the Internet. It provides a simple web service interface that enables you to store and retrieve any amount of data from anywhere on the web. **Type of Object Storage.**

49

Amazon S3 Functionality

1. **Write, read, and delete** objects containing from 1 byte to 5 terabytes of data each.
2. The number of objects you can store is **unlimited**.
3. Each object is stored in a bucket and retrieved via a **unique, developer-assigned key**.
4. A bucket can be stored in one of several Regions. You can choose a Region to **optimize for latency, minimize costs**, or address regulatory requirements.
5. Objects stored in a Region **never leave the Region** unless you transfer them out. For example, objects stored in the EU (Ireland) Region never leave the EU.
6. **Authentication** mechanisms are provided to ensure that data is kept secure from unauthorized access. Objects can be made private or public, and rights can be granted to specific users.
7. Options for **secure data upload/download and encryption** of data at rest are provided for additional data protection.
8. Uses standards-based **REST and SOAP interfaces** designed to work with any Internet-development toolkit.

Amazon Cloud S3: Use Cases

Content Storage, Distribution and Sharing

- Amazon S3 can store a variety of content ranging from web applications to media files. A user can offload an entire storage infrastructure onto the cloud.

Big Data Storage for Data Analysis

- Whether a user is storing pharmaceutical data for analysis, financial data for computation and pricing, or photo images for resizing, Amazon S3 can be used to store the original content. The user can then send this content to Amazon EC2 for computation, resizing, or other large scale analytics – **without incurring any data transfer charges for moving the data between the services.**

Backup, Archiving and Disaster Recovery

- The Amazon S3 solution offers a scalable and secure solution for backing up and archiving critical data.

50

Using Amazon S3 is easy. To get started you:

Create a Bucket to store your data. You can choose a Region where your bucket and object(s) reside to optimize latency, minimize costs, or address regulatory requirements.

Upload Objects to your Bucket. Your data is durably stored and backed by the Amazon S3 Service Level Agreement.

Using Amazon S3 is easy. To get started you:

Create a Bucket to store your data. You can choose a Region where your bucket and object(s) reside to optimize latency, minimize costs, or address regulatory requirements.

Upload Objects to your Bucket. Your data is durably stored and backed by the Amazon S3 Service Level Agreement.

Optionally, set access controls. You can grants others access to your data from anywhere in the world.

Amazon Cloud: Databases

If the application running on EC2 needs a database, the common ways to implement a database for the application are:

- Use Amazon Relational Database Service (Amazon RDS) to get a managed relational database in the cloud
- Launch an instance of a database AMI, and use that EC2 instance as the database

Amazon RDS offers the advantage of handling database management tasks, such as patching the software, backing up and storing the backups

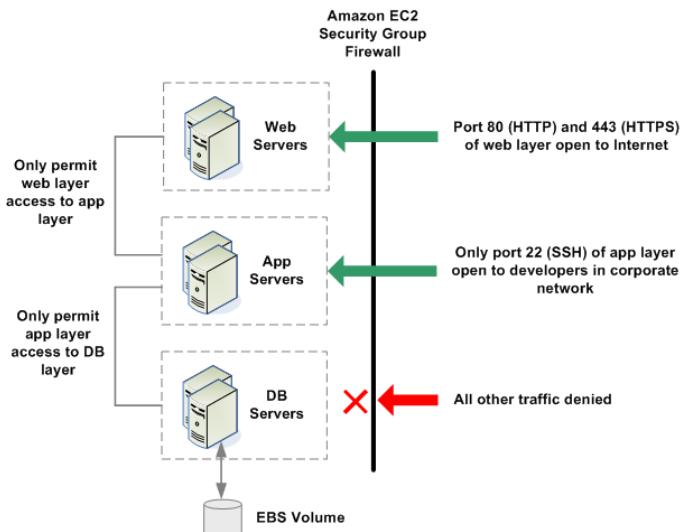
Amazon Cloud: Networking and Security

- Each instance is launched into the Amazon EC2 network space and assigned a **public IP address**. If an instance fails and a replacement instance is launched, the **replacement will have a different public IP address** than the original.
- *Security groups* are used to **control access to user instances**. These are analogous to an inbound network **firewall** that allows a user to specify the protocols, ports, and source IP ranges that are allowed to reach user instances.
- A user can create **multiple security groups** and assign different rules to each group. Each instance can be assigned to one or more security groups, and the rules determine which traffic is allowed in to the instance. A security group can be configured so that only specific IP addresses or specific security groups have access to the instance.

52

However, your application might need a static IP address. Amazon EC2 offers *elastic IP addresses* for those situations. For more information, see [Instance IP Addresses](#).

Amazon Cloud: Networking and Security



53

The figure shows a basic three-tier web-hosting architecture running on Amazon EC2 instances. Each layer has a different security group (indicated by the dotted line around each set of instances). The security group for the web servers only allows access from hosts over TCP on ports 80 and 443 (HTTP and HTTPS) and from instances in the App Servers security group on port 22 (SSH) for direct host management.

The security group for the app servers allows access from the *Web Servers* security group for web requests, and from the corporate subnet over TCP on port 22 (SSH) for direct host management. The user's support engineers could log directly into the application servers from the corporate network, and then access

the other instances from the application server boxes.

The *DB Servers* security group permits only the App Servers security group to access the database servers.

Microsoft Azure

- Cloud-computing platform
 - Platform-as-a-Service running custom applications on pre-configured virtual machines
- Benefits
 - Virtualization
 - Scalability
 - Utility Computing
- Azure includes compute, storage, and application services
- It enables customers to store, process and analyze all types of data - from structured, semi-structured, to unstructured. SQL Data Services (SDS), the first service available within SQL Services, is the relational database.

54

Azure Services Platform

The Azure Services Platform is an internet-scale service-based application platform hosted in Microsoft data centers. Azure includes compute, storage, and application services, all available as elastic resources, and all accessible via standard communication protocols and models. Azure works with the tools, skills, data, and applications customers already have, including .NET, Windows, SQL Server, and more. With Azure, customers will build new connected applications or extend existing ones, quickly and cost-effectively. When launched, Azure will offer guaranteed reliable hosted infrastructure under a pay-as-you-go plan. Suitable for use in a wide variety of application types, and by all types and sizes of organizations and companies, Azure will be able to cut capital and operating expenses and reduce risk.

Components of the Azure Services Platform include:

Windows Azure: Our cloud services operating system that provides flexible and scalable service hosting, data storage, and automated service management.

Windows Azure is Microsoft's platform for building and deploying cloud

based applications, which provides customers with a service hosting and management environment for the Azure Services Platform.

Microsoft .NET Services: A set of higher level developer services like Access Control, Workflow, and Messaging that empower developers to build richer, better connected applications with less code.

.NET Services make developing loosely-coupled on-premises and cloud-based applications easier. .NET Services include a hosted service bus for connecting applications and services across network boundaries, access control for securing applications, and message orchestration. These hosted services allow customers to easily create federated applications that span from on-premises to the cloud.

Microsoft SQL Services delivers on the vision of extending the SQL Server Data Platform capabilities to cloud web-based services. It enables customers to store, process and analyze all types of data - from structured, semi-structured, to unstructured. SQL Data Services (SDS), the first service available within SQL Services, is the relational database. In the future, SQL Services will deliver a comprehensive set of integrated services such as search, reporting, analytics, data integration and data synchronization in the cloud.

Microsoft SQL Services: A set of cloud-based SQL Server capabilities. The first of these capabilities is SQL Data Services, which offers an Internet-facing database service with query capabilities and support for standard protocols.

Microsoft SharePoint Services: Applications available as services today that offer the power of choice to businesses of all sizes and are supported by a set of capabilities unique to the world of hybrid scenarios, such as the ability to federate user identities across the server-service continuum, so a user can authenticate to a portfolio of applications, some on-premises and some consumed as a service.

Live Services provides an easy on ramp for developers to connect with over 460 Million Windows Live users and build web apps with rich social sharing embedded experiences. The Live Framework provides the one uniform way to consume and program Live Services and over time will provide the ability to connect the power and scale of web apps to rich client experiences across a world of digital devices.

Live Services: A set of foundational services and APIs that allows third parties to tap into the massive Windows Live consumer network and its core infrastructure to build rich, engaging applications.

On top of the Azure Services Platform, we offer finished consumer services such as Windows Live, MSN, and Office Live, as well as enterprise-class applications such as Microsoft Exchange Online, Sharepoint Online, and CRM Online. These

solutions provide the power of choice for businesses of all sizes and vertical segments and present a set of exciting new opportunities for Microsoft

Azure Overview

Runs on Microsoft data centers

- Commodity hardware

Wide range of app dev technology:

- .NET Framework, Unmanaged code, others..
- C#, VB, C++, Java, ASP.NET, WCF, PHP

Several Storage Options

- BLOBs and simple data structures
 - RESTful approach to Windows Azure storage
- Traditional Relational, SQL Azure Database

Connectivity with other distributed applications

55

-Windows Azure is designed to run on cheap commodity hardware running in Microsoft-managed data centers.

- There is a growing list of application development technology that it supports.

 - From Microsoft centric technologies like .NET and C# to Java and other scripting languages.

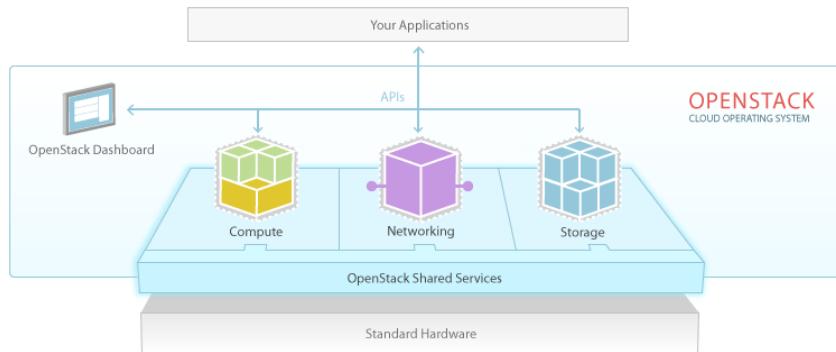
-It also has several storage options so you can choose what's most appropriate for your application

 - For example, for large images and videos, there's support for storing BLOBs on the cloud

 - If you need more traditional database functionality, there's SQL Azure Database

- The Azure Platform also provides mechanisms for connecting distributed applications on and off the cloud.

OpenStack



OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

Designing Cloud-Native Applications

- Designing cloud applications means taking vision and awareness of all the aspects involved, of the opportunities they enable and the issues they raise;
- Consider alternative solutions, carry out choices and approach decisions in a conscious manner in order to achieve what is required by the business in the best possible way, also considering all those maintenance and evolution activities that the high complexity of these areas requires.
- Rather than being conceived as monoliths, applications are **decomposed into decentralized and smaller size**. These services communicate through **APIs** or using **event management** or **asynchronous messaging**. Applications scale out by adding new instances as needed.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>

Designing Cloud-Native Applications

- The cloud solutions differ in terms of the calculation service provided (**IaaS**, **PaaS**, **SaaS**, **FaaS**) and in data storage system used (**relational databases**, **key/value archives**, **document databases**, **graph databases**, **columnar databases**, **search engine databases**, **time series databases**, **files shared**). For the latter there are a number of criteria to be taken into consideration in order to take the right one decision.

Designing Cloud-Native Applications

In order for a cloud application to be scalable, resilient and manageable, a series of principles must be followed:

- Design for **self-healing**. In a distributed system they can errors occur. Design the application to be able to correct automatically correct your mistakes.
- Make everything **redundant**. Apply redundancy in the for application avoid single points of failure.
- **Minimize coordination** between application services to achieve scalability.
- Design for **horizontal scaling**. Design the application to scale horizontally, adding or removing new instances as needed.

Designing Cloud-Native Applications

- Use **managed services**. Whenever possible, use Platform as a Service (PaaS) instead infrastructure as a service (IaaS).
- Use the **best data archive** for the process. Choosing the suitable storage technology for data and the methods of envisaged use.
- Designing from an **evolutionary** point of view. All efficient applications change over time. A Evolving design is critical to continuous innovation.
- Every **design decision** must be justified by a business requirement.

Managed cloud services are **services that offer partial or complete management of a client's cloud resources or infrastructure**. Management responsibilities can include migration, configuration, optimization, security, and maintenance.

Every design decision must be justified by a business requirement. This design principle might seem obvious, but is crucial to keep in mind when designing applications. Must your application support millions of users, or a few thousand? Are there large traffic bursts, or a steady workload? What level of application outage is acceptable? Ultimately, business requirements drive these design considerations.



Cloud Design Patterns

Design patterns are useful for building **reliable, scalable, secure**, applications in the cloud.

Challenges in cloud development

- Data management

Data management is the key element of cloud applications, and it influences most of the quality attributes. **Data is typically hosted in different locations and across multiple servers for performance, scalability or availability.** This can present various challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations.

Additionally, **data should be protected at rest, in transit, and via authorized access mechanisms** to maintain security assurances of confidentiality, integrity, and availability.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>



Cloud Design Patterns

Data Management Patterns

Pattern	Summary
Cache-Aside	Load data on demand into a cache from a data store
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Sharding	Divide a data store into a set of horizontal partitions or shards .
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>



Cache-Aside pattern

Load data **on demand** into a cache from a data store. This can improve performance and also helps to maintain consistency between data held in the cache and data in the underlying data store.

Context and problem

Applications use a cache to improve repeated access to information held in a data store. However, it's impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is as up-to-date as possible, but can also detect and handle situations that arise when the data in the cache has become stale.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>



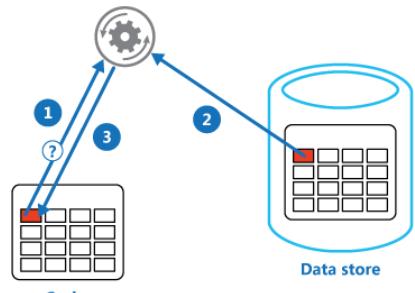
Cache-Aside pattern

Solution

Many commercial caching systems provide **read-through and write-through/write-behind** operations. In these systems, an application retrieves data by referencing the cache. If the data isn't in the cache, it's retrieved from the data store and added to the cache.

For caches that don't provide this functionality, it's the responsibility of the applications that use the cache to maintain the data. An application can emulate the functionality of read-through caching by implementing the **cache-aside strategy**. This strategy loads data into the cache on demand.

The figure illustrates using the Cache-Aside pattern to store data in the cache.



- 1: Determine whether the item is currently held in the cache.
- 2: If the item is not currently in the cache, read the item from the data store.
- 3: Store a copy of the item in the cache.

•Cache-aside: This is where the application is responsible for reading and writing from the database and the cache doesn't interact with the database at all. The cache is "kept aside" as a faster and more scalable in-memory data store. The application checks the cache before reading anything from the database. And, the application updates the cache after making any updates to the database. This way, the application ensures that the cache is kept synchronized with the database.

•Read-through/Write-through (RT/WT): This is where the application treats cache as the main data store and reads data from it and writes data to it. The cache is responsible for reading and writing this data to the database, thereby relieving the application of this responsibility.

If an application updates information, it can follow the write-through strategy by making the modification to the data store, and by invalidating the corresponding item in the cache.

When the item is next required, using the cache-aside strategy will cause the updated data to be retrieved from the data store and added back into the cache.

e.g. Documentation of Oracle Coherence:

https://docs.oracle.com/cd/E15357_01/coh.360/e15723/cache_rtwtwbra.htm#CO

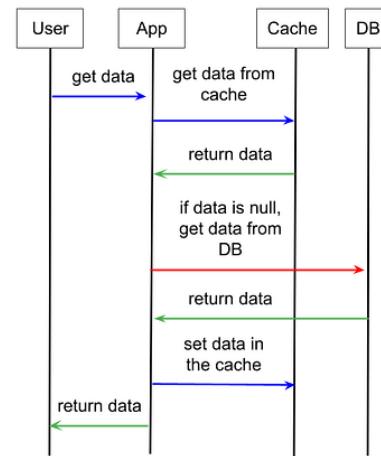
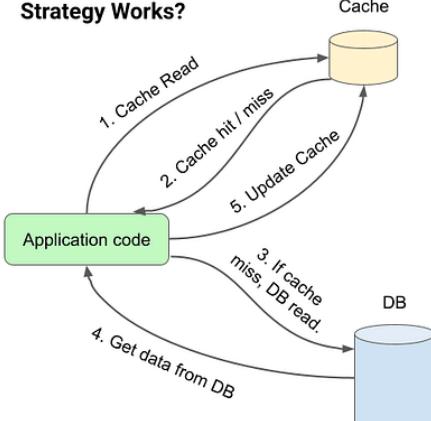
When an application asks the cache for an entry, for example the key X, and X is not already in the cache, Coherence will automatically delegate to the CacheStore and ask it to load X from the underlying data source. If X exists in the data source, the CacheStore will load it, return it to Coherence, then Coherence will place it in the cache for future use and finally will return X to the application code that requested it. This is called **Read-Through caching**.

Write-Through: In this case, when the application updates a piece of data in the cache (that is, calls put(...) to change a cache entry,) the operation will not complete (that is, the put will not return) until Coherence has gone through the CacheStore and successfully stored the data to the underlying data source. This does not improve write performance at all, since you are still dealing with the latency of the write to the data source. Improving the write performance is the purpose for the Write-Behind Cache functionality.

In the **Write-Behind** scenario, modified cache entries are asynchronously written to the data source after a configured delay, whether after 10 seconds, 20 minutes, a day, a week or even longer. Note that this only applies to cache inserts and updates - cache entries are removed synchronously from the data source. For Write-Behind caching, Coherence maintains a write-behind queue of the data that must be updated in the data source. When the application updates X in the cache, X is added to the write-behind queue (if it isn't there already; otherwise, it is replaced), and after the specified write-behind delay Coherence will call the CacheStore to update the underlying data source with the latest state of X. Note that the write-behind delay is relative to the first of a series of modifications—in other words, the data in the data source will never lag behind the cache by more than the write-behind delay.

Cache-Aside pattern

How Cache-aside Strategy Works?





Cache-Aside pattern

Use this pattern when:

- A cache doesn't provide native read-through and write-through operations.
- Resource demand is unpredictable. This pattern enables applications to load data on demand. It makes no assumptions about which data an application will require in advance.

This pattern might not be suitable:

- When the cached data set is static. If the data will fit into the available cache space, prime the cache with the data on startup and apply a policy that prevents the data from expiring.
- For caching session state information in a web application hosted in a web farm. In this environment, you should avoid introducing dependencies based on client-server affinity.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>

Caching session state information in a web application hosted in a web farm can lead to several issues and challenges, making it generally unsuitable for this purpose. Here's why:

Session data inconsistency: In a web farm environment, where multiple servers are serving requests, caching session state information can lead to inconsistencies. Each server may cache session data independently, so when a subsequent request is directed to a different server, it may not have access to the cached session data, leading to inconsistencies or errors in the application.

Scalability challenges: Caching session state information can create scalability challenges, especially as the number of servers in the web farm increases. As more servers are added, managing and synchronizing the cached session data across all servers becomes increasingly complex and resource-intensive.

Increased memory usage: Caching session data requires additional memory resources on each server. In a web farm environment with multiple servers, this can quickly lead to increased memory usage across the entire infrastructure, potentially impacting the performance and stability of the application.

Potential data security risks: Storing session data in a cache introduces potential security risks, especially if the cached data is not properly secured or encrypted. Unauthorized access to the cache could result in sensitive session information being compromised.

Difficulty in cache invalidation: Managing the invalidation of cached session data can be challenging, especially when users log out or session data expires. Ensuring that all cached session data is invalidated across all servers in the web farm requires careful coordination and can be prone to errors.

Instead of caching session state information in a web farm environment, it's typically better to use other methods for managing session state, such as storing session data in a centralized database or using stateless session management techniques like JSON Web Tokens (JWT). These approaches help ensure consistency, scalability, and security in distributed web application environments.



CQRS

CQRS stands for **Command and Query Responsibility Segregation**, a pattern that separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security.

Context and problem

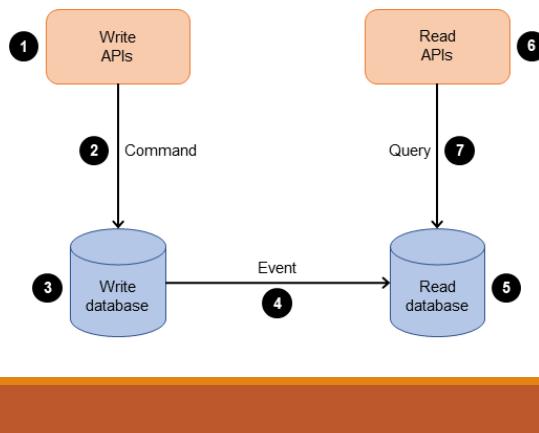
In traditional architectures, the same data model is used to update and query a database. That's simple and works well for basic CRUD (create, read, update, and delete) operations. In more complex applications, however, this approach can become unwieldy.

Read and write workloads are often **asymmetrical**, with very different **performance and scale requirements**.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>

CQRS

The CQRS pattern separates the data mutation, or the command part of a system, from the query part. You can use the CQRS pattern to separate updates and queries if they have different requirements for throughput, latency, or consistency. The CQRS pattern splits the application into two parts—the **command side** and the **query side**—as shown in the following diagram. The command side handles create, update, and delete requests. The query side runs the query part by using the read replicas.



<https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/cqrs-pattern.html>

The diagram shows the following process:

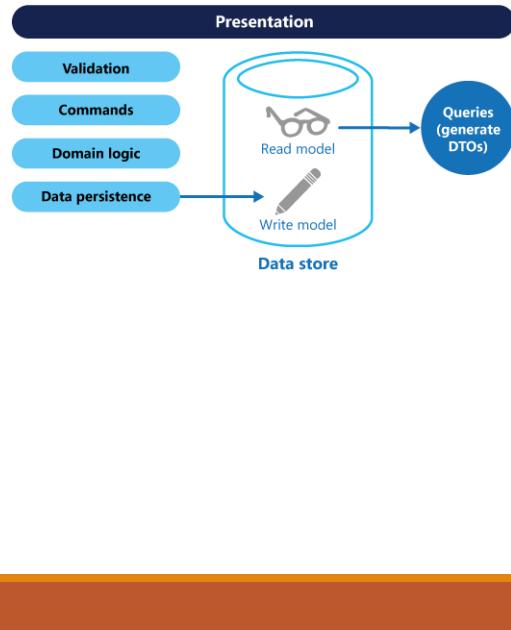
1. The business interacts with the application by sending commands through an API. Commands are actions such as creating, updating or deleting data.
2. The application processes the incoming command on the command side. This involves validating, authorizing, and running the operation.
3. The application persists the command's data in the write (command) database.
4. After the command is stored in the write database, events are triggered to update the data in the read (query) database.
5. The read (query) database processes and persists the data. Read databases are designed to be optimized for specific query requirements.
6. The business interacts with read APIs to send queries to the query side of the application.
7. The application processes the incoming query on the query side and retrieves the data from the read database.



CQRS

CQRS separates reads and writes into different models, using **commands to update data**, and **queries to read data**.

- **Commands should be task-based**, rather than data centric. ("Book hotel room", not "set ReservationStatus to Reserved").
- **Queries never modify the database**. A query returns a DTO (data transfer object) that does not encapsulate any domain knowledge. A DTO can aggregate the data that would have been transferred by the several calls, but that is served by one call only.



Having separate query and update models simplifies the design and implementation. However, one disadvantage is that CQRS code can't automatically be generated from a database schema using scaffolding mechanisms such as O/RM tools (However, you will be able to build your customization on top of the generated code).

A command is an instruction, a directive to perform a specific task. It is an intention to change something. A command (procedure) does something but does not return a result.

A command should convey the intent of the user. Handling a command should result in one transaction on one aggregate; basically, each command should clearly state one well-defined change.

The commands make up [the write model](#), and the write model should be as close as possible to the business processes.

A query is a request for information. A query (function or attribute) returns a result but does not change the state. It is an intention to get data, or the status of data, from a specific place. Nothing in the data should be changed by the request. As queries do not change anything, they don't need to involve the Domain Model.

The queries make up the read model. The read model should be derived from the write model. It also doesn't have to be permanent: new read models can be

introduced into the system without impacting the existing ones. Read models can be deleted and recreated without losing the business logic or information, as this is stored in the write model.

A Data Transfer Object (DTO) is a design pattern used in software development, particularly in the context of distributed systems or layered architectures like client-server applications. A DTO is an object that carries data between processes. The motivation for its use is that communication between processes is usually done resorting to remote interfaces (e.g., web services), where each call is an expensive operation. Because the majority of the cost of each call is related to the round-trip time between the client and the server, one way of reducing the number of calls is to use an object (the DTO) that aggregates the data that would have been transferred by the several calls, but that is served by one call only.

A Data Transfer Object (DTO) is a design pattern used in software development, particularly in the context of distributed systems or layered architectures like client-server applications. The main purpose of a DTO is to transfer data between software application subsystems or across network boundaries.

The key characteristics and considerations regarding DTOs are:

Data Structure: A DTO is essentially a plain old data structure (PODS) that typically contains fields to represent data. It may include attributes, properties, or fields corresponding to the data being transferred.

Lightweight: DTOs are often lightweight, containing only data and no business logic. They are meant to be simple containers for data transfer, devoid of any behavior or functionality.

Transfer Between Layers: DTOs are commonly used to transfer data between different layers of an application, such as between the presentation layer (UI) and the business logic layer, or between the business logic layer and the data access layer.

Serialization: DTOs are often designed to be serializable, meaning they can be converted into a format suitable for transmission over a network, such as JSON or XML. This allows them to be easily passed between distributed components or across different platforms.

Reducing Network Calls: In distributed systems, DTOs can help reduce the number of network calls by aggregating data into a single object for transmission, rather than making multiple calls for individual pieces of data.

Customization for Specific Use Cases: DTOs can be customized to meet specific use cases or requirements. Different DTOs may be created for different scenarios,

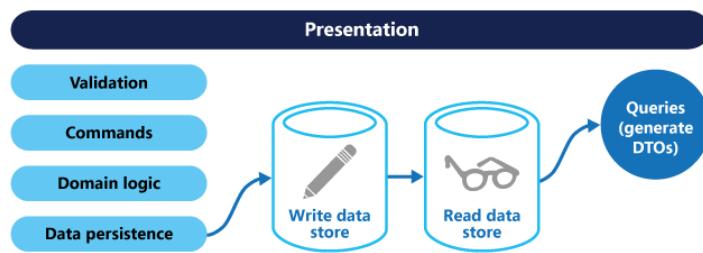
containing only the necessary data fields for each particular use case.

Avoiding Tight Coupling: DTOs help avoid tight coupling between different parts of an application by providing a standardized data format for communication. This allows subsystems to evolve independently without impacting each other's interfaces.

Overall, Data Transfer Objects provide a structured and efficient means of transferring data between different parts of a software system, facilitating communication and interoperability in distributed architectures.

CQRS

For greater isolation, you can physically separate the read data from the write data. In that case, the read database can use its own data schema that is optimized for queries. For example, it can store a [materialized view](#) of the data, in order to avoid complex joins or complex O/RM mappings. It might even use a different type of data store. For example, the write database might be relational, while the read database is a document database.



If separate read and write databases are used, they must be kept in sync. Typically this is accomplished by having the write model publish an event whenever it updates the database. For more information about using events, see [Event-driven architecture style](#). Since message brokers and databases usually cannot be enlisted into a single distributed transaction, there can be challenges in guaranteeing consistency when updating the database and publishing events. For more information, see the [guidance on idempotent message processing](#).

Object-relational mapping (ORM, O/RM, and O/R mapping tool) is a programming technique for converting data between a relational database and the heap of an object-oriented programming language. This creates, in effect, a virtual object database that can be used from within the programming language.

CQRS

Consider CQRS for the following scenarios:

- **Collaborative domains** where many users access the same data in parallel.
- Scenarios where **performance of data reads must be fine-tuned** separately from performance of data writes, especially when the number of **reads is much greater than the number of writes**.
- Scenarios where one team of developers can focus on the complex domain model that is part of the **write model**, and another team can focus on the **read model** and the user interfaces.
- Scenarios where the system is expected to evolve over time and might contain multiple versions of the model, or where **business rules change regularly**.
- **Integration with other systems**, especially in combination with event sourcing, where the temporal failure of one subsystem shouldn't affect the availability of the others.

Consider CQRS for the following scenarios:

- Collaborative domains where many users access the same data in parallel. CQRS allows you to define commands with enough granularity to minimize merge conflicts at the domain level, and conflicts that do arise can be merged by the command.
- Task-based user interfaces where users are guided through a complex process as a series of steps or with complex domain models. The write model has a full command-processing stack with business logic, input validation, and business validation. The write model may treat a set of associated objects as a single unit for data changes (an aggregate, in DDD terminology) and ensure that these objects are always in a consistent state. The read model has no business logic or validation stack, and just returns a DTO for use in a view model. The read model is eventually consistent with the write model.
- Scenarios where performance of data reads must be fine-tuned separately from performance of data writes, especially when the number of reads is much greater than the number of writes. In this scenario, you can scale out the read model, but run the write model on just a few instances. A small number of write model instances also helps to minimize the occurrence of merge conflicts.
- Scenarios where one team of developers can focus on the complex domain model that is part of the write model, and another team can focus on the read model and the user interfaces.
- Scenarios where the system is expected to evolve over time and might contain multiple versions of the model, or where business rules change regularly.

- Integration with other systems, especially in combination with event sourcing, where the temporal failure of one subsystem shouldn't affect the availability of the others.



CQRS

- This pattern isn't recommended when:
 - The domain or the business rules are **simple**.
 - A simple **CRUD-style user interface** and data access operations are sufficient.
- Consider applying CQRS to limited sections of your system where it will be most valuable.





Event Sourcing pattern

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with it, as in the traditional create, read, update, and delete (CRUD) model.

The CRUD approach has some limitations:

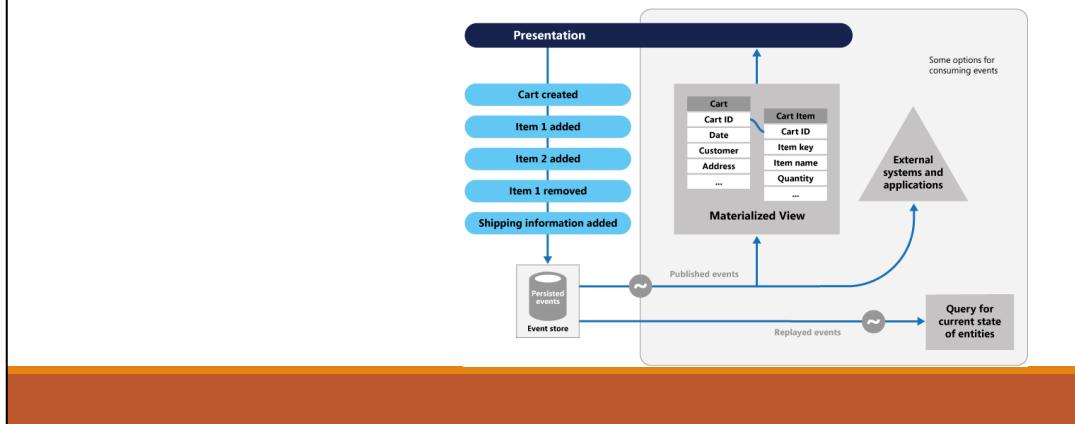
- CRUD systems perform update operations directly against a data store. These operations can slow down performance and responsiveness and can limit scalability, due to the processing overhead it requires.
- In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data.
- Unless there's another auditing mechanism that records the details of each operation in a separate log, history is lost.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>



Event Sourcing pattern

The Event Sourcing pattern defines an approach to handling operations on data that is driven by a sequence of events, each of which is **recorded in an append-only store**. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they are persisted. Each event represents a set of changes to the data (such as `AddedItemToOrder`).



Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

The events are persisted in an event store that acts as the system of record (the authoritative data source) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.

Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems. For example, a system can maintain a materialized view of all customer orders that's used to populate parts of the UI. The application adds new orders, adds or removes items on the order, and adds shipping information. **The events that describe these changes can be handled and used to update the materialized view.**

At any point, it's possible for applications to **read the history of events**. You can then use it to materialize the current state of an entity by playing back and consuming all the events that are related to that entity. This process can occur on

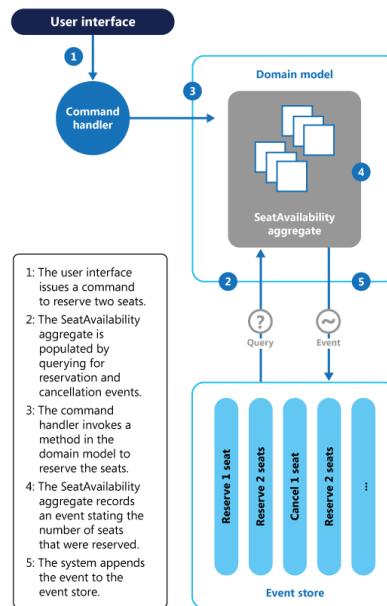
demand to materialize a domain object when handling a request. Or, the process occurs through a scheduled task so that the state of the entity can be stored as a materialized view, to support the presentation layer.



Event Sourcing pattern

Example: Conference management system

- A conference management system needs to track the number of completed bookings for a conference. This way it can check whether there are seats still available, when a potential attendee tries to make a booking.



The system could store the total number of bookings for a conference in at least two ways:

- The system could store the information about the total number of bookings as a separate entity in a database that holds booking information. As bookings are made or canceled, the system could increment or decrement this number as appropriate. This approach is simple in theory, but can cause scalability issues if a large number of attendees are attempting to book seats during a short period of time. For example, in the last day or so prior to the booking period closing.
- The system could store information about bookings and cancellations as events held in an event store. It could then calculate the number of seats available by replaying these events. This approach can be more scalable due to the immutability of events. The system only needs to be able to read data from the event store, or append data to the event store. Event information about bookings and cancellations is never modified.

The sequence of actions for reserving two seats is as follows:

1. The user interface issues a command to reserve seats for two attendees. The command is handled by a separate command handler. A piece of logic that is decoupled from the user interface and is responsible for handling requests posted as commands.
2. An aggregate containing information about all reservations for the conference is

constructed by querying the events that describe bookings and cancellations. This aggregate is called `SeatAvailability`, and is contained within a domain model that exposes methods for querying and modifying the data in the aggregate.

3. Some optimizations to consider are using snapshots (so that you don't need to query and replay the full list of events to obtain the current state of the aggregate), and maintaining a cached copy of the aggregate in memory.
4. The command handler invokes a method exposed by the domain model to make the reservations.

5. The `SeatAvailability` aggregate records an event containing the number of seats that were reserved. The next time the aggregate applies events, all the reservations will be used to compute how many seats remain.

6. The system appends the new event to the list of events in the event store.

If a user cancels a seat, the system follows a similar process except the command handler issues a command that generates a seat cancellation event and appends it to the event store.

In addition to providing more scope for scalability, using an event store also provides a complete history, or audit trail, of the bookings and cancellations for a conference. The events in the event store are the accurate record. There's no need to persist aggregates in any other way because the system can easily replay the events and restore the state to any point in time.



Event Sourcing pattern

Use this pattern in the following scenarios:

- When you want to **capture intent, purpose, or reason** in the data.
- When it is vital to **minimize or completely avoid** the occurrence of **conflicting updates** to data.
- When you want to record events that occur, to replay them to **restore the state of a system, to roll back changes, or to keep a history and audit log**.
- When you use **events**. It is a natural feature of the operation of the application, and it requires little extra development or implementation effort.
- When you need to **decouple** the process of inputting, or updating data from the tasks required to apply these actions.
- When used with CQRS, and **eventual consistency is acceptable while a read model is updated**, or the performance impact of rehydrating entities and data from an event stream is acceptable.



Event Sourcing pattern

This pattern might not be useful in the following situations:

- Small or simple domains, systems that have little or no business logic, or nondomain systems that naturally work well with traditional CRUD data management mechanisms.
- Systems where consistency and real-time updates to the views of the data are required.
- Systems where audit trails, history, and capabilities to roll back and replay actions aren't required.
- Systems where there is only a low occurrence of conflicting updates to the underlying data. For example, systems that predominantly add data rather than updating it.



Index Table pattern

Create indexes over the fields in data stores that are frequently referenced by queries. This pattern can improve query performance by allowing applications to more quickly locate the data to retrieve from a data store.

Context and problem

- Many data stores organize the data for a collection of entities using the primary key. An application can use this key to locate and retrieve data.
- While the primary key is valuable for queries that fetch data based on the value of this key, an application might not be able to use the primary key if it needs to retrieve data based on some other field.
 - For example, if Customer ID is the primary key to retrieve customers, if an application queries data solely by referencing the value of some other attribute, such as the town in which the customer is located the application might have to fetch and examine every customer record, which could be a slow process.
- Many relational database management systems support secondary indexes, but most NoSQL data stores used by cloud applications don't provide an equivalent feature

Many relational database management systems support secondary indexes. A secondary index is a separate data structure that's organized by one or more nonprimary (secondary) key fields, and it indicates where the data for each indexed value is stored. The items in a secondary index are typically sorted by the value of the secondary keys to enable fast lookup of data. These indexes are usually maintained automatically by the database management system.

You can create as many secondary indexes as you need to support the different queries that your application performs. For example, in a Customers table in a relational database where the Customer ID is the primary key, it's beneficial to add a secondary index over the town field if the application frequently looks up customers by the town where they reside.

However, although secondary indexes are common in relational systems, most NoSQL data stores used by cloud applications don't provide an equivalent feature.



Index Table pattern

If the data store does not support secondary indexes, you can emulate them manually by creating your own index tables.

- Three strategies are commonly used for structuring an index table, depending on the number of secondary indexes that are required and the nature of the queries that an application performs.
- The first strategy is to duplicate the data in each index table but organize it by different keys (complete denormalization).

Secondary Key (Town)	Customer Data
Chicago	ID: 5, LastName: Smith, Town: Chicago, ...
Chicago	ID: 9, LastName: Jones, Town: Chicago, ...
Chicago	ID: 1000, LastName: Clarke, Town: Chicago, ...
...	...
Portland	ID: 3, LastName: Robinson, Town: Portland, ...
Portland	ID: 7, LastName: Clarke, Town: Portland, ...
Redmond	ID: 1, LastName: Smith, Town: Redmond, ...
Redmond	ID: 4, LastName: Brown, Town: Redmond, ...
Redmond	ID: 6, LastName: Green, Town: Redmond, ...
Redmond	ID: 8, LastName: Smith, Town: Redmond, ...
Seattle	ID: 2, LastName: Jones, Town: Seattle, ...
...	...

Secondary Key (LastName)	Customer Data
Brown	ID: 4, LastName: Brown, Town: Redmond, ...
Clarke	ID: 7, LastName: Clarke, Town: Portland, ...
Clarke	ID: 1000, LastName: Clarke, Town: Chicago, ...
Green	ID: 6, LastName: Green, Town: Redmond, ...
Jones	ID: 2, LastName: Jones, Town: Seattle, ...
Jones	ID: 9, LastName: Jones, Town: Chicago, ...
...	...
Robinson	ID: 3, LastName: Robinson, Town: Portland, ...
Smith	ID: 1, LastName: Smith, Town: Redmond, ...
Smith	ID: 5, LastName: Smith, Town: Chicago, ...
Smith	ID: 8, LastName: Smith, Town: Redmond, ...
...	...

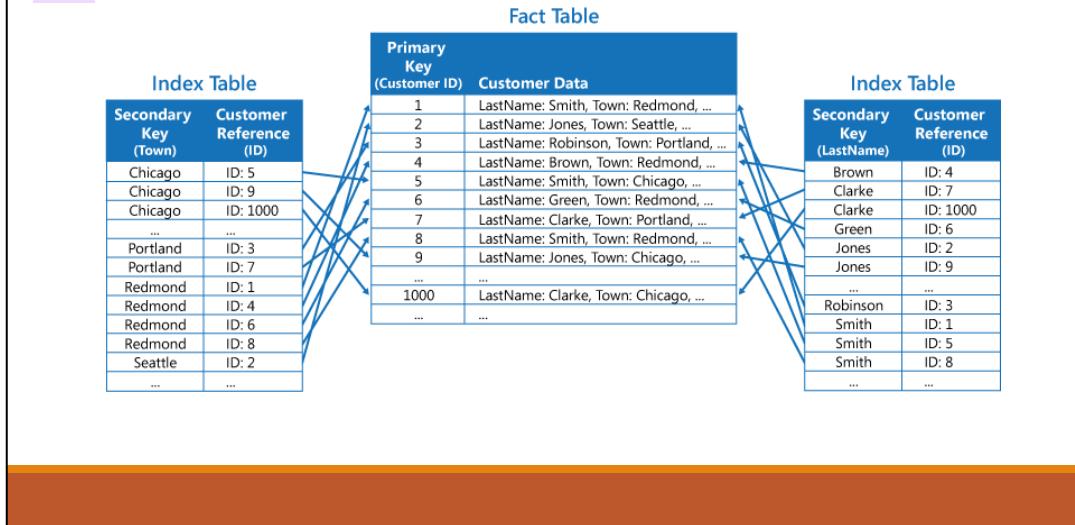
If the data store doesn't support secondary indexes, you can emulate them manually by creating your own index tables. An index table organizes the data by a specified key. Three strategies are commonly used for structuring an index table, depending on the number of secondary indexes that are required and the nature of the queries that an application performs.

The first strategy is appropriate if the data is relatively static compared to the number of times it's queried using each key. If the data is more dynamic, the processing overhead of maintaining each index table becomes too large for this approach to be useful. Also, if the volume of data is very large, the amount of space required to store the duplicate data is significant.



Index Table pattern

The **second strategy** is to **create normalized index tables organized by different keys** and **reference the original data** by using the primary key rather than duplicating it, as shown in the following figure. The original data is called a fact table.

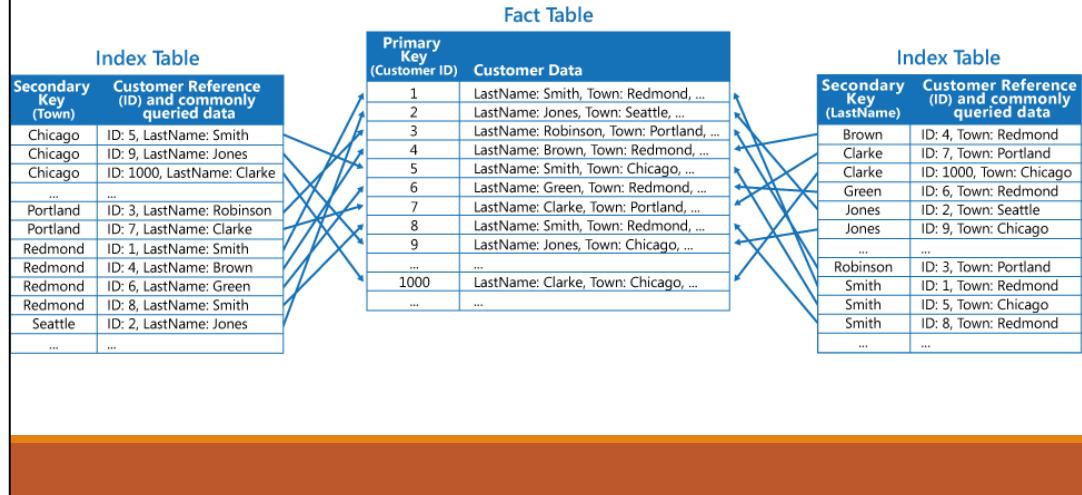


This technique **saves space** and **reduces the overhead** of maintaining duplicate data. The **disadvantage** is that an application has to perform **two lookup** operations to find data using a secondary key. It has to find the primary key for the data in the index table, and then use the primary key to look up the data in the fact table.



Index Table pattern

The third strategy is to create **partially normalized index tables organized by different keys that duplicate frequently retrieved fields**. Reference the fact table to access less frequently accessed fields. The next figure shows how commonly accessed data is duplicated in each index table.



With this strategy, you can strike a balance between the first two approaches. The data for common queries can be retrieved quickly by using a single lookup, while the space and maintenance overhead isn't as significant as duplicating the entire data set.



Index Table pattern

If an application frequently queries data by specifying a combination of values (for example, "Find all customers that live in Redmond and that have a last name of Smith"), you could implement the keys to the items in the index table as a concatenation of the Town attribute and the LastName attribute. The next figure shows an index table based on composite keys. The keys are sorted by Town, and then by LastName for records that have the same value for Town.

Index Table		Fact Table	
Composite Key (Town, LastName)	Customer Reference (ID) and commonly queried data	Primary Key (Customer ID)	Customer Data
Chicago, Clarke	ID: 1000, ...	1	LastName: Smith, Town: Redmond, ...
Chicago, Jones	ID: 9, ...	2	LastName: Jones, Town: Seattle, ...
Chicago, Smith	ID: 5, ...	3	LastName: Robinson, Town: Portland, ...
...	...	4	LastName: Brown, Town: Redmond, ...
Portland, Clarke	ID: 7, ...	5	LastName: Smith, Town: Chicago, ...
Portland, Robinson	ID: 3, ...	6	LastName: Green, Town: Redmond, ...
Redmond, Brown	ID: 4, ...	7	LastName: Clarke, Town: Portland, ...
Redmond, Green	ID: 6, ...	8	LastName: Smith, Town: Redmond, ...
Redmond, Smith	ID: 1, ...	9	LastName: Jones, Town: Chicago, ...
Redmond, Smith	ID: 8,
Seattle, Jones	ID: 2, ...	1000	LastName: Clarke, Town: Chicago, ...
...

Index tables can speed up query operations over sharded data, and are especially useful where the shard key is hashed. T



Index Table pattern

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- The **overhead of maintaining secondary indexes** can be significant. You must analyze and understand the queries that your application uses. Only create index tables when they're likely to be used regularly. Do not create speculative index tables to support queries that an application does not perform, or performs only occasionally.
- **Duplicating data in an index table can add significant overhead** in storage costs and the effort required to maintain multiple copies of data.
- Implementing an index table as a normalized structure that references the original data requires **an application to perform two lookup operations** to find data. The first operation searches the index table to retrieve the primary key, and the second uses the primary key to fetch the data.
- If a system incorporates a number of index tables over very large data sets, it can be **difficult to maintain consistency** between index tables and the original data.



Index Table pattern

When to use this pattern

Use this pattern to **improve query performance** when an application frequently needs to retrieve data by using a **key other than the primary (or shard) key**.

This pattern might not be useful when:

- **Data is volatile.** An **index table** can become out of date very quickly, making it ineffective or making the **overhead of maintaining the index table greater** than any savings made by using it.
- A **field selected as the secondary key** for an index table is **nondiscriminating** and can only have a small set of values (for example, gender).
- The balance of the **data values** for a field selected as the **secondary key** for an index table are **highly skewed**. For example, if 90% of the records contain the same value in a field, then creating and maintaining an index table to look up data based on this field might create more overhead than scanning sequentially through the data. However, if queries very frequently target values that lie in the remaining 10%, this index can be useful. You should understand the queries that your application is performing, and how frequently they're performed.

When a field selected as the secondary key for an index table is described as nondiscriminating, it means that it doesn't effectively differentiate between rows, and therefore, it may not greatly improve query performance when used as a criterion for searching or filtering data. In this case, the application of this pattern could be very effective.



Materialized View pattern

Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations. This can help support efficient querying and data extraction, and improve application performance.

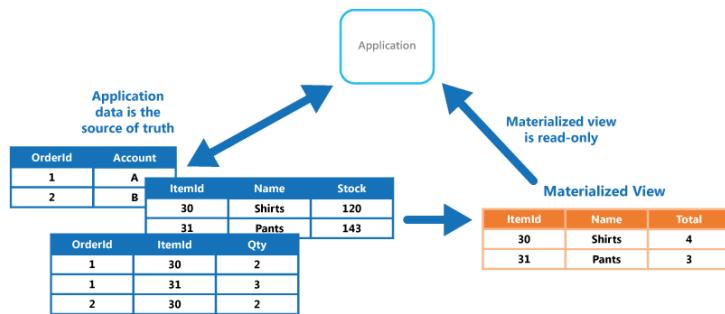
Context and problem

- When storing data, the priority for developers and data administrators is often focused on how the data is stored, as opposed to how it's read. The chosen storage format is usually closely related to the format of the data, requirements for managing data size and data integrity, and the kind of store in use. For example, when using NoSQL document store, the data is often represented as a series of aggregates, each containing all of the information for that entity.
- However, this can have a negative effect on queries. When a query only needs a subset of the data from some entities, such as a summary of orders for several customers without all of the order details, it must extract all of the data for the relevant entities in order to obtain the required information.



Materialized View pattern

- To support efficient querying, a common solution is to generate, in advance, a view that materializes the data in a format suited to the required results set. The Materialized View pattern describes generating prepopulated views of data in environments where the source data isn't in a suitable format for querying, where generating a suitable query is difficult, or where query performance is poor due to the nature of the data or the data store.
- These materialized views, which only contain data required by a query, allow applications to quickly obtain the information they need.



In addition to joining tables or combining data entities, materialized views can include the current values of calculated columns or data items, the results of combining values or executing transformations on the data items, and values specified as part of the query. **A materialized view can even be optimized for just a single query.**

A key point is that a materialized view and the data it contains is completely disposable because it can be entirely rebuilt from the source data stores. **A materialized view is never updated directly by an application, and so it's a specialized cache.**

When the source data for the view changes, the view must be updated to include the new information. You can schedule this to **happen automatically**, or when the system detects a change to the original data. In some cases it might be necessary to regenerate the view manually. The figure shows an example of how the Materialized View pattern might be used.



Materialized View pattern

Issues and considerations

- **How and when** the view will be updated.
- **In some systems**, such as when using the Event Sourcing pattern to maintain a store of only the events that modified the data, **materialized views are necessary**.
- Materialized views tend to be **specifically tailored to one, or a small number of queries**.
- Consider the impact on **data consistency** when generating the view, and when updating the view if this occurs on a schedule.
- Consider **where you will store the view**. A view can be rebuilt if lost. Because of that, if the view is transient and is only used to improve query performance by reflecting the current state of the data, or to improve scalability, it can be stored in a cache or in a less reliable location.
- When defining a materialized view, **maximize its value by adding data items or columns to it based on computation or transformation of existing data items**
- Where the storage mechanism supports it, consider **indexing the materialized view** to further increase performance.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

How and when the view will be updated. Ideally it'll regenerate in response to an event indicating a change to the source data, although this can lead to excessive overhead if the source data changes rapidly. Alternatively, consider using a scheduled task, an external trigger, or a manual action to regenerate the view.

In some systems, such as when using the Event Sourcing pattern to maintain a store of only the events that modified the data, materialized views are necessary. Prepopulating views by examining all events to determine the current state might be the only way to obtain information from the event store. If you're not using Event Sourcing, you need to consider whether a materialized view is helpful or not. Materialized views tend to be specifically tailored to one, or a small number of queries. If many queries are used, materialized views can result in unacceptable storage capacity requirements and storage cost.

Consider the impact on data consistency when generating the view, and when updating the view if this occurs on a schedule. If the source data is changing at the point when the view is generated, the copy of the data in the view won't be fully consistent with the original data.

Consider where you'll store the view. The view doesn't have to be located in the same store or partition as the original data. It can be a subset from a few different partitions combined.

A view can be rebuilt if lost. Because of that, if the view is transient and is only

used to improve query performance by reflecting the current state of the data, or to improve scalability, it can be stored in a cache or in a less reliable location.

When defining a materialized view, maximize its value by adding data items or columns to it based on computation or transformation of existing data items, on values passed in the query, or on combinations of these values when appropriate.

Where the storage mechanism supports it, consider indexing the materialized view to further increase performance. Most relational databases support indexing for views, as do big data solutions based on Apache Hadoop.



Materialized View pattern

This pattern is useful when:

- Creating materialized views over data that's **difficult to query** directly
- Creating temporary views that can dramatically **improve query performance**
- Supporting **occasionally connected or disconnected scenarios** where connection to the data store isn't always available.
- **Simplifying queries** and exposing data for experimentation in a way that doesn't require knowledge of the source data format.
- Providing access to specific subsets of the source data that, for **security or privacy reasons**, shouldn't be generally accessible.
- **Bridging different data stores**, to take advantage of their individual capabilities.
- When using **microservices**, you are recommended to keep them **loosely coupled, including their data storage**. Therefore, materialized views can help you consolidate data from your services.

This pattern is useful when:

- Creating materialized views over data that's difficult to query directly, or where queries must be very complex to extract data that's stored in a normalized, semi-structured, or unstructured way.
- Creating temporary views that can dramatically improve query performance, or can act directly as source views or data transfer objects for the UI, for reporting, or for display.
- Supporting occasionally connected or disconnected scenarios where connection to the data store isn't always available. The view can be cached locally in this case.
- Simplifying queries and exposing data for experimentation in a way that doesn't require knowledge of the source data format. For example, by joining different tables in one or more databases, or one or more domains in NoSQL stores, and then formatting the data to fit its eventual use.
- Providing access to specific subsets of the source data that, for security or privacy reasons, shouldn't be generally accessible, open to modification, or fully exposed to users.
- Bridging different data stores, to take advantage of their individual capabilities. For example, using a cloud store that's efficient for writing as the reference data store, and a relational database that offers good query and read performance to hold the materialized views.
- When using microservices, you are recommended to keep them loosely coupled,

including their data storage. Therefore, materialized views can help you consolidate data from your services. If materialized views are not appropriate in your microservices architecture or specific scenario, please consider having well-defined boundaries that align to [domain driven design \(DDD\)](#) and aggregate their data when requested.



Materialized View pattern

This pattern isn't useful in the following situations:

- The source data is simple and easy to query.
- The source data changes very quickly, or can be accessed without using a view. In these cases, you should avoid the processing overhead of creating views.
- Consistency is a high priority. The views might not always be fully consistent with the original data.



Sharding pattern

Divide a data store into a set of horizontal partitions or shards. This can improve scalability when storing and accessing large volumes of data.

Context and problem

- **Storage space.** A data store for a large-scale cloud application is expected to contain a huge volume of data that could increase significantly over time.
- **Computing resources.** A cloud application is required to support a large number of concurrent users, each of which run queries that retrieve information from the data store.
- **Network bandwidth.** It's possible that the volume of network traffic might exceed the capacity of the network used to connect to the server, resulting in failed requests.
- **Geography.** It might be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons, or to reduce latency of data access.

A data store hosted by a single server might be subject to the following limitations:

- **Storage space.** A data store for a large-scale cloud application is expected to contain a huge volume of data that could increase significantly over time. A server typically provides only a finite amount of disk storage, but you can replace existing disks with larger ones, or add further disks to a machine as data volumes grow. However, the system will eventually reach a limit where it isn't possible to easily increase the storage capacity on a given server.
- **Computing resources.** A cloud application is required to support a large number of concurrent users, each of which run queries that retrieve information from the data store. A single server hosting the data store might not be able to provide the necessary computing power to support this load, resulting in extended response times for users and frequent failures as applications attempting to store and retrieve data time out. It might be possible to add memory or upgrade processors, but the system will reach a limit when it isn't possible to increase the compute resources any further.
- **Network bandwidth.** Ultimately, the performance of a data store running on a single server is governed by the rate the server can receive requests and send replies. It's possible that the volume of network traffic might exceed the capacity of the network used to connect to the server, resulting in failed requests.
- **Geography.** It might be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons, or to reduce latency of data access. If the users are dispersed across different

countries or regions, it might not be possible to store the entire data for the application in a single data store.

Scaling vertically by adding more disk capacity, processing power, memory, and network connections can postpone the effects of some of these limitations, but it's likely to only be a temporary solution. A commercial cloud application capable of supporting large numbers of users and high volumes of data must be able to scale almost indefinitely, so vertical scaling isn't necessarily the best solution.



Sharding pattern

Each shard has the **same schema**, but holds its own **distinct subset of the data**. A shard is a data store in its own right (it can contain the data for many entities of different types), running on a server acting as a storage node.

This pattern has the following benefits:

- You can **scale the system out** by adding further shards running on **additional storage nodes**.
- A system can use **off-the-shelf hardware** rather than specialized and expensive computers for each storage node.
- You can **reduce contention** and **improve performance** by balancing the workload across shards.
- In the cloud, **shards can be located physically close to the users** that'll access the data.

When dividing a data store up into shards, decide which data should be placed in each shard. A shard typically contains items that fall within a specified range determined by one or more attributes of the data. These attributes form the shard key (sometimes referred to as the partition key). The shard key should be static. It shouldn't be based on data that might change.

Sharding physically organizes the data. When an application stores and retrieves data, the sharding logic directs the application to the appropriate shard. This sharding logic can be implemented as part of the data access code in the application, or it could be implemented by the data storage system if it transparently supports sharding.

Abstracting the physical location of the data in the sharding logic provides a high level of control over which shards contain which data. It also enables data to migrate between shards without reworking the business logic of an application if the data in the shards need to be redistributed later (for example, if the shards become unbalanced). The tradeoff is the additional data access overhead required in determining the location of each data item as it's retrieved.

To ensure optimal performance and scalability, it's important to split the data in a way that's appropriate for the types of queries that the application performs. In many cases, it's unlikely that the sharding scheme will exactly match the requirements of every query. For example, in a multi-tenant system an application might need to retrieve tenant data using the tenant ID, but it might also need to look up this data based on some other attribute such as the tenant's name or location. To handle these situations, implement a sharding strategy with a shard

key that supports the most commonly performed queries.

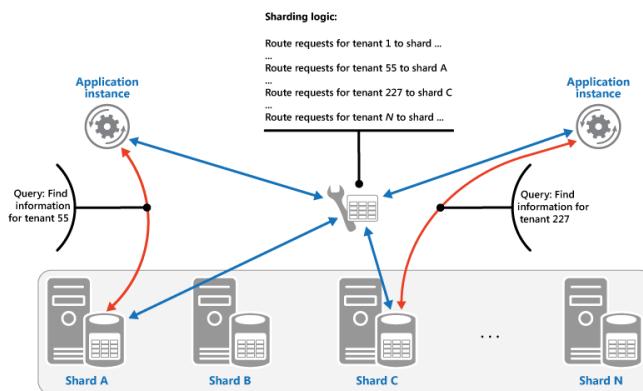


Sharding pattern

Sharding strategies

Three strategies are commonly used when selecting the shard key and deciding how to distribute data across shards.

The Lookup strategy. In this strategy the **sharding logic** implements a **map** that routes a **request** for data to the shard that contains that data using the **shard key**.



In a multi-tenant application all the data for a tenant might be stored together in a shard using the tenant ID as the shard key. Multiple tenants might share the same shard, but the data for a single tenant won't be spread across multiple shards.

Note that there doesn't have to be a one-to-one correspondence between shards and the servers that host them—a single server can host multiple shards.

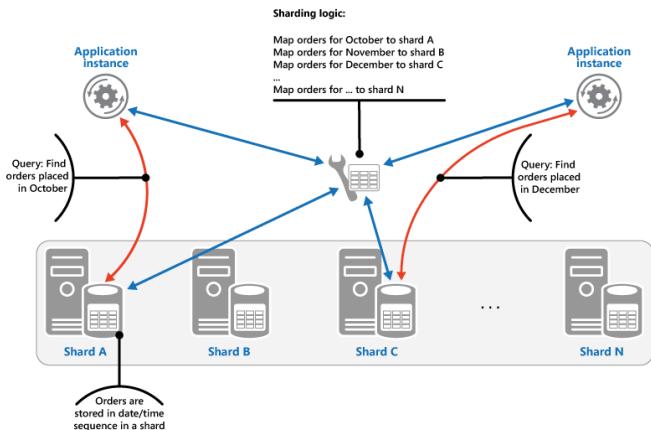
In this strategy the sharding logic implements a map that routes a request for data to the shard that contains that data using the shard key. In a multi-tenant application all the data for a tenant might be stored together in a shard using the tenant ID as the shard key. Multiple tenants might share the same shard, but the data for a single tenant won't be spread across multiple shards. The figure illustrates sharding tenant data based on tenant IDs.

The mapping between the shard key and the physical storage can be based on physical shards where each shard key maps to a physical partition. Alternatively, a more flexible technique for rebalancing shards is virtual partitioning, where shard keys map to the same number of virtual shards, which in turn map to fewer physical partitions. In this approach, an application locates data using a shard key that refers to a virtual shard, and the system transparently maps virtual shards to physical partitions. The mapping between a virtual shard and a physical partition can change without requiring the application code to be modified to use a different set of shard keys.



Sharding pattern

The Range strategy. This strategy **groups related items together in the same shard**, and orders them by shard key—**the shard keys are sequential**. It's useful for applications that frequently retrieve sets of items using range queries (queries that return a set of data items for a shard key that falls within a given range).



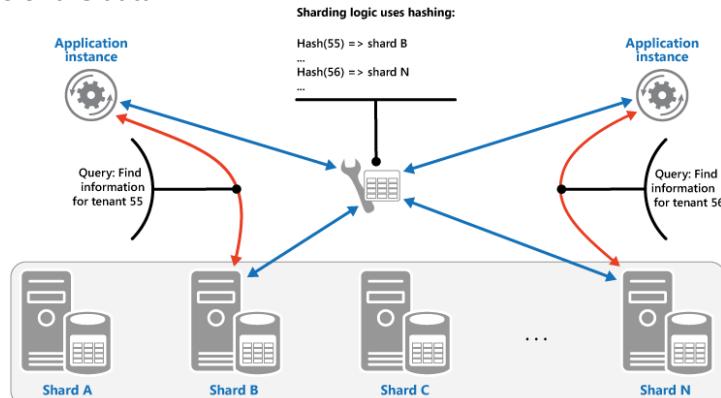
For example, if an application regularly needs to find all orders placed in a given month, this data can be retrieved more quickly if all orders for a month are stored in date and time order in the same shard. If each order was stored in a different shard, they'd have to be fetched individually by performing a large number of point queries (queries that return a single data item). The next figure illustrates storing sequential sets (ranges) of data in shard.

In this example, the shard key is a composite key containing the order month as the most significant element, followed by the order day and the time. The data for orders is naturally sorted when new orders are created and added to a shard. Some data stores support two-part shard keys containing a partition key element that identifies the shard and a row key that uniquely identifies an item in the shard. Data is usually held in row key order in the shard. Items that are subject to range queries and need to be grouped together can use a shard key that has the same value for the partition key but a unique value for the row key.



Sharding pattern

The Hash strategy. The purpose of this strategy is to reduce the chance of hotspots (shards that receive a disproportionate amount of load). It distributes the data across the shards in a way that achieves a balance between the size of each shard and the average load that each shard will encounter. The sharding logic computes the shard to store an item in based on a hash of one or more attributes of the data.



The chosen hashing function should distribute data evenly across the shards, possibly by introducing some random element into the computation. The next figure illustrates sharding tenant data based on a hash of tenant IDs.

To understand the advantage of the Hash strategy over other sharding strategies, consider how a multi-tenant application that enrolls new tenants sequentially might assign the tenants to shards in the data store. When using the Range strategy, the data for tenants 1 to n will all be stored in shard A, the data for tenants n+1 to m will all be stored in shard B, and so on. If the most recently registered tenants are also the most active, most data activity will occur in a small number of shards, which could cause hotspots. In contrast, the Hash strategy allocates tenants to shards based on a hash of their tenant ID. This means that sequential tenants are most likely to be allocated to different shards, which will distribute the load across them. The previous figure shows this for tenants 55 and 56.



Sharding pattern

The three sharding strategies have the following advantages and considerations:

- **Lookup.** This offers **more control over the way that shards are configured** and used. Using **virtual shards** reduces the impact when rebalancing data because **new physical partitions can be added** to even out the workload. The mapping between a virtual shard and the physical partitions that implement the shard can be modified without affecting application code that uses a shard key to store and retrieve data. Looking up shard locations can impose an additional overhead.
- **Range.** This is **easy to implement** and **works well with range queries** because they can often fetch multiple data items from a single shard in a single operation. This strategy offers **easier data management**. However, this strategy **doesn't provide optimal balancing between shards**. **Rebalancing shards is difficult** and might not resolve the problem of uneven load if the majority of activity is for adjacent shard keys.
- **Hash.** This strategy offers a **better chance of more even data and load distribution**. Request routing can be accomplished directly by using the hash function. There's no need to maintain a map. Note that computing the hash might impose an additional overhead. Also, **rebalancing shards is difficult**.

Most common sharding systems implement one of the approaches described above, but you should also consider the business requirements of your applications and their patterns of data usage. For example, in a multi-tenant application:

You can shard data based on workload. You could segregate the data for highly volatile tenants in separate shards. The speed of data access for other tenants might be improved as a result.

You can shard data based on the location of tenants. You can take the data for tenants in a specific geographic region offline for backup and maintenance during off-peak hours in that region, while the data for tenants in other regions remains online and accessible during their business hours.

High-value tenants could be assigned their own private, high performing, lightly loaded shards, whereas lower-value tenants might be expected to share more densely-packed, busy shards.

The data for tenants that need a high degree of data isolation and privacy can be stored on a completely separate server.



Static Content Hosting pattern

Deploy static content to a cloud-based storage service that can deliver them directly to the client. This can reduce the need for potentially expensive compute instances.

Context and problem

- Web applications typically include some elements of static content. This static content might include HTML pages and other resources such as images and documents that are available to the client, either as part of an HTML page (such as inline images, style sheets, and client-side JavaScript files) or as separate downloads (such as PDF documents).
- Although web servers are optimized for dynamic rendering and output caching, they still have to handle requests to download static content. This consumes processing cycles that could often be put to better use.



Static Content Hosting pattern

Issues and considerations

- The hosted storage service **must expose an HTTP endpoint** that users can access to download the static resources. Some storage services also **support HTTPS**, so it's possible to host resources in storage services that require SSL.
- Consider using a **content delivery network (CDN)** to **cache** the contents of the storage container in multiple datacenters. However, you'll likely have to pay for it.
- Storage accounts are often **geo-replicated by default to provide resiliency**. This means that the IP address might change, but the URL will remain the same.
- When some content is located in a storage account and other content is in a hosted compute instance, it becomes **more challenging to deploy and update the application**.
- The storage containers must be configured for **public read access**, but it's vital to ensure that they are not configured for public write access to prevent users being able to upload content.
- Consider **using a valet key or token to control access to resources** that shouldn't be available anonymously.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- The hosted storage service must expose an HTTP endpoint that users can access to download the static resources. Some storage services also support HTTPS, so it's possible to host resources in storage services that require SSL.
- For maximum performance and availability, consider using a content delivery network (CDN) to cache the contents of the storage container in multiple datacenters around the world. However, you'll likely have to pay for using the CDN.
- Storage accounts are often geo-replicated by default to provide resiliency against events that might affect a datacenter. This means that the IP address might change, but the URL will remain the same.
- When some content is located in a storage account and other content is in a hosted compute instance, it becomes more challenging to deploy and update the application. You might have to perform separate deployments, and version the application and content to manage it more easily—especially when the static content includes script files or UI components. However, if only static resources have to be updated, they can simply be uploaded to the storage account without needing to redeploy the application package.
- Storage services might not support the use of custom domain names. In this case it's necessary to specify the full URL of the resources in links because they'll be in a different domain from the dynamically-generated content containing the

links.

- The storage containers must be configured for public read access, but it's vital to ensure that they aren't configured for public write access to prevent users being able to upload content.
- Consider using a valet key or token to control access to resources that shouldn't be available anonymously. See the [Valet Key pattern](#) for more information.



Static Content Hosting pattern

This pattern is useful for:

- Minimizing the hosting cost for websites and applications that contain some **static resources**.
- Exposing static resources and content for **applications running in other hosting environments or on-premises servers**.
- Locating content in more than one geographical area using a **content delivery network** that caches the contents of the storage account in multiple datacenters around the world.
- **Monitoring costs and bandwidth usage.** Using a separate storage account for some or all of the static content allows the costs to be more easily separated from hosting and runtime costs.



Static Content Hosting pattern

This pattern might not be useful in the following situations:

- The application needs to **perform some processing** on the static content before delivering it to the client. For example, it might be necessary to add a timestamp to a document.
- The **volume of static content is very small**. The overhead of retrieving this content from separate storage can outweigh the cost benefit of separating it out from the compute resource.



Valet Key pattern

Use a token that provides clients with restricted direct access to a specific resource, in order to offload data transfer from the application. This is particularly useful in applications that use cloud-hosted storage systems or queues, and can minimize cost and maximize scalability and performance.

Context and problem

- Client programs and web browsers often need to read and write files or data streams to and from a storage. The application could handle data transfer. However, this approach absorbs valuable resources such as compute, memory, and bandwidth.
- Data stores could be given the ability to handle upload and download of data directly, without requiring that the application perform any processing to move this data. But, this typically requires the client to have access to the security credentials for the store. This can be a useful technique to minimize data transfer costs and the requirement to scale out the application, and to maximize performance. Nevertheless, **the application is no longer able to manage the security of the data**.

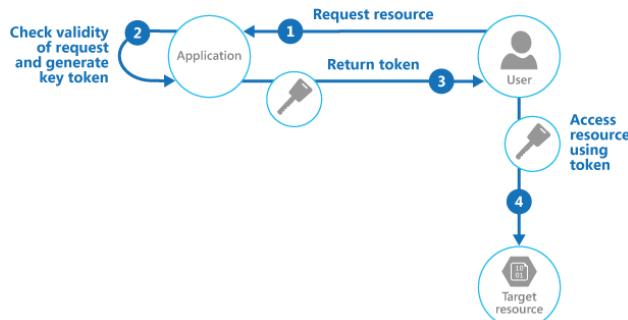
Data stores have the ability to handle upload and download of data directly, without requiring that the application perform any processing to move this data. But, this typically requires the client to have access to the security credentials for the store. This can be a useful technique to minimize data transfer costs and the requirement to scale out the application, and to maximize performance. It means, though, that the application is no longer able to manage the security of the data. After the client has a connection to the data store for direct access, the application can't act as the gatekeeper. It's no longer in control of the process and can't prevent subsequent uploads or downloads from the data store.

This isn't a realistic approach in distributed systems that need to serve untrusted clients. Instead, applications must be able to securely control access to data in a granular way, but still reduce the load on the server by setting up this connection and then allowing the client to communicate directly with the data store to perform the required read or write operations.



Valet Key pattern

- You need to resolve the problem of controlling access to a data store where the store can't manage authentication and authorization of clients. One typical solution is to restrict access to the data store's public connection and provide the client with a key or token that the data store can validate.
- This key or token is usually referred to as a valet key. It provides time-limited access to specific resources and allows only predefined operations such as reading and writing to storage or queues, or uploading and downloading in a web browser.



Applications can create and issue valet keys to client devices and web browsers quickly and easily, allowing clients to perform the required operations without requiring the application to directly handle the data transfer. This removes the processing overhead, and the impact on performance and scalability, from the application and the server.

The client uses this token to access a specific resource in the data store for only a specific period, and with specific restrictions on access permissions, as shown in the figure. After the specified period, the key becomes invalid and won't allow access to the resource.

It's also possible to configure a key that has other dependencies, such as the scope of the data. For example, depending on the data store capabilities, the key can specify a complete table in a data store, or only specific rows in a table. In cloud storage systems the key can specify a container, or just a specific item within a container.

The key can also be invalidated by the application. This is a useful approach if the client notifies the server that the data transfer operation is complete. The server can then invalidate that key to prevent further access.

Using this pattern can simplify managing access to resources because there's no requirement to create and authenticate a user, grant permissions, and then remove the user again. It also makes it easy to limit the location, the permission, and the validity period—all by simply generating a key at runtime. The important factors are to limit the validity period, and especially the location of the resource,

as tightly as possible so that the recipient can only use it for the intended purpose.



Valet Key pattern

Consider the following points when deciding how to implement this pattern:

- **Manage the validity status and period of the key.** If leaked or compromised, the key effectively unlocks the target item and makes it available for malicious use during the validity period.
- **Control the level of access the key will provide.** Typically, the key should allow the user to only perform the actions necessary to complete the operation, such as read-only access if the client shouldn't be able to upload data to the data store. For file uploads, it's common to specify a key that provides write-only permission.
- **Consider how to control users' behavior.** Typically, the key should allow the user to only perform the actions necessary to complete the operation.
- **Validate, and optionally sanitize, all uploaded data.** A malicious user that gains access to the key could upload data designed to compromise the system.
- **Audit all operations.** Many key-based mechanisms can **log operations** such as uploads, downloads, and failures. These logs can usually be incorporated into an audit process, and also used for billing if the user is charged based on file size or data volume. Use the logs to detect authentication failures that might be caused by issues with the key provider, or accidental removal of a stored access policy.

Manage the validity status and period of the key. If leaked or compromised, the key effectively unlocks the target item and makes it available for malicious use during the validity period. A key can usually be revoked or disabled, depending on how it was issued. Server-side policies can be changed or, the server key it was signed with can be invalidated. Specify a short validity period to minimize the risk of allowing unauthorized operations to take place against the data store. However, if the validity period is too short, the client might not be able to complete the operation before the key expires. Allow authorized users to renew the key before the validity period expires if multiple accesses to the protected resource are required.

Control the level of access the key will provide. Typically, the key should allow the user to only perform the actions necessary to complete the operation, such as read-only access if the client shouldn't be able to upload data to the data store. For file uploads, it's common to specify a key that provides write-only permission, as well as the location and the validity period. It's critical to accurately specify the resource or the set of resources to which the key applies.

Consider how to control users' behavior. Implementing this pattern means some loss of control over the resources users are granted access to. The level of control that can be exerted is limited by the capabilities of the policies and permissions available for the service or the target data store. For example, it's usually not possible to create a key that limits the size of the data to be written to storage, or the number of times the key can be used to access a file. This can result in huge unexpected costs for data transfer, even when used by the intended client, and might be caused by an error in the code that causes

repeated upload or download. To limit the number of times a file can be uploaded, where possible, force the client to notify the application when one operation has completed. For example, some data stores raise events the application code can use to monitor operations and control user behavior. However, it's hard to enforce quotas for individual users in a multi-tenant scenario where the same key is used by all the users from one tenant.

Validate, and optionally sanitize, all uploaded data. A malicious user that gains access to the key could upload data designed to compromise the system. Alternatively, authorized users might upload data that's invalid and, when processed, could result in an error or system failure. To protect against this, ensure that all uploaded data is validated and checked for malicious content before use.

Audit all operations. Many key-based mechanisms can log operations such as uploads, downloads, and failures. These logs can usually be incorporated into an audit process, and also used for billing if the user is charged based on file size or data volume. Use the logs to detect authentication failures that might be caused by issues with the key provider, or accidental removal of a stored access policy.

Deliver the key securely. It can be embedded in a URL that the user activates in a web page, or it can be used in a server redirection operation so that the download occurs automatically. Always use HTTPS to deliver the key over a secure channel.

Protect sensitive data in transit. Sensitive data delivered through the application will usually take place using TLS, and this should be enforced for clients accessing the data store directly.



Valet Key pattern

Consider the following points when deciding how to implement this pattern:

- **Deliver the key securely.** It can be embedded in a URL that the user activates in a web page, or it can be used in a server redirection operation so that the download occurs automatically. **Always use HTTPS to deliver the key over a secure channel.**
- **Protect sensitive data in transit.** Sensitive data delivered through the application will usually take place **using TLS**, and this should be **enforced** for clients accessing the data store directly.



Cloud Design Patterns

- **Design and implementation patterns**

Pattern	Summary
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Routing	Route requests to multiple services using a single endpoint.

<https://learn.microsoft.com/en-us/azure/architecture/patterns/>

[**Ambassador**](#) Create helper services that send network requests on behalf of a consumer service or application. [**Anti-Corruption Layer**](#) Implement a façade or adapter layer between a modern application and a legacy system. [**Backends for Frontends**](#) Create separate backend services to be consumed by specific frontend applications or interfaces. [**CQRS**](#) Segregate operations that read data from operations that update data by using separate interfaces. [**Compute Resource Consolidation**](#) Consolidate multiple tasks or operations into a single computational unit [**External Configuration Store**](#) Move configuration information out of the application deployment package to a centralized location. [**Gateway Aggregation**](#) Use a gateway to aggregate multiple individual requests into a single request. [**Gateway Offloading**](#) Offload shared or specialized service functionality to a gateway proxy. [**Gateway Routing**](#) Route requests to multiple services using a single endpoint. [**Leader Election**](#) Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances. [**Pipes and Filters**](#) Break down a task that performs complex processing into a series of separate elements that can be reused. [**Sidecar**](#) Deploy components of an application into a separate process or container to provide isolation and encapsulation. [**Static Content Hosting**](#) Deploy static content to a cloud-based storage service that can deliver them directly to the client. [**Strangler Fig**](#) Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.



Ambassador pattern

Create helper services that send network requests on behalf of a consumer service or application. An ambassador service can be thought of as an out-of-process proxy that is co-located with the client.

Context and problem

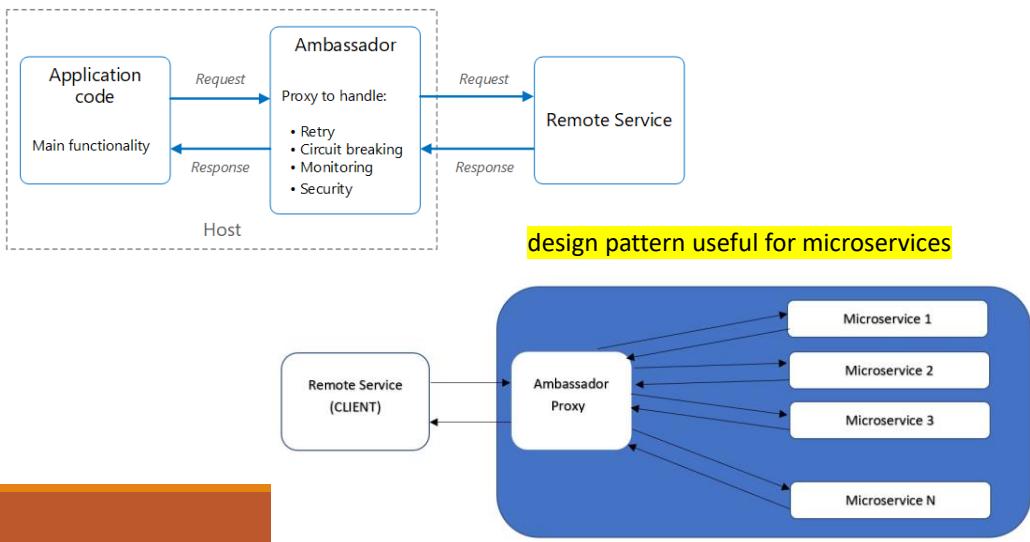
- Resilient cloud-based applications require features such as **circuit breaking, routing, metering and monitoring**, and the ability to make **network-related configuration updates**. It may be difficult or impossible to update legacy applications or existing code libraries to add these features, because the code is no longer maintained or can't be easily modified by the development team.
- Network calls may also require substantial configuration for **connection, authentication, and authorization**. If these calls are used across multiple applications, built using multiple languages and frameworks, the calls must be configured for each of these instances. In addition, network and security functionality may need to be managed by a central team within your organization. With a large code base, it can be risky for that team to update application code they aren't familiar with.

This pattern **can be useful for offloading** common client connectivity tasks such as **monitoring, logging, routing, security** (such as TLS), and resiliency patterns in a language agnostic way. It is often used with legacy applications, or other applications that are difficult to modify, in order to extend their networking capabilities. It can also enable a specialized team to implement those features.



Ambassador pattern

Put client frameworks and libraries into an external process that acts as a proxy between your application and external services. Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions.



You can also use the ambassador pattern to standardize and extend instrumentation. The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application.

Features that are offloaded to the ambassador can be managed independently of the application. You can update and modify the ambassador without disturbing the application's legacy functionality. It also allows for separate, specialized teams to implement and maintain security, networking, or authentication features that have been moved to the ambassador.

Ambassador services can be deployed as a [sidecar](#) to accompany the lifecycle of a consuming application or service. Alternatively, if an ambassador is shared by multiple separate processes on a common host, it can be deployed as a daemon or Windows service. If the consuming service is containerized, the ambassador should be created as a separate container on the same host, with the appropriate links configured for communication.



Ambassador pattern

Issues and considerations

- The proxy adds some **latency overhead**. Consider whether a client library, invoked directly by the application, is a better approach.
- Consider the **possible impact of including generalized features** in the proxy. For example, **the ambassador could handle retries**, but that might **not be safe unless all operations are idempotent**.
- Consider a mechanism to **allow the client to pass some context to the proxy**, as well as back to the client. For example, include HTTP request headers to opt out of retry or specify the maximum number of times to retry.
- Consider how you will **package and deploy** the proxy.
- Consider whether to use a single shared instance for all clients or an instance for each client.

An **idempotent operation** is one that has **no additional effect if it is called more than once** with the same input parameters. For example, removing an item from a set can be considered an idempotent operation on the set.



Ambassador pattern

Use this pattern when you:

- Need to **offload cross-cutting client connectivity concerns to infrastructure developers** or other more specialized teams.
- Need to support cloud or cluster connectivity requirements **in a legacy application or an application that is difficult to modify**.

This pattern may not be suitable:

- When network request latency is critical. A proxy will introduce some overhead, although minimal, and in some cases this may affect the application.
- When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
- When connectivity features cannot be generalized and require deeper integration with the client application.



Gateway Aggregation pattern

Use a gateway to aggregate multiple individual requests into a single request.
This pattern is useful when a client must make multiple calls to different backend systems to perform an operation.

Context and problem

- Consider situations such as an IoT Hub receiving requests from different clients. It could easily aggregate individual requests to remote agents (backends).
- To perform a single task, a client may have to make multiple calls to various backend services. An application that relies on many services to perform a task must expend resources on each request. When any new feature or service is added to the application, additional requests are needed, further increasing resource requirements and network calls. This chattiness between a client and a backend can adversely impact the performance and scale of the application.
- Microservice architectures have made this problem more common, as applications built around many smaller services naturally have a higher amount of cross-service calls.



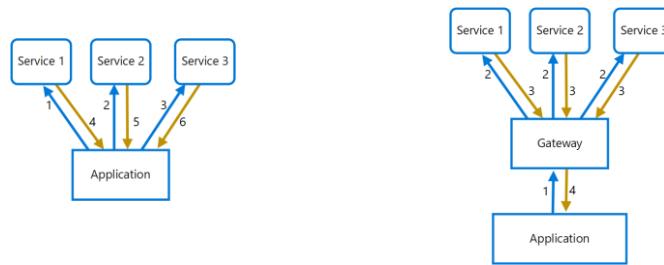
Gateway Aggregation pattern

- Use a gateway to reduce chattiness between the client and the services. The gateway receives client requests, dispatches requests to the various backend systems, and then aggregates the results and sends them back to the requesting client.
- This pattern can reduce the number of requests that the application makes to backend services, and improve application performance over high-latency networks.



Gateway Aggregation pattern

In the following diagram, the application sends a request to the gateway (1). The request contains a package of additional requests. The gateway decomposes these and processes each request by sending it to the relevant service (2). Each service returns a response to the gateway (3). The gateway combines the responses from each service and sends the response to the application (4). The application makes a single request and receives only a single response from the gateway.





Gateway Aggregation pattern

Issues and considerations

- The gateway should not introduce service coupling across the backend services.
- The gateway should be located near the backend services to reduce latency as much as possible.
- The gateway service may introduce a single point of failure. Ensure the gateway is properly designed to meet your application's availability requirements.
- The gateway may introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can be scaled to meet your anticipated growth.
- Perform load testing against the gateway to ensure you don't introduce cascading failures for services.
- If one or more service calls takes too long, it may be acceptable to timeout and return a partial set of data. Consider how your application will handle this scenario.

- Use asynchronous I/O to ensure that a delay at the backend doesn't cause performance issues in the application.
- Implement a resilient design, using techniques such as [bulkheads](#), [circuit breaking](#), [retry](#), and timeouts.
- Implement distributed tracing using correlation IDs to track each individual call.
- Monitor request metrics and response sizes.
- Consider returning cached data as a failover strategy to handle failures.
- Instead of building aggregation into the gateway, consider placing an aggregation service behind the gateway. Request aggregation will likely have different resource requirements than other services in the gateway and may impact the gateway's routing and offloading functionality



Gateway Aggregation pattern

Use this pattern when:

- A client needs to communicate with multiple backend services to perform an operation.
- The client may use networks with significant latency, such as cellular networks.

This pattern may not be suitable when:

- You want to reduce the number of calls between a client and a single service across multiple operations. In that scenario, it may be better to add a batch operation to the service.
- The client or application is located near the backend services and latency is not a significant factor.



Gateway Routing pattern

Route requests to multiple services or multiple service instances using a single endpoint. The pattern is useful when you want to:

- Expose **multiple services** on a single endpoint and route to the appropriate service based on the request
- Expose **multiple instances** of the same service on a single endpoint for load balancing or availability purposes
- Expose **differing versions of the same service** on a single endpoint and route traffic across the different versions

Context and problem

- Consider the following scenarios.
 - **Multiple disparate services.**
 - **Multiple instances of the same service**
 - **Multiple versions of the same service**

When a client needs to consume multiple services, multiple service instances or a combination of both, the client must be updated when services are added or removed. Consider the following scenarios.

• **Multiple disparate services** - An e-commerce application might provide services such as search, reviews, cart, checkout, and order history. Each service has a different API that the client must interact with, and the client must know about each endpoint in order to connect to the services. If an API changes, the client must be updated as well. If you refactor a service into two or more separate services, the code must change in both the service and the client.

• **Multiple instances of the same service** - The system can require running multiple instances of the same service in the same or different regions. Running multiple instances can be done for load balancing purposes or to meet availability requirements. Each time an instance is spun up or down to match demand, the client must be updated.

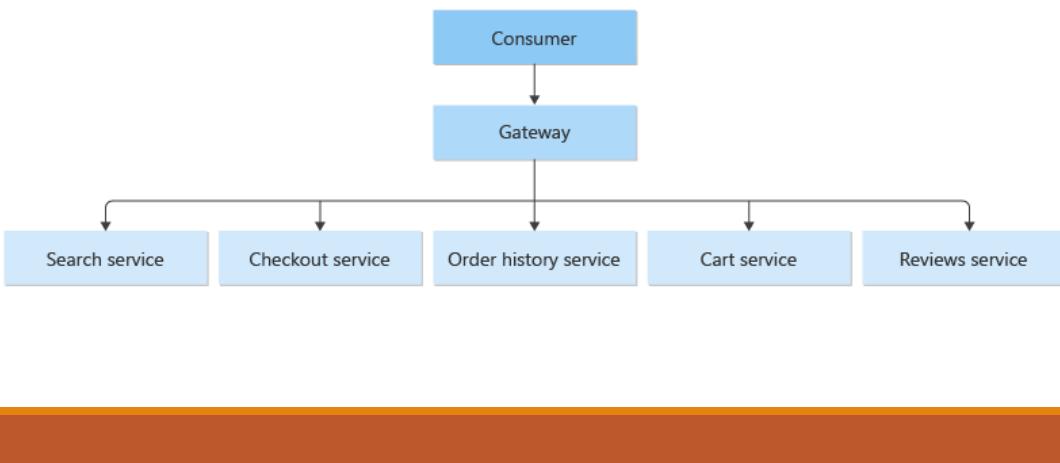
• **Multiple versions of the same service** - As part of the deployment strategy, new versions of a service can be deployed along side existing versions. This is known as blue green deployments. In these scenarios, the client must be updated each time there are changes to the percentage of traffic being routed to the new version and existing endpoint.



Gateway Routing pattern

Place a gateway in front of a set of applications, services, or deployments. Use application Layer 7 routing to route the request to the appropriate instances...

Multiple disparate services:



With this pattern, the client application only needs to know about a single endpoint and communicate with a single endpoint. The following illustrate how the Gateway Routing pattern addresses the three scenarios outlined in the context and problem section.

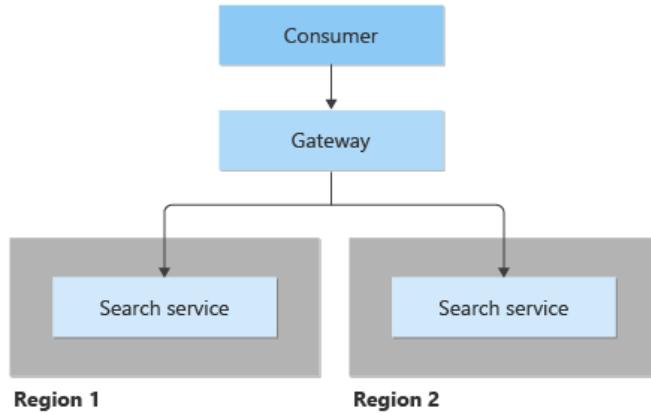
The gateway routing pattern is useful in this scenario where a client is consuming multiple services. If a service is consolidated, decomposed or replaced, the client doesn't necessarily require updating. It can continue making requests to the gateway, and only the routing changes.

A gateway also lets you abstract backend services from the clients, allowing you to keep client calls simple while enabling changes in the backend services behind the gateway. Client calls can be routed to whatever service or services need to handle the expected client behavior, allowing you to add, split, and reorganize services behind the gateway without changing the client.



Gateway Routing pattern

Multiple instances of the same service:



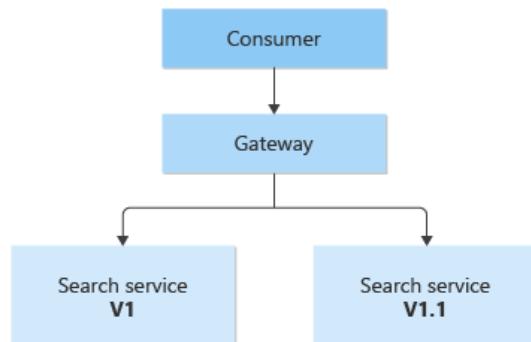
Elasticity is key in cloud computing. Services can be spun up to meet increasing demand or spun down when demand is low to save money. The complexity of registering and unregistering service instances is encapsulated in the gateway. The client is unaware of an increase or decreases in the number of services.

Service instances can be deployed in a single or multiple regions. The [Geode pattern](#) details how a multi-region, active-active deployment can improve latency and increase availability of a service.



Gateway Routing pattern

- Multiple versions of the same service



This pattern can be used for deployments, by allowing you to manage how updates are rolled out to users. When a new version of your service is deployed, it can be deployed in parallel with the existing version. Routing lets you control what version of the service is presented to the clients, giving you the flexibility to use various release strategies, whether incremental, parallel, or complete rollouts of updates. Any issues discovered after the new service is deployed can be quickly reverted by making a configuration change at the gateway, without affecting clients.



Gateway Routing pattern

Issues and considerations

- The gateway service can introduce a **single point of failure**.
- The gateway service can introduce a **bottleneck**.
- Perform **load testing against the gateway** to ensure you don't introduce cascading failures for services.
- Gateway routing is **level 7**. It can be based on IP, port, header, or URL.
- Gateway services can be **global or regional**.
- **The gateway service is the public endpoint for services it sits in front of.** Consider limiting public network access to the backend services, by making the services only accessible via the gateway or via a private virtual network.

- The gateway service can introduce a single point of failure. Ensure it's properly designed to meet your availability requirements. Consider resiliency and fault tolerance capabilities in the implementation.
- The gateway service can introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can easily scale in line with your growth expectations.
- Perform load testing against the gateway to ensure you don't introduce cascading failures for services.
- Gateway routing is level 7. It can be based on IP, port, header, or URL.
- Gateway services can be global or regional. Use a global gateway if your solution requires multi-region deployments of services. Consider using Application Gateway if you have a regional workload that requires granular control how traffic is balanced. For example, you want to balance traffic between virtual machines.
- The gateway service is the public endpoint for services it sits in front of. Consider limiting public network access to the backend services, by making the services only accessible via the gateway or via a private virtual network.



Gateway Routing pattern

Use this pattern when:

- A client needs to consume multiple services that can be accessed behind a gateway.
- You want to simplify client applications by using a single endpoint.
- You need to route requests from externally addressable endpoints to internal virtual endpoints, such as exposing ports on a VM to cluster virtual IP addresses.
- A client needs to consume services running in multiple regions for latency or availability benefits.
- A client needs to consume a variable number of service instances.
- You want to implement a deployment strategy where clients access multiple versions of the service at the same time.

- The gateway service can introduce a single point of failure. Ensure it's properly designed to meet your availability requirements. Consider resiliency and fault tolerance capabilities in the implementation.
- The gateway service can introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can easily scale in line with your growth expectations.
- Perform load testing against the gateway to ensure you don't introduce cascading failures for services.
- Gateway routing is level 7. It can be based on IP, port, header, or URL.
- Gateway services can be global or regional. Use a global gateway if your solution requires multi-region deployments of services. Consider using Application Gateway if you have a regional workload that requires granular control how traffic is balanced. For example, you want to balance traffic between virtual machines.
- The gateway service is the public endpoint for services it sits in front of. Consider limiting public network access to the backend services, by making the services only accessible via the gateway or via a private virtual network.



Cloud Design Patterns

Reliability

Pattern	Summary
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes, to smooth intermittent heavy loads.
Throttling	Control the consumption of resources by an instance of an application, an individual tenant, or an entire service.



Queue-Based Load Leveling pattern

Use a queue that acts as a buffer between an application task and a service it invokes in order to smooth intermittent heavy loads that can cause the service to fail or the task to time out. This can help to minimize the impact of peaks in demand on availability and responsiveness for both the task and the service.

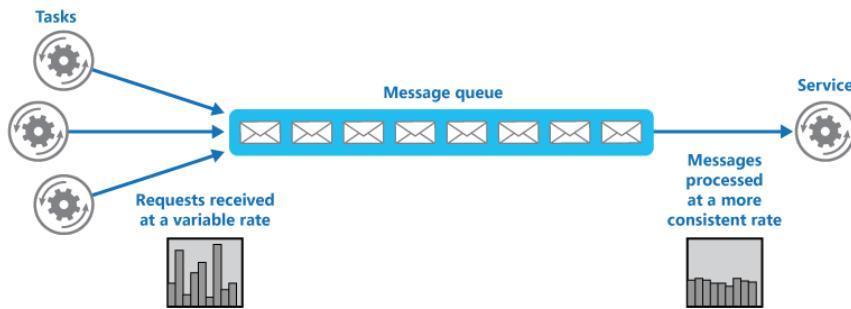
Context and problem

- Many solutions in the cloud involve running tasks that invoke services. In this environment, if a service is subjected to intermittent heavy loads, it can cause **performance or reliability issues**.
- A service could be part of the same solution as the tasks that use it (e.g. in the same cluster), or it could be a third-party service providing access to frequently used resources such as a cache or a storage service. If the same service is used by a number of tasks running concurrently, it can be **difficult to predict the volume of requests to the service at any time**.
- In case of overloading, the service could be **unable to respond to requests in a timely manner**. Flooding a service with a large number of concurrent requests can also result in the service failing if it's unable to handle the contention these requests cause.



Queue-Based Load Leveling pattern

Introduce a **queue** between the task and the service. The task and the service run asynchronously. The task posts a message containing the data required by the service to a queue. The queue acts as a buffer, storing the message until it's retrieved by the service. The service retrieves the messages from the queue and processes them. Requests from a number of tasks, which can be generated at a highly variable rate, can be passed to the service through the same message queue.



Question: How to size the queue?

The queue decouples the tasks from the service, and the service can handle the messages at its own pace regardless of the volume of requests from concurrent tasks. Additionally, there's no delay to a task if the service isn't available at the time it posts a message to the queue.

This pattern provides the following benefits:

- It can help to maximize availability because delays arising in services won't have an immediate and direct impact on the application, which can continue to post messages to the queue even when the service isn't available or isn't currently processing messages.
- It can help to maximize scalability because both the number of queues and the number of services can be varied to meet demand.
- It can help to control costs because the number of service instances deployed only have to be adequate to meet average load rather than the peak load.
- Some services implement throttling when demand reaches a threshold beyond which the system could fail. Throttling can reduce the functionality available. You can implement load leveling with these services to ensure that this threshold isn't reached.



Queue-Based Load Leveling pattern

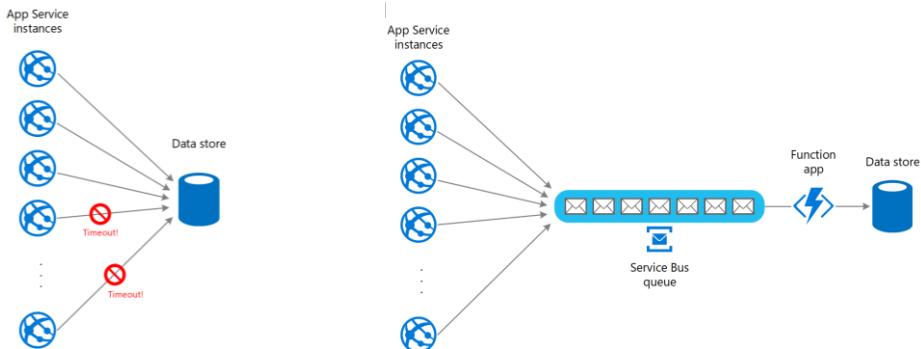
Consider the following points when deciding how to implement this pattern:

- It's necessary to **implement application logic** (see next slide) that controls the rate at which services handle messages to avoid overwhelming the target resource. Avoid passing spikes in demand to the next stage of the system.
- Test the system under load to ensure that it provides the required leveling, and adjust the number of queues and the number of service instances that handle messages to achieve this.
- Message queues are a one-way communication mechanism. If a task expects a reply from a service, it might be necessary to implement a mechanism that the service can use to send a response.
- Be careful if you apply autoscaling to services that are listening for requests on the queue. This can result in increased contention for any resources that these services share and diminish the effectiveness of using the queue to level the load.
- The pattern can lose information depending on the persistence of the Queue. If your queue **crashes or drops information (due to system limits)** there's a possibility that you don't have a guaranteed delivery. The behavior of your queue and system limits needs to be taken into consideration based on the needs of your solution.



Queue-Based Load Leveling pattern

- This pattern is useful to any application that uses services that are subject to overloading.
- This pattern is not useful if the application expects a response from the service with minimal latency.
- Example: A web app writes data to an external data store. If a large number of instances of the web app run concurrently, the data store might be unable to respond to requests quickly enough, causing requests to time out, be throttled, or otherwise fail.



To resolve this, you can use a queue to level the load between the application instances and the data store. A Function app reads the messages from the queue and performs the read/write requests to the data store. The application logic in the function app can control the rate at which it passes requests to the data store, to prevent the store from being overwhelmed. (Otherwise the function app will just re-introduce the same problem at the back end.)



Throttling pattern

Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service. This can allow the system to continue to function and meet service level agreements, even when an increase in demand places an extreme load on resources.

Context and problem

If the processing requirements of the system exceed the capacity of the resources that are available, it will suffer from poor performance and can even fail. If the system has to meet an agreed level of service, such failure could be unacceptable.

The load on a cloud application typically varies over time based on the number of active users or the types of activities they are performing. For example, more users are likely to be active during **business hours** or the system might be required to perform **computationally expensive analytics at the end of each month**. There might also be sudden and unanticipated bursts in activity.

There're many strategies available for handling varying load in the cloud, depending on the business goals for the application. One strategy is to use autoscaling to match the provisioned resources to the user needs at any given time. This has the potential to consistently meet user demand, while optimizing running costs. However, while autoscaling can trigger the provisioning of additional resources, this provisioning isn't immediate. If demand grows quickly, there can be a window of time where there's a resource deficit.

Throttling pattern

An **alternative strategy to autoscaling** is to allow applications to **use resources only up to a limit, and then throttle them when this limit is reached**. The system should monitor how it's using resources so that, when usage exceeds the threshold, it can **throttle requests from one or more users**. This will enable the system to continue functioning and meet any service level agreements (SLAs) that are in place. For more information on monitoring resource usage, see the Instrumentation and Telemetry Guidance..

The system could implement several throttling strategies, including:

- **Rejecting requests** from an individual user who's already accessed system APIs more than n times per second over a given period of time.
- **Disabling or degrading** the functionality of selected nonessential services.

The system could implement several throttling strategies, including:

- Rejecting requests from an individual user who's already accessed system APIs more than n times per second over a given period of time. This requires the system to meter the use of resources for each tenant or user running an application. For more information, see the [Service Metering Guidance](#).
- Disabling or degrading the functionality of selected nonessential services so that essential services can run unimpeded with sufficient resources. For example, if the application is streaming video output, it could switch to a lower resolution.



Throttling pattern

The system could implement several throttling strategies, including (cont'd):

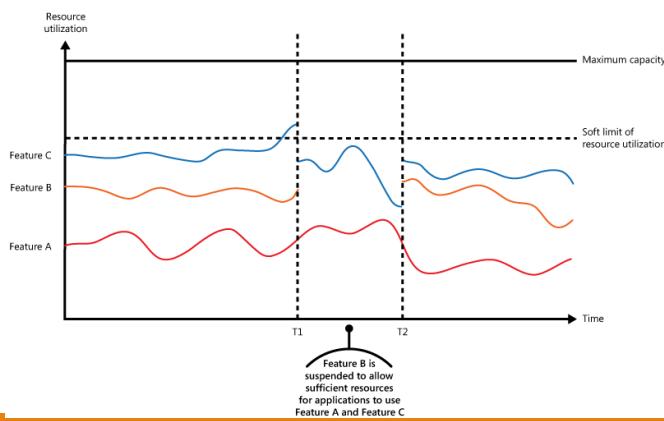
- Using **load leveling** to smooth the volume of activity (this approach is covered in more detail by the [Queue-based Load Leveling pattern](#)). In a multi-tenant environment, this approach will reduce the performance for every tenant. Requests for other tenants can be held back, and handled when the backlog has eased. The [Priority Queue pattern](#) could be used to help implement this approach.
- **Deferring operations** being performed on behalf of **lower priority** applications or tenants.
- You should **be careful when integrating with some 3rd-party services that might become unavailable or return errors**. Reduce the number of concurrent requests being processed so that the logs do not unnecessarily fill up with errors. You also avoid the costs that are associated with needlessly retrying the processing of requests that would fail because of that 3rd-party service.

- Using load leveling to smooth the volume of activity (this approach is covered in more detail by the [Queue-based Load Leveling pattern](#)). In a multi-tenant environment, this approach will reduce the performance for every tenant. If the system must support a mix of tenants with different SLAs, the work for high-value tenants might be performed immediately. Requests for other tenants can be held back, and handled when the backlog has eased. The [Priority Queue pattern](#) could be used to help implement this approach.
- Deferring operations being performed on behalf of lower priority applications or tenants. These operations can be suspended or limited, with an exception generated to inform the tenant that the system is busy and that the operation should be retried later.
- You should be careful when integrating with some 3rd-party services that might become unavailable or return errors. Reduce the number of concurrent requests being processed so that the logs do not unnecessarily fill up with errors. You also avoid the costs that are associated with needlessly retrying the processing of requests that would fail because of that 3rd-party service. Then, when requests are processed successfully, go back to regular unthrottled request processing. One library that implements this functionality is [NServiceBus](#).



Throttling pattern

The figure shows an area graph for **resource use** (a combination of memory, CPU, bandwidth, and other factors) against time for applications that are making use of three features. A **feature** is an area of functionality, such as a component that performs a specific set of tasks, a piece of code that performs a complex calculation, or an element that provides a service such as an in-memory cache. These features are labeled A, B, and C.



The area immediately below the line for a feature indicates the resources that are used by applications when they invoke this feature. For example, the area below the line for Feature A shows the resources used by applications that are making use of Feature A, and the area between the lines for Feature A and Feature B indicates the resources used by applications invoking Feature B. Aggregating the areas for each feature shows the total resource use of the system.

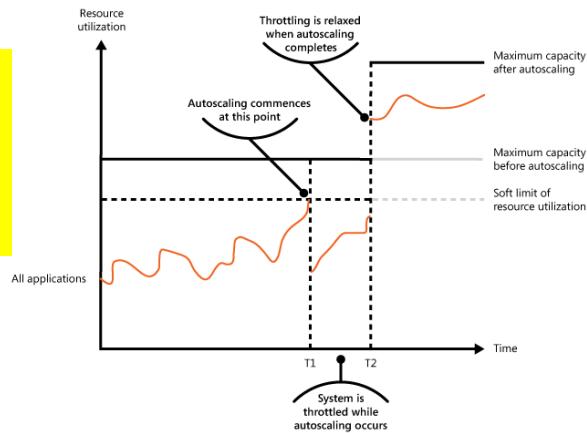
The previous figure illustrates the effects of deferring operations. Just prior to time T1, the total resources allocated to all applications using these features reach a threshold (the limit of resource use). At this point, the applications are in danger of exhausting the resources available. In this system, Feature B is less critical than Feature A or Feature C, so it's temporarily disabled and the resources that it was using are released. Between times T1 and T2, the applications using Feature A and Feature C continue running as normal. Eventually, the resource use of these two features diminishes to the point when, at time T2, there is sufficient capacity to enable Feature B again.



Throttling pattern

- The autoscaling and throttling approaches can also be combined to help keep the applications responsive and within SLAs. If the demand is expected to remain high, throttling provides a temporary solution while the system scales out. At this point, the full functionality of the system can be restored.
- The next figure shows an area graph of the overall resource use by all applications running in a system against time, and illustrates how throttling can be combined with autoscaling.

Throttling can be used as a temporary measure while a system autoscales.



At time T1, the threshold specifying the soft limit of resource use is reached. At this point, the system can start to scale out. However, **if the new resources don't become available quickly enough**, then the existing resources might be exhausted and the system could fail. To prevent this from occurring, the system is temporarily throttled, as described earlier. When autoscaling has completed and the additional resources are available, throttling can be relaxed.

Throttling pattern

Issues and considerations

- Throttling an application, and the strategy to use, is an architectural decision that impacts the entire design of a system. Throttling should be considered early in the application design process because it isn't easy to add once a system has been implemented.
- Throttling must be performed quickly. The system must be capable of detecting an increase in activity and react accordingly. The system must also be able to revert to its original state quickly after the load has eased. This requires that the appropriate performance data is continually captured and monitored.
- Using the returned error code the client application can understand that the refusal to perform an operation is due to throttling. The client application can wait for a period before retrying the request

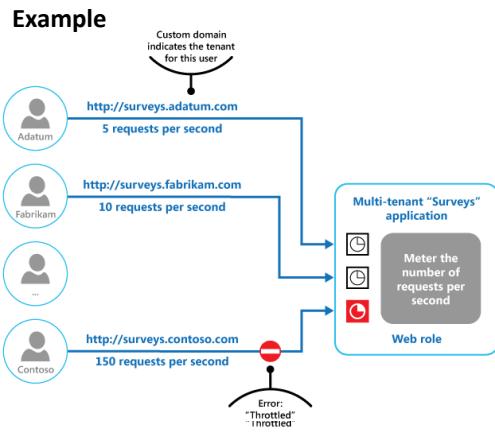
- If throttling is being used as a temporary measure while a system autoscales, and if resource demands grow very quickly, the system might not be able to continue functioning—even when operating in a throttled mode. If this isn't acceptable, consider maintaining larger capacity reserves and configuring more aggressive autoscaling.



Throttling pattern

Use this pattern:

- To ensure that a system continues to meet service level agreements.
- To prevent a single tenant from monopolizing the resources provided by an application.
- To handle bursts in activity.
- To help cost-optimize a system by limiting the maximum resource levels needed to keep it functioning.



- If throttling is being used as a temporary measure while a system autoscales, and if resource demands grow very quickly, the system might not be able to continue functioning—even when operating in a throttled mode. If this isn't acceptable, consider maintaining larger capacity reserves and configuring more aggressive autoscaling.



Cloud Design Patterns

High Availability

Pattern	Summary
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.



Bulkhead pattern

The Bulkhead pattern is a type of application design that is **tolerant of failure**. In a bulkhead architecture, **elements of an application are isolated into pools so that if one fails, the others will continue to function**. It's named after the sectioned partitions (bulkheads) of a ship's hull. If the hull of a ship is compromised, only the damaged section fills with water, which prevents the ship from sinking.

Context and problem

- A cloud-based application may include multiple services, with each service having one or more consumers. Excessive load or failure in a service will impact all consumers of the service.
- A consumer could send requests to multiple services. If a service is misconfigured or not responding, the resources used by the client's request may not be freed in a timely manner. As requests to the service continue, those resources may be exhausted. For example, the client's connection pool may be exhausted. At that point, requests by the consumer to other services are affected. Eventually the consumer can no longer send requests to other services, not just the original unresponsive service.

The same issue of resource exhaustion affects services with multiple consumers. A large number of requests originating from one client may exhaust available resources in the service. Other consumers are no longer able to consume the service, causing a cascading failure effect.



Bulkhead pattern

- **Partition service instances into different groups, based on consumer** load and availability requirements. This design helps to **isolate failures**, and allows you to sustain service functionality for some consumers, even during a failure.
- A consumer can also **partition resources**, to ensure that **resources used to call one service don't affect the resources used to call another service**. For example, a consumer that calls multiple services may be assigned a **connection pool for each service**. If a service begins to fail, it only affects the connection pool assigned for that service, **allowing the consumer to continue using the other services**.

The benefits of this pattern include:

- **Isolates consumers and services from cascading failures.** An issue affecting a consumer or service can be isolated within its own bulkhead, preventing the entire solution from failing.
- **Allows you to preserve some functionality** in the event of a service failure. Other services and features of the application will continue to work.
- Allows you to deploy services that offer a **different quality of service for consuming applications**. A high-priority consumer pool can be configured to use high-priority services.

For example, in distributed service architectures, a connection pool is a mechanism used to manage and reuse a collection of established network connections to external services or resources. These connections are typically maintained between a client application and various backend services, such as databases, APIs, or other microservices.

Here's how a connection pool typically works and why it's important:

1.Connection Management: When a client application needs to communicate with a backend service, it often needs to establish a network connection. Establishing and tearing down connections can be resource-intensive and time-consuming.

2.Pooling Connections: Instead of creating and destroying connections for each request, a connection pool keeps a set of pre-established connections ready for use. When a client application requires a connection, it requests one from the pool.

3.Reuse and Efficiency: By reusing existing connections from the pool, the overhead of establishing new connections is avoided. This improves performance and reduces latency, especially in distributed architectures where network latency can be a significant factor.

4.Connection Limits: Connection pools can also help manage the number of simultaneous connections to backend services. They can enforce connection limits to prevent overwhelming the backend service with too many concurrent connections, which can lead to performance degradation or denial of service.

5. Connection Recycling: In addition to reusing connections, connection pools may implement strategies for recycling or refreshing connections periodically to prevent issues like stale connections or resource leaks.

6. Configurability: Connection pools typically offer configuration options to tune parameters such as the maximum number of connections allowed, timeouts for idle connections, and strategies for connection acquisition and eviction.

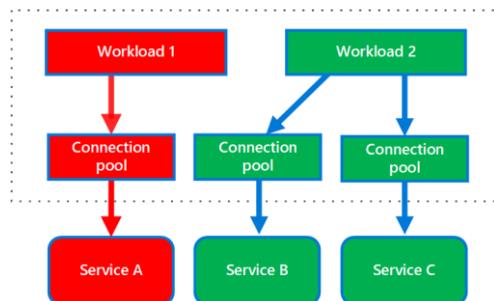
7. Fault Tolerance: Some connection pool implementations incorporate fault tolerance mechanisms, such as connection health checks or automatic reconnection, to handle transient failures or network issues gracefully.

In distributed service architectures where multiple client instances interact with various backend services concurrently, efficient connection management via connection pooling is crucial for optimizing resource utilization, improving performance, and ensuring scalability and reliability of the overall system.



Bulkhead pattern

- The following diagram shows bulkheads structured around connection pools that call individual services. If Service A fails or causes some other issue, the connection pool is isolated, so only workloads using the thread pool assigned to Service A are affected. Workloads that use Service B and C are not affected and can continue working without interruption.



For simple operations at small scale, the steps involved in opening and closing a connection are not expensive enough to warrant worrying about. As your application scales up, however, the constant opening and closing of connections becomes more expensive and can begin to impact your application's performance.

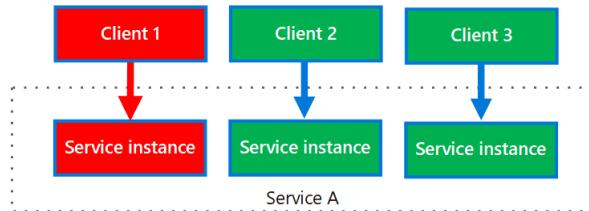
Often, it makes sense to find a way of keeping connections open and passing them from operation to operation as they're needed, rather than opening and closing a brand new connection for each operation.

Database connection pooling is a way to reduce the cost of opening and closing connections by maintaining a "pool" of open connections that can be passed from database operation to database operation as needed. This way, we are spared the expense of having to open and close a brand new connection for each operation the database is asked to perform.



Bulkhead pattern

- The next diagram shows multiple clients calling a single service. Each client is assigned a separate service instance. Client 1 has made too many requests and overwhelmed its instance. Because each service instance is isolated from the others, the other clients can continue making calls.





Bulkhead pattern

Issues and considerations

- Define partitions around the **business and technical requirements** of the application.
- When partitioning services or consumers into bulkheads, consider the **level of isolation offered by the technology** as well as the overhead in terms of cost, performance and manageability.
- Consider **combining bulkheads with retry, circuit breaker, and throttling patterns** to provide more sophisticated fault handling.
- When partitioning services into bulkheads, consider deploying them into **separate virtual machines, containers, or processes**. Containers offer a good balance of resource isolation with fairly low overhead.
- Services that communicate using asynchronous messages **can be isolated through different sets of queues**. Each queue can have a dedicated set of instances processing messages on the queue, or a single group of instances using an algorithm to dequeue and dispatch processing.

When partitioning consumers into bulkheads, consider using processes, thread pools, and semaphores. Projects like [resilience4j](#) and [Polly](#) offer a framework for creating consumer bulkheads.

Determine the level of granularity for the bulkheads. For example, if you want to distribute tenants across partitions, you could place each tenant into a separate partition, or put several tenants into one partition.



Bulkhead pattern

Use this pattern to:

- Isolate resources used to consume a set of backend services, especially if the application can provide some level of functionality even when one of the services is not responding.
- Isolate critical consumers from standard consumers.
- Protect the application from cascading failures.

This pattern may not be suitable when:

- Less efficient use of resources may not be acceptable in the project.
- The added complexity is not necessary



Bulkhead pattern

Example

The following Kubernetes configuration file creates an isolated container to run a single service, with its own CPU and memory resources and limits.

```
apiVersion: v1
kind: Pod
metadata:
  name: drone-management
spec:
  containers:
    - name: drone-management-container
      image: drone-service
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "1"
```



Circuit Breaker pattern

Handle faults that might take a variable amount of time to recover from, when connecting to a remote service or resource. This can improve the stability and resiliency of an application.

Context and problem

- There can be situations where faults are due to unanticipated events, and that might take long to fix. In these situations it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.
- If a service is very busy, failure in one part of the system might lead to cascading failures. An operation that invokes a service could be configured to implement a timeout, and reply with a failure message if the service fails to respond within this period. However, this strategy could cause many concurrent requests to the same operation to be blocked until the timeout period expires. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on. Consequently, these resources could become exhausted. It would be preferable for the operation to fail immediately, and only attempt to invoke the service if it is likely to succeed.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them by using a strategy such as the [Retry pattern](#).

However, there can also be situations where faults are due to unanticipated events, and that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure in one part of the system might lead to cascading failures. For example, an operation that invokes a service could be configured to implement a timeout, and reply with a failure message if the service fails to respond within this period. However, this strategy could cause many concurrent requests to the same operation to be blocked until the timeout period expires. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on. Consequently, these resources could become exhausted, causing failure of other possibly unrelated parts of the system that need to use the same resources. In these situations, it would be preferable for the operation to fail immediately, and only attempt to invoke the service if it is likely to succeed. Note that setting a shorter timeout might help to resolve this problem, but the timeout shouldn't be so short that the operation fails most of the time, even if the request to the service would

eventually succeed.



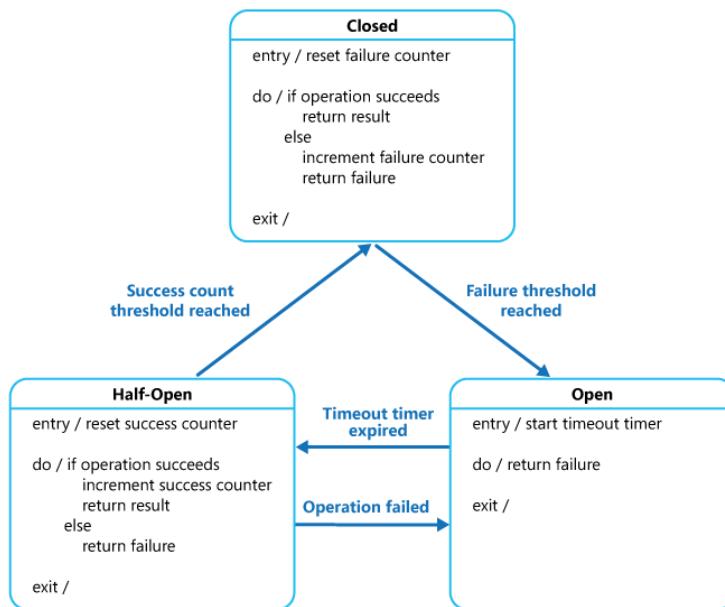
Circuit Breaker pattern

- The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.
- A circuit breaker acts as a **proxy for operations that might fail**. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.
- The proxy can be implemented as a **state machine** with the following states that mimic the functionality of an electrical circuit breaker.

The purpose of the Circuit Breaker pattern is different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it'll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.



Circuit Breaker pattern



• **Closed:** In this state, the request from the application is routed to the operation. The proxy maintains a count of the number of recent failures, and if the call to the operation is unsuccessful the proxy increments this count. If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the **Open** state. At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the **Half-Open** state.

• The purpose of the timeout timer is to give the system time to fix the problem that caused the failure before allowing the application to try to perform the operation again.

• **Open:** The request from the application fails immediately and an exception is returned to the application.

• **Half-Open:** A limited number of requests from the application are allowed to pass through and invoke the operation. If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state (the failure counter is reset). If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure.

• The **Half-Open** state is useful to prevent a recovering service from suddenly being flooded with requests. As a service recovers, it might be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work can cause the service to time out or fail again.

The Circuit Breaker pattern provides stability while the system recovers from a

failure and minimizes the impact on performance. It can help to maintain the response time of the system by quickly rejecting a request for an operation that's likely to fail, rather than waiting for the operation to time out, or never return. If the circuit breaker raises an event each time it changes state, this information can be used to monitor the health of the part of the system protected by the circuit breaker, or to alert an administrator when a circuit breaker trips to the **Open** state.

The pattern is customizable and can be adapted according to the type of the possible failure. For example, you can apply an increasing timeout timer to a circuit breaker. You could place the circuit breaker in the **Open** state for a few seconds initially, and then if the failure hasn't been resolved increase the timeout to a few minutes, and so on. In some cases, rather than the **Open** state returning failure and raising an exception, it could be useful to return a default value that is meaningful to the application.



Circuit Breaker pattern

Issues and considerations

- **Concurrency.** The same circuit breaker could be accessed by a large number of concurrent instances of an application. The implementation should not block concurrent requests or add excessive overhead to each call to an operation.
- **Accelerated Circuit Breaking.** Sometimes a failure response can contain enough information for the circuit breaker to trip immediately and stay tripped for a minimum amount of time.
- **Replaying Failed Requests.** In the **Open** state, rather than simply failing quickly, a circuit breaker could also record the details of each request to a journal and arrange for these requests to be replayed when the remote resource or service becomes available.
- **Inappropriate Timeouts on External Services.** A circuit breaker might not be able to fully protect applications from operations that fail in external services that are configured with a lengthy timeout period. If the timeout is too long, a thread running a circuit breaker might be blocked for an extended period before the circuit breaker indicates that the operation has failed.

Issues and considerations

You should consider the following points when deciding how to implement this pattern:

Exception Handling. An application invoking an operation through a circuit breaker must be prepared to handle the exceptions raised if the operation is unavailable. The way exceptions are handled will be application specific. For example, an application could temporarily degrade its functionality, invoke an alternative operation to try to perform the same task or obtain the same data, or report the exception to the user and ask them to try again later.

Types of Exceptions. A request might fail for many reasons, some of which might indicate a more severe type of failure than others. For example, a request might fail because a remote service has crashed and will take several minutes to recover, or because of a timeout due to the service being temporarily overloaded. A circuit breaker might be able to examine the types of exceptions that occur and adjust its strategy depending on the nature of these exceptions. For example, it might require a larger number of timeout exceptions to trip the circuit breaker to the **Open** state compared to the number of failures due to the service being completely unavailable.

Logging. A circuit breaker should log all failed requests (and possibly successful requests) to enable an administrator to monitor the health of the operation.

Recoverability. You should configure the circuit breaker to match the likely recovery pattern of the operation it's protecting. For example, if the circuit breaker remains in the **Open** state for a long period, it could raise exceptions even if the

reason for the failure has been resolved. Similarly, a circuit breaker could fluctuate and reduce the response times of applications if it switches from the **Open** state to the **Half-Open** state too quickly.

Testing Failed Operations. In the **Open** state, rather than using a timer to determine when to switch to the **Half-Open** state, a circuit breaker can instead periodically ping the remote service or resource to determine whether it's become available again. This ping could take the form of an attempt to invoke an operation that had previously failed, or it could use a special operation provided by the remote service specifically for testing the health of the service, as described by the [Health Endpoint Monitoring pattern](#).

Manual Override. In a system where the recovery time for a failing operation is extremely variable, it's beneficial to provide a manual reset option that enables an administrator to close a circuit breaker (and reset the failure counter). Similarly, an administrator could force a circuit breaker into the **Open** state (and restart the timeout timer) if the operation protected by the circuit breaker is temporarily unavailable.

Concurrency. The same circuit breaker could be accessed by a large number of concurrent instances of an application. The implementation should not block concurrent requests or add excessive overhead to each call to an operation.

Resource Differentiation. Be careful when using a single circuit breaker for one type of resource if there might be multiple underlying independent providers. For example, in a data store that contains multiple shards, one shard might be fully accessible while another is experiencing a temporary issue. If the error responses in these scenarios are merged, an application might try to access some shards even when failure is highly likely, while access to other shards might be blocked even though it's likely to succeed.

Accelerated Circuit Breaking. Sometimes a failure response can contain enough information for the circuit breaker to trip immediately and stay tripped for a minimum amount of time. For example, the error response from a shared resource that's overloaded could indicate that an immediate retry isn't recommended and that the application should instead try again in a few minutes.

Note

A service can return HTTP 429 (Too Many Requests) if it is throttling the client, or HTTP 503 (Service Unavailable) if the service is not currently available. The response can include additional information, such as the anticipated duration of the delay.

Replaying Failed Requests. In the **Open** state, rather than simply failing quickly, a circuit breaker could also record the details of each request to a journal and arrange for these requests to be replayed when the remote resource or service becomes available.

Inappropriate Timeouts on External Services. A circuit breaker might not be able to fully protect applications from operations that fail in external services that are configured with a lengthy timeout period. If the timeout is too long, a thread running a circuit breaker might be blocked for an extended period before the circuit breaker indicates that the operation has failed. In this time, many other

application instances might also try to invoke the service through the circuit breaker and tie up a significant number of threads before they all fail.



Cloud Design Patterns

Messaging

Pattern	Summary
<u>Choreography</u>	Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control.
<u>Priority Queue</u>	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
<u>Publisher-Subscriber</u>	Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.



Choreography pattern

Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control.

Context and problem

- In microservices architecture, application is divided into several small services that work together to process a business transaction end-to-end. To **lower coupling between services**, each service is responsible for a single business operation.
- Services communicate with each other by using well-defined APIs. Even a single business operation can result in multiple point-to-point calls among all services. A **common pattern for communication is to use a centralized service that acts as the orchestrator**. It manages the workflow of the entire business transaction. Each service is not aware of the overall workflow.
- **The orchestrator pattern has some drawbacks** because of the tight coupling between the orchestrator and other services. To execute tasks in a sequence, the orchestrator needs to have some domain knowledge about the responsibilities of those services. If you want to add or remove services, existing logic will break, and you'll need to rewire portions of the communication path. Such an implementation is complex and hard to maintain.

Context and problem

In microservices architecture, it's often the case that a cloud-based application is divided into several small services that work together to process a business transaction end-to-end. To lower coupling between services, each service is responsible for a single business operation. Some benefits include faster development, smaller code base, and scalability. However, designing an efficient and scalable workflow is a challenge and often requires complex interservice communication.

The services communicate with each other by using well-defined APIs. Even a single business operation can result in multiple point-to-point calls among all services. A common pattern for communication is to use a centralized service that acts as the orchestrator. It acknowledges all incoming requests and delegates operations to the respective services. In doing so, it also manages the workflow of the entire business transaction. Each service just completes an operation and is not aware of the overall workflow.

The orchestrator pattern reduces point-to-point communication between services but has some drawbacks because of the tight coupling between the orchestrator and other services that participate in processing of the business transaction. To execute tasks in a sequence, the orchestrator needs to have some domain knowledge about the responsibilities of those services. If you want to add or remove services, existing logic will break, and you'll need to rewire portions of the communication path. While you can configure the workflow, add or remove services easily with a well-designed orchestrator, such an implementation is

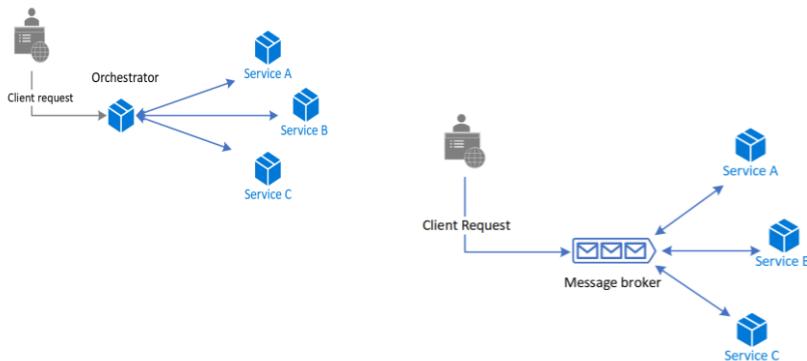
complex and hard to maintain.



Choreography pattern

Let each service decide when and how a business operation is processed, instead of depending on a central orchestrator.

One way to implement choreography is to use the [asynchronous messaging pattern](#) to coordinate the business operations.



Choreography pattern

- A client request publishes messages to a message queue. As messages arrive, they are pushed to subscribers, or services, interested in that message. Each subscribed service does their operation as indicated by the message and responds to the message queue with success or failure of the operation. In case of success, the service can push a message back to the same queue or a different message queue so that another service can continue the workflow if needed. If an operation fails, the message bus can retry that operation.
- This way, services choreograph the workflow among themselves without depending on an orchestrator or having direct communication between them.
- Because there is not point-to-point communication, this pattern helps reduce coupling between services. Also, it can remove the performance bottleneck caused by the orchestrator when it has to deal with all transactions.



Choreography pattern

When to use this pattern

- Use the choreography pattern if you expect to update, remove, or add new services frequently. The entire app can be modified with lesser effort and minimal disruption to existing services.
- Consider this pattern if you experience performance bottlenecks in the central orchestrator.
- This pattern is a natural model for the **serverless architecture** where all services can be short lived, or event driven. Services can spin up because of an event, do their task, and are removed when the task is finished.



Choreography pattern

Issues and considerations

- The **workflow can become complicated** when choreography needs to occur in a sequence. For instance, Service C can start its operation only after Service A and Service B have completed their operations with success. One approach is to have multiple message buses that get messages in the required order.
- The choreography pattern becomes **a challenge if the number of services grow rapidly**.
- For choreography, the resiliency management role is distributed between all services and **resiliency becomes less robust**.
- Each service isn't only responsible for the resiliency of its operation but also the workflow. This **responsibility can be burdensome for the service and hard to implement**.

Issues and considerations

Decentralizing the orchestrator can cause issues while managing the workflow.

If a service fails to complete a business operation, it can be difficult to recover from that failure. One way is to have the service indicate failure by firing an event. Another service subscribes to those failed events takes necessary actions such as applying [compensating transactions](#) to undo successful operations in a request. The failed service might also fail to fire an event for the failure. In that case, consider using a retry and, or time out mechanism to recognize that operation as a failure. For an example, see the [Example](#) section.

It's simple to implement a workflow when you want to process independent business operations in parallel. You can use a single message bus. However, the workflow can become complicated when choreography needs to occur in a sequence. For instance, Service C can start its operation only after Service A and Service B have completed their operations with success. One approach is to have multiple message buses that get messages in the required order. For more information, see the [Example](#) section.

The choreography pattern becomes a challenge if the number of services grow rapidly. Given the high number of independent moving parts, the workflow between services tends to get complex. Also, distributed tracing becomes difficult.

The orchestrator centrally manages the resiliency of the workflow and it can become a single point of failure. On the other hand, for choreography, the role is distributed between all services and resiliency becomes less robust.

Each service isn't only responsible for the resiliency of its operation but also the workflow. This responsibility can be burdensome for the service and hard to implement. Each service must retry transient, nontransient, and time-out failures, so that the request terminates gracefully, if needed. Also, the service must be diligent about communicating the success or failure of the operation so that other services can act accordingly.

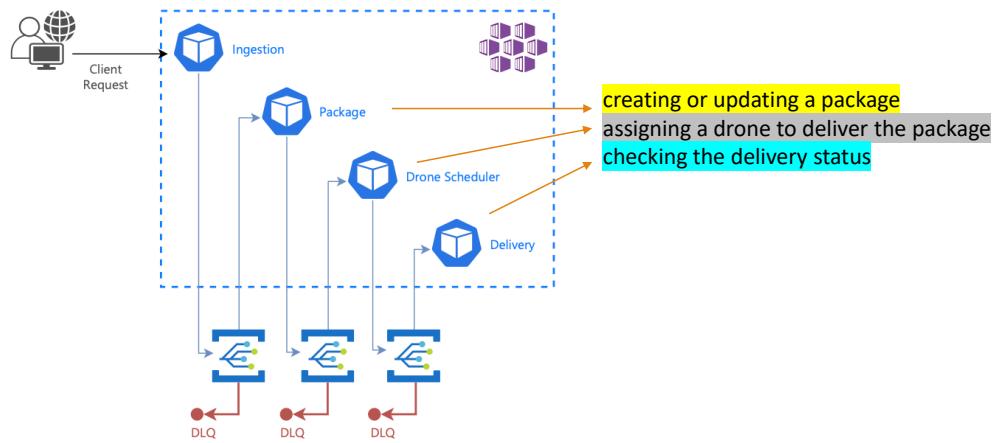


no

Choreography pattern

Example

This example shows the choreography pattern with the [Drone Delivery app](#). When a client requests a pickup, the app assigns a drone and notifies the client.



A single client business transaction requires three distinct business operations: creating or updating a package, assigning a drone to deliver the package, and checking the delivery status. Those operations are performed by three microservices: Package, Drone Scheduler, and Delivery services. Instead of a central orchestrator, the services use messaging to collaborate and coordinate the request among themselves.

The dead-letter queue (or undelivered-message queue) is the queue to which messages are sent if they cannot be routed to their correct destination. Each queue manager typically has a dead-letter queue.

A dead-letter queue (DLQ), sometimes referred to as an undelivered-message queue, is a holding queue for messages that cannot be delivered to their destination queues, for example because the queue does not exist, or because it is full. Dead-letter queues are also used at the sending end of a channel, for data-conversion errors.. Every queue manager in a network typically has a local queue to be used as a dead-letter queue so that messages that cannot be delivered to their correct destination can be stored for later retrieval.



Publisher-Subscriber pattern

Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.

Also called: Pub/sub messaging

Context and problem

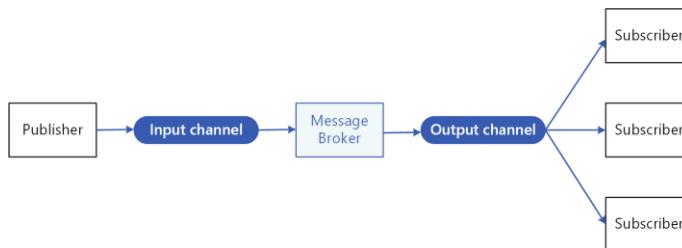
- In cloud-based and distributed applications, components of the system often need to provide information to other components as events happen.
- Asynchronous messaging is an effective way to decouple senders from consumers, and avoid blocking the sender to wait for a response. However, using a dedicated message queue for each consumer does not effectively scale to many consumers. Also, some of the consumers might be interested in only a subset of the information. How can the sender announce events to all interested consumers without knowing their identities.



Publisher-Subscriber pattern

Introduce an asynchronous messaging subsystem that includes the following:

- An input messaging channel used by the sender. The sender packages events into messages, using a known message format, and sends these messages via the input channel. The sender in this pattern is also called the *publisher*.
- One output messaging channel per consumer. The consumers are known as *subscribers*.
- A mechanism for copying each message from the input channel to the output channels for all subscribers interested in that message. This operation is typically handled by an intermediary such as a **message broker** or **event bus**.



Publisher-Subscriber pattern

pub/sub messaging has the following benefits:

- It decouples subsystems that still need to communicate.
- It increases scalability and improves responsiveness of the sender.
- It improves reliability.
- It allows for deferred or scheduled processing.
- It enables simpler integration between systems using different platforms, programming languages, or communication protocols, as well as between on-premises systems and applications running in the cloud.
- It facilitates asynchronous workflows across an enterprise.
- It improves testability. Channels can be monitored and messages can be inspected or logged as part of an overall integration test strategy.
- It provides separation of concerns for your applications.

Pub/sub messaging has the following benefits:

- It decouples subsystems that still need to communicate. Subsystems can be managed independently, and messages can be properly managed even if one or more receivers are offline.
- It increases scalability and improves responsiveness of the sender. The sender can quickly send a single message to the input channel, then return to its core processing responsibilities. The messaging infrastructure is responsible for ensuring messages are delivered to interested subscribers.
- It improves reliability. Asynchronous messaging helps applications continue to run smoothly under increased loads and handle intermittent failures more effectively.
- It allows for deferred or scheduled processing. Subscribers can wait to pick up messages until off-peak hours, or messages can be routed or processed according to a specific schedule.
- It enables simpler integration between systems using different platforms, programming languages, or communication protocols, as well as between on-premises systems and applications running in the cloud.
- It facilitates asynchronous workflows across an enterprise.
- It improves testability. Channels can be monitored and messages can be inspected or logged as part of an overall integration test strategy.
- It provides separation of concerns for your applications. Each application can focus on its core capabilities, while the messaging infrastructure handles

everything required to reliably route messages to multiple consumers.



Publisher-Subscriber pattern

Issues and considerations

- **Use existing technologies.** It is strongly recommended to use available messaging products rather than building your own.
- **Subscription handling.** The messaging infrastructure must provide mechanisms that consumers can use to subscribe to or unsubscribe from available channels.
- **Security.** Connecting to any message channel must be restricted by security policy to prevent eavesdropping by unauthorized users or applications.
- **Subsets of messages.** Subscribers are usually only interested in subset of the messages distributed by a publisher.
- **Wildcard subscribers.** Consider allowing subscribers to subscribe to multiple topics via wildcards.
- **Bi-directional communication.** The channels in a publish-subscribe system are treated as unidirectional.
- **Message ordering.** The order in which consumer instances receive messages is not guaranteed,

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- **Existing technologies.** It is strongly recommended to use available messaging products and services that support a publish-subscribe model, rather than building your own. In Azure, consider using [Service Bus](#), [Event Hubs](#) or [Event Grid](#). Other technologies that can be used for pub/sub messaging include Redis, RabbitMQ, and Apache Kafka.
- **Subscription handling.** The messaging infrastructure must provide mechanisms that consumers can use to subscribe to or unsubscribe from available channels.
- **Security.** Connecting to any message channel must be restricted by security policy to prevent eavesdropping by unauthorized users or applications.
- **Subsets of messages.** Subscribers are usually only interested in subset of the messages distributed by a publisher. Messaging services often allow subscribers to narrow the set of messages received by:
 - **Topics.** Each topic has a dedicated output channel, and each consumer can subscribe to all relevant topics.
 - **Content filtering.** Messages are inspected and distributed based on the content of each message. Each subscriber can specify the content it is interested in.
- **Wildcard subscribers.** Consider allowing subscribers to subscribe to multiple topics via wildcards.

- **Bi-directional communication.** The channels in a publish-subscribe system are treated as unidirectional. If a specific subscriber needs to send acknowledgment or communicate status back to the publisher, consider using the [Request/Reply Pattern](#). This pattern uses one channel to send a message to the subscriber, and a separate reply channel for communicating back to the publisher.

- **Message ordering.** The order in which consumer instances receive messages isn't guaranteed, and doesn't necessarily reflect the order in which the messages were created. Design the system to ensure that message processing is idempotent to help eliminate any dependency on the order of message handling.



Publisher-Subscriber pattern

Issues and considerations

- **Message priority.** Some solutions may require that messages are processed in a specific order. The [Priority Queue pattern](#) provides a mechanism for ensuring specific messages are delivered before others.
- **Poison messages.** A malformed message, or a task that requires access to resources that aren't available, can cause a service instance to fail.
- **Repeated messages.** The same message might be sent more than once. For example, the sender might fail after posting a message. Then a new instance of the sender might start up and repeat the message.
- **Message expiration.** A message might have a limited lifetime. If it isn't processed within this period, it might no longer be relevant and should be discarded.
- **Message scheduling.** A message might be temporarily embargoed, that is the message should not be processed until a specific date and time. The message should not be available to a receiver until this time.

• **Message priority.** Some solutions may require that messages are processed in a specific order. The [Priority Queue pattern](#) provides a mechanism for ensuring specific messages are delivered before others.

• **Poison messages.** A malformed message, or a task that requires access to resources that aren't available, can cause a service instance to fail. The system should prevent such messages being returned to the queue. Instead, capture and store the details of these messages elsewhere so that they can be analyzed if necessary. Some message brokers, like Azure Service Bus, support this via their [dead-letter queue functionality](#).

• **Repeated messages.** The same message might be sent more than once. For example, the sender might fail after posting a message. Then a new instance of the sender might start up and repeat the message. The messaging infrastructure should implement duplicate message detection and removal (also known as deduping) based on message IDs in order to provide at-most-once delivery of messages. Alternatively, if using messaging infrastructure which doesn't deduplicate messages, make sure the [message processing logic is idempotent](#).

• **Message expiration.** A message might have a limited lifetime. If it isn't processed within this period, it might no longer be relevant and should be discarded. A sender can specify an expiration time as part of the data in the message. A receiver can examine this information before deciding whether to perform the business logic associated with the message.

• **Message scheduling.** A message might be temporarily embargoed and should not be processed until a specific date and time. The message should not be

available to a receiver until this time.

Publisher-Subscriber pattern

Use this pattern when:

- An application needs to broadcast information to a significant number of consumers.
- An application needs to communicate with one or more independently-developed applications or services, which may use different platforms, programming languages, and communication protocols.
- An application can send information to consumers without requiring real-time responses from the consumers.
- The systems being integrated are designed to support an eventual consistency model for their data.
- An application needs to communicate information to multiple consumers, which may have different availability requirements or uptime schedules than the sender.

This pattern might not be useful when:

- An application has only a few consumers who need significantly different information from the producing application.
- An application requires near real-time interaction with consumers.



Priority Queue pattern

Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority. This pattern is useful in applications that offer different service level guarantees to individual clients.

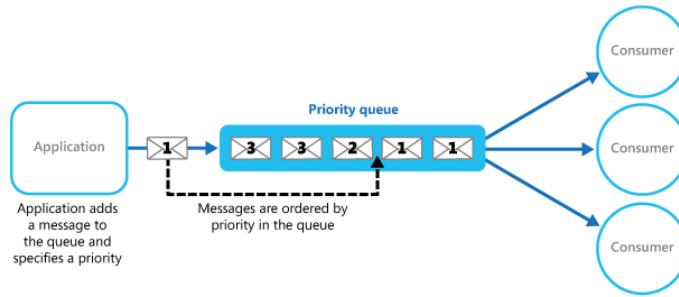
Context and problem

Applications can delegate specific tasks to other services, for example, to perform background processing or to integrate with other applications or services. In the cloud, a message queue is typically used to delegate tasks to background processing. In many cases, the order requests are received in by a service is not important. In some cases, though, it is necessary to prioritize specific requests. These requests should be processed earlier than lower priority requests that were sent previously by the application.



Priority Queue pattern

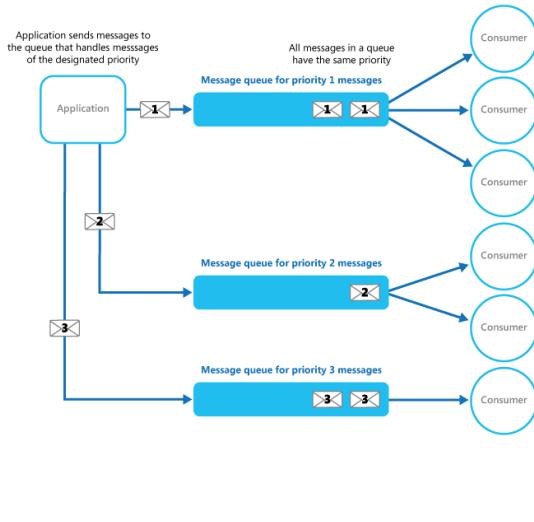
A queue is usually a first-in, first-out (FIFO) structure, and consumers typically receive messages in the same order that they were posted to the queue. However, some message queues support priority messaging. The application posting a message can assign a priority and the messages in the queue are automatically reordered so that those with a higher priority will be received before those with a lower priority. The figure illustrates a queue with priority messaging.





Priority Queue pattern

In systems that don't support priority-based message queues, an alternative solution is to maintain a separate queue for each priority. The application is responsible for posting messages to the appropriate queue. Each queue can have a **separate pool of consumers**. Higher priority queues can have a larger pool of consumers running on faster hardware than lower priority queues. The next figure illustrates using separate message queues for each priority.



A variation on this strategy is to have a single pool of consumers that check for messages on high priority queues first, and only then start to fetch messages from lower priority queues. There are some semantic differences between a solution that uses a single pool of consumer processes (either with a single queue that supports messages with different priorities or with multiple queues that each handle messages of a single priority), and a solution that uses multiple queues with a separate pool for each queue.

In the single pool approach, higher priority messages are always received and processed before lower priority messages. In theory, messages that have a very low priority could be continually superseded and might never be processed. In the multiple pool approach, lower priority messages will always be processed, just not as quickly as those of a higher priority (depending on the relative size of the pools and the resources that they have available).



Priority Queue pattern

Using a priority-queuing mechanism can provide the following advantages:

- It allows applications to **meet business requirements** that require prioritization of availability or performance, such as offering different levels of service to specific groups of customers.
- It can help to **minimize operational costs**. In the single queue approach, you can scale back the number of consumers if necessary. High priority messages will still be processed first (although possibly more slowly), and lower priority messages might be delayed for longer. If you've implemented the multiple message queue approach with separate pools of consumers for each queue, you can reduce the pool of consumers for lower priority queues, or even suspend processing for some very low priority queues by stopping all the consumers that listen for messages on those queues.
- The **multiple message queue** approach can **help maximize application performance** and scalability by partitioning messages based on processing requirements. For example, vital tasks can be prioritized to be handled by receivers that run immediately while less important background tasks can be handled by receivers that are scheduled to run at less busy periods.



Priority Queue pattern

Issues and considerations

Define the priorities in the context of the solution. For example, high priority could mean that messages should be processed within ten seconds.

Decide if **all** high priority items must be processed before any lower priority items.

Monitor the processing speed on high and low priority queues to ensure that messages in these queues are processed at the expected rates.

If you need to guarantee that **low priority messages will be processed**

Using a separate queue for each message priority works best for systems that have a few well-defined priorities.

Message priorities can be determined logically by the system. For example, rather than having explicit high and low priority messages, they could be designated as "fee-paying customer," or "non-fee paying customer."

There might be a **financial and processing cost** associated with checking a queue. This cost increases when checking multiple queues.

It's possible to **dynamically adjust the size of a pool** of consumers based on the length of the queue that the pool is servicing.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

Define the priorities in the context of the solution. For example, high priority could mean that messages should be processed within ten seconds. Identify the requirements for handling high priority items, and the other resources that should be allocated to meet these criteria.

Decide if all high priority items must be processed before any lower priority items. If the messages are being processed by a single pool of consumers, you have to provide a mechanism that can preempt and suspend a task that's handling a low priority message if a higher priority message becomes available.

In the multiple queue approach, when using a single pool of consumer processes that listen on all queues rather than a dedicated consumer pool for each queue, the consumer must apply an algorithm that ensures it always services messages from higher priority queues before those from lower priority queues.

Monitor the processing speed on high and low priority queues to ensure that messages in these queues are processed at the expected rates.

If you need to guarantee that low priority messages will be processed, it's necessary to implement the multiple message queue approach with multiple pools of consumers. Alternatively, in a queue that supports message prioritization, it's possible to dynamically increase the priority of a queued message as it ages. However, this approach depends on the message queue providing this feature.

Using a separate queue for each message priority works best for systems that have a few well-defined priorities.

Message priorities can be determined logically by the system. For example, rather than having explicit high and low priority messages, they could be designated as "fee-paying customer," or "non-fee paying customer." Depending on your business model, your system can allocate more resources to processing messages from fee-paying customers than non-fee paying ones.

There might be a financial and processing cost associated with checking a queue for a message (some commercial messaging systems charge a small fee each time a message is posted or retrieved, and each time a queue is queried for messages). This cost increases when checking multiple queues.

It's possible to dynamically adjust the size of a pool of consumers based on the length of the queue that the pool is servicing. For more information, see the [Autoscaling Guidance](#).

Software Defined Networking and Network Function Virtualization

Gianluca Reali

Outline

- SDN motivations: Internet ossification, network complexity, barriers to innovation
- SDN approach
- OpenFlow
- Application examples
- SDN and cloud
- SDN and Network Function Virtualization

Internet success

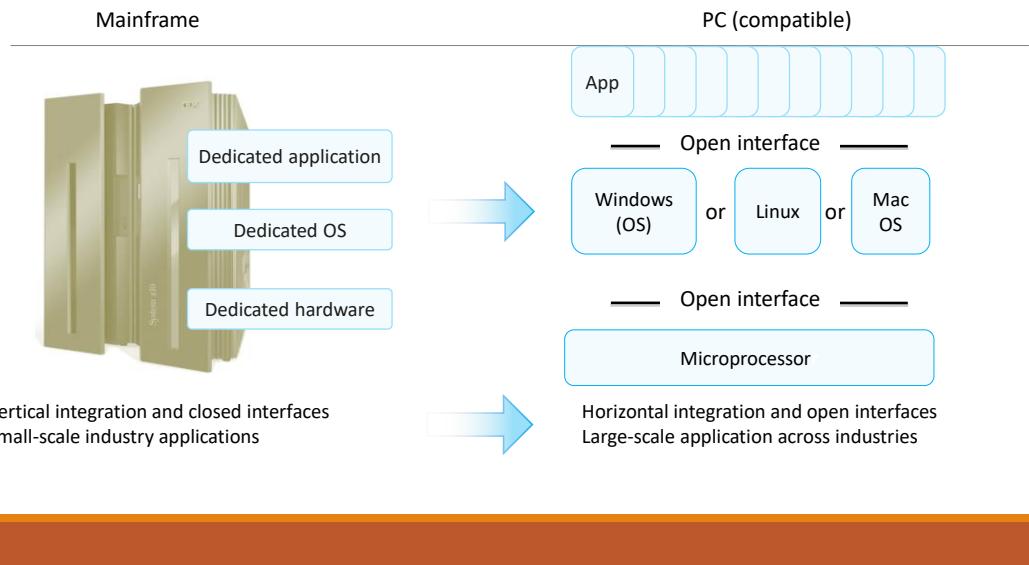
- The Internet success is a remarkable story, from a research infrastructure to a global network, interconnecting billions of devices and people
- Innovation looks easy on the Internet as we witness always new and more powerful services and applications
 - Web, P2P, VoIP, IoT, Mobile Apps, social networks, video streaming...
 - Extensive use of Datacentres and Clouds

Network ossification

The history is a bit different behind the scene:

- Huge complexity
- Few people can innovate
- Closed equipment
- Network «ossification»

Evolution of the Computer Era



In 1964, IBM spent US\$5 billion on developing IBM System/360 (S/360), which started the history of mainframes. Mainframes typically use the centralized architecture. The architecture features excellent I/O processing capability and is the most suitable for processing large-scale transaction data. Compared with PCs, mainframes have dedicated hardware, operating systems, and applications.

PCs have undergone multiple innovations from hardware, operating systems, to applications. Every innovation has brought about great changes and development. The following three factors support rapid innovation of the entire PC ecosystem:

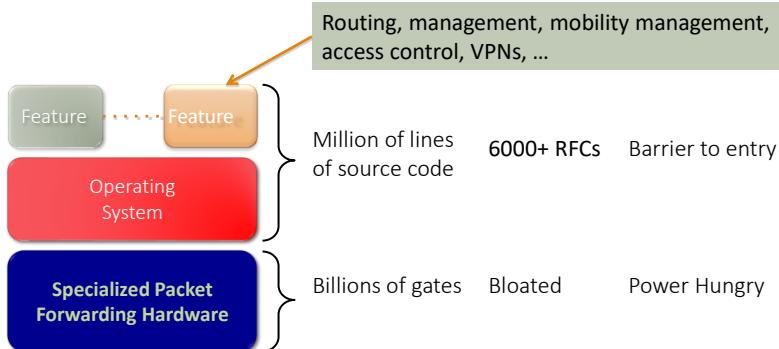
Hardware substrate: The PC industry has adapted a simple and universal hardware base, x86 instruction set.

Software-defined: The upper-layer applications and lower-layer basic software (OS and virtualization) are greatly innovated.

Open-source: The flourishing development of Linux has verified the correctness of open source and bazaar model. Thousands of developers can quickly formulate standards to accelerate innovation.



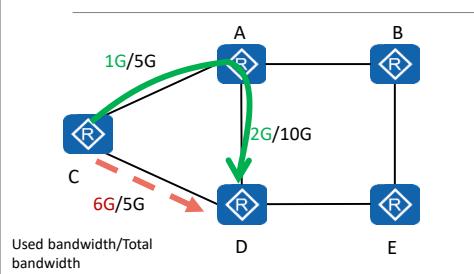
The Ossified Network



- Many complex functions baked into the infrastructure
OSPF, BGP, multicast, differentiated services, Traffic Engineering, NAT, firewalls, MPLS, redundant layers, ...
- An industry with a “mainframe-mentality”, reluctant to change

Frequent Network Congestion

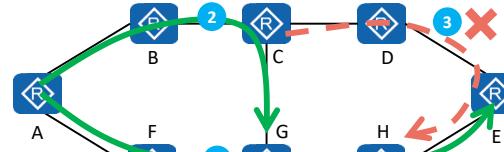
Problem and Solution of Bandwidth-based Route Selection



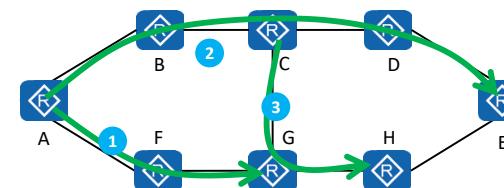
The network computes forwarding paths based on bandwidth. The link from router C to router D is the shortest forwarding path. The volume of service traffic from router C to router D exceeds the bandwidth, causing packet loss. Although other links are idle, the algorithm still selects the shortest path for forwarding. The optimal traffic forwarding path is C-A-D.

Problem and Solution of Tunnel Establishment Based on Fixed Sequence

Tunnels are established in sequence: 1. A-E; 2. A-G; 3. C-H. Tunnel 3 fails to be established due to insufficient bandwidth.

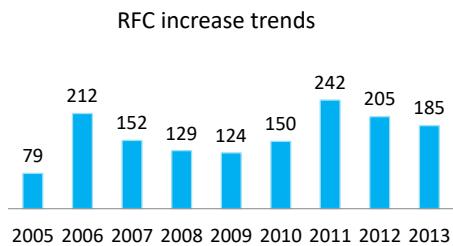


Global path calculation and optimal tunnel path adjustment:



Complex Network Technologies

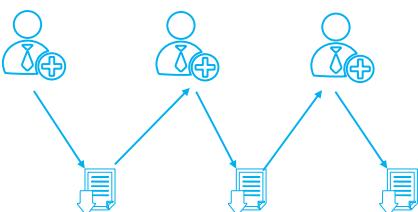
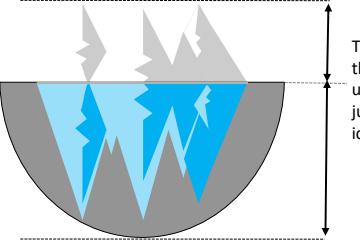
Many network protocols: Network technology experts need to learn many RFCs related to network devices. Understanding the RFCs takes a long time, and the number of RFCs is still increasing.



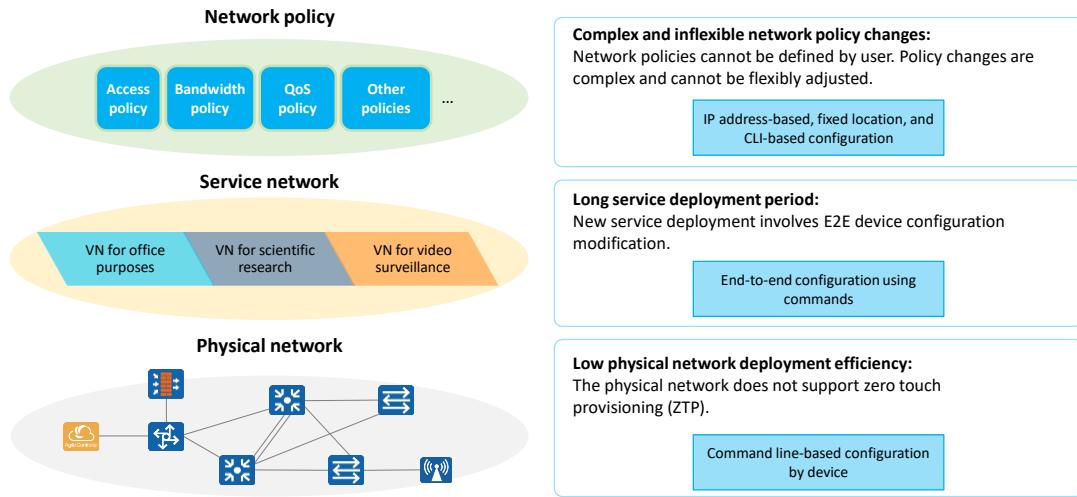
Difficult network configuration: To be familiar with devices of a specific vendor, you need to master tens of thousands of commands. Additionally, the number of commands is still increasing.



Difficulty in Locating and Analyzing Network Faults

Difficult to Spot Faults	Difficult to Locate Faults
<p>Manual fault identification</p> <p>Manual packet obtaining for locating faults</p> <p>Manual fault diagnosis</p> 	<p>Abnormal flows account for 3.65% of all flows on the network.</p>  <p>The network faults that are found upon user complaints are just the tip of the iceberg.</p> <ul style="list-style-type: none">Traditional O&M networks rely on manual fault identification, location, and diagnosis.More than 85% of network faults are found only after service complaints. Problems cannot be proactively identified or analyzed. <ul style="list-style-type: none">Traditional O&M only monitors device indicators. Some indicators are normal, but user experience is poor. There is no correlated analysis of users and networks.According to data center network (DCN) statistics, it takes 76 minutes to locate a fault on average.

Slow Network Service Deployment



Vision of network service deployment:

- Free mobility based on network policies, regardless of physical locations
- Quick deployment of new service
- ZTP deployment on the physical network
- Plug-and-play of devices

Zero-touch provisioning (ZTP) is a method of setting up devices that automatically configures the device using a switch feature. ZTP helps IT teams quickly deploy network devices in a large-scale environment, eliminating most of the manual labor involved with adding them to a network.

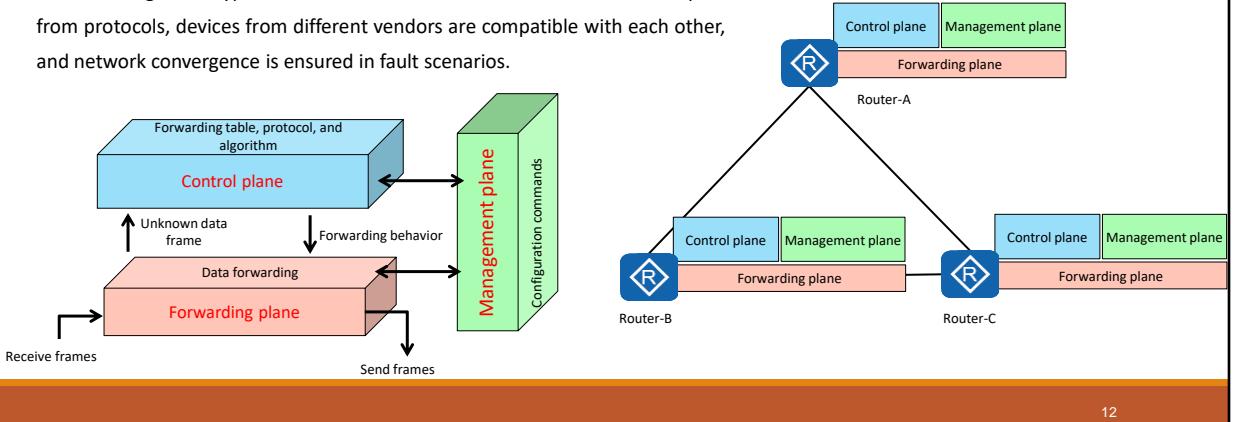
Outline

- SDN motivations: Internet ossification, network complexity, barriers to innovation
- SDN approach
- OpenFlow
- Application examples
- SDN and cloud
- SDN and Network Function Virtualization

Current Situation of the Network Industry: Typical IP Network - Distributed Network

The typical IP network is a distributed network with peer-to-peer control. Each network device has independent forwarding, control, and management planes. The control plane of a network device exchanges packets of a routing protocol to generate an independent data plane to guide packet forwarding.

- The advantage of a typical IP network is that network devices are decoupled from protocols, devices from different vendors are compatible with each other, and network convergence is ensured in fault scenarios.



12

The switch is used as an example to describe the forwarding plane, control plane, and management plane.

Forwarding plane: provides high-speed, non-blocking data channels for service switching between service modules. The basic task of a switch is to process and forward various types of data on its interfaces. Specific data processing and forwarding, such as Layer 2, Layer 3, ACL, QoS, multicast, and security protection, occur on the forwarding plane.

Control plane: provides functions such as protocol processing, service processing, route calculation, forwarding control, service scheduling, traffic statistics collection, and system security. The control plane of a switch is used to control and manage the running of all network protocols. The control plane provides various network information and forwarding query entries required for data processing and forwarding on the data plane.

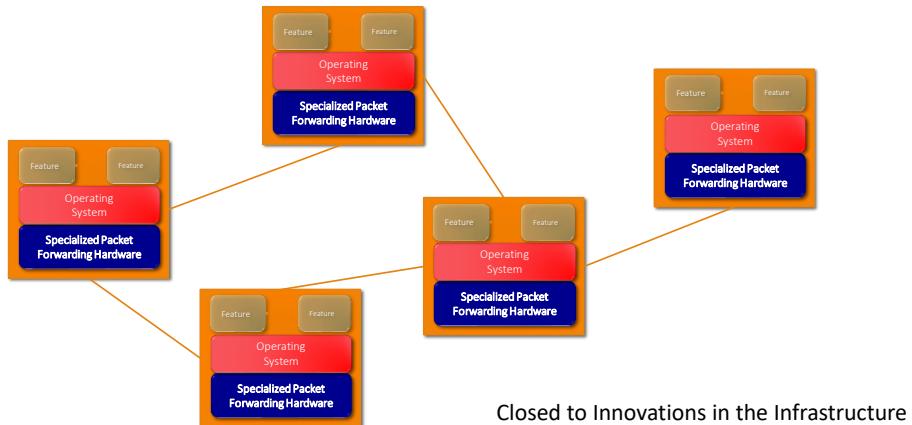
Management plane: provides functions such as system monitoring, environment monitoring, log and alarm processing, system software loading, and system upgrade. The management plane of a switch provides network management personnel with Telnet, web, SSH, SNMP, and RMON to manage devices, and supports, parses, and executes the commands for setting network protocols. On the management plane, parameters related to various protocols on the control plane must be pre-configured, and the running of the control plane can be intervened if necessary.

Some Huawei series products are divided into the data plane, management plane, and monitoring plane.

Classical network architecture

Distributed control plane

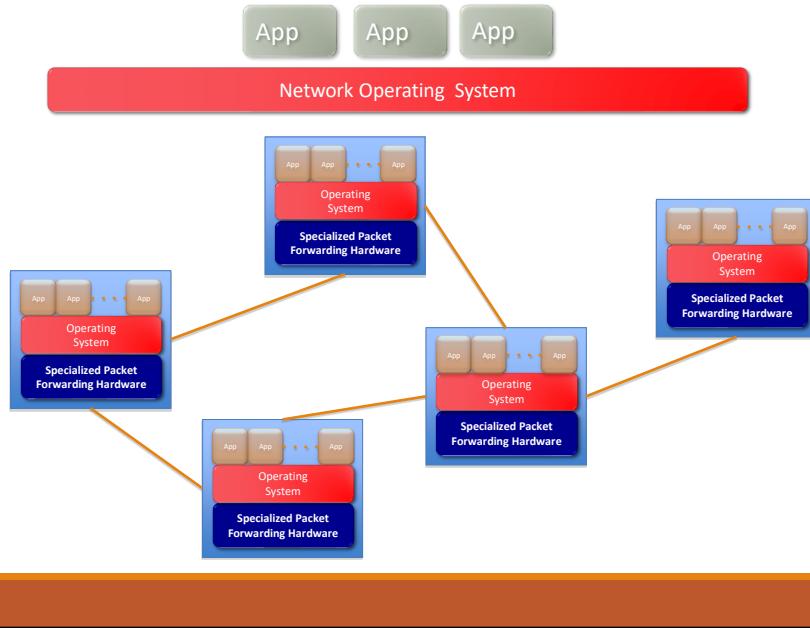
Distributed routing protocols: OSPF, IS-IS, BGP, etc.



13

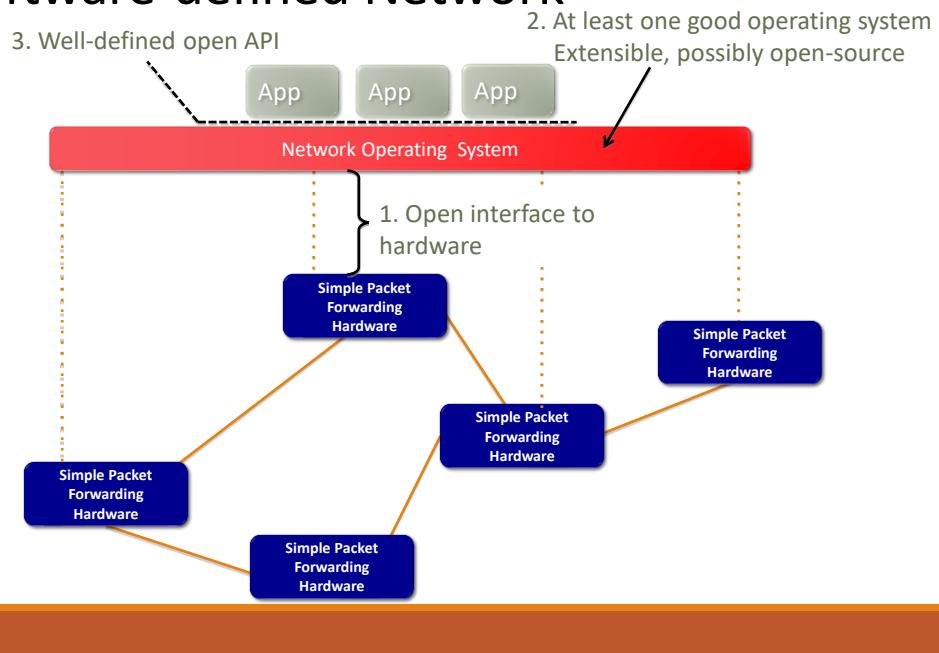
Classical computer network architecture

“Software Defined Networking” approach to open it



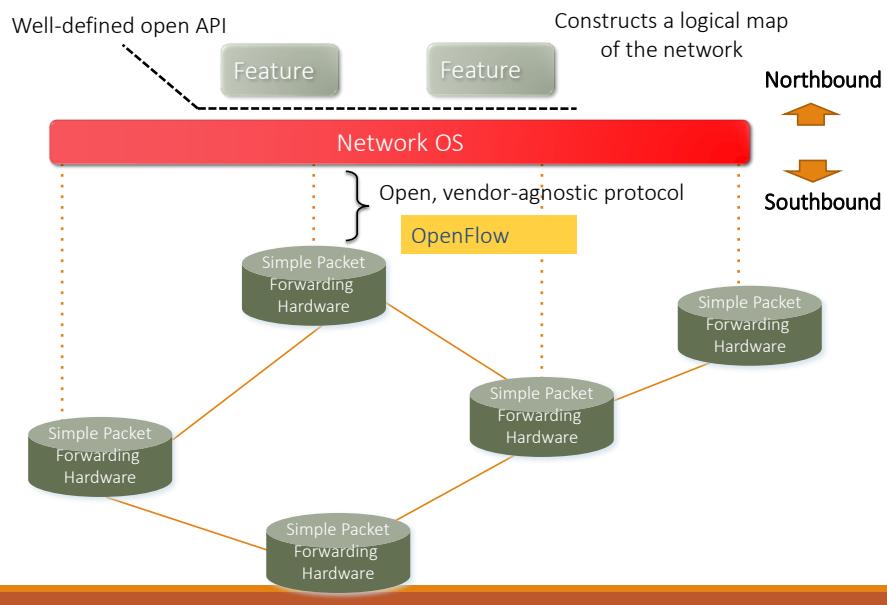
- How do we redefine the architecture to open up networking infrastructure and the industry!
- By bringing to the networking industry what we did to the computing world

The “Software-defined Network”

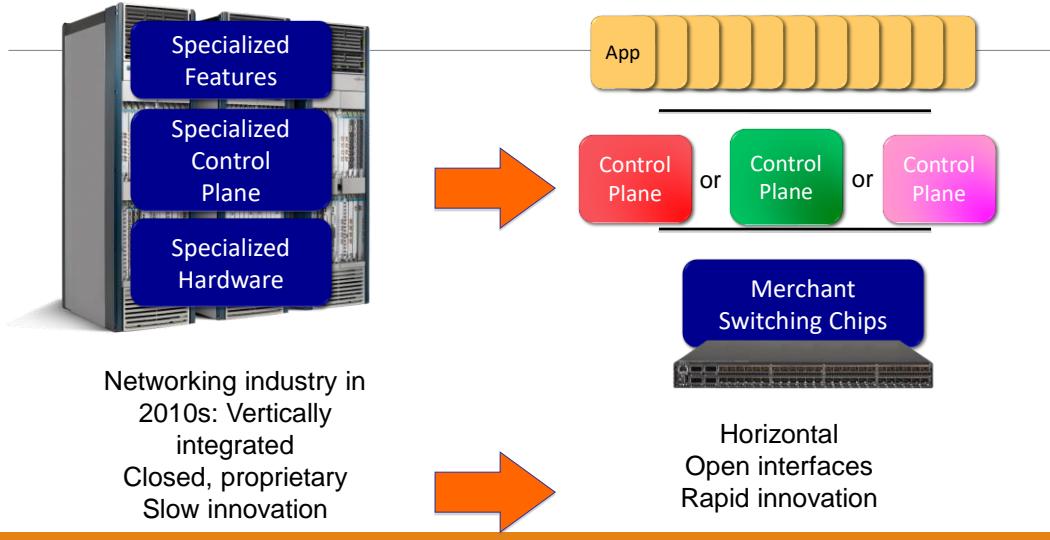


- Switches, routers and other middleboxes are dumbed down
- The key is to have a standardized control interface that speaks directly to hardware

Software Defined Network

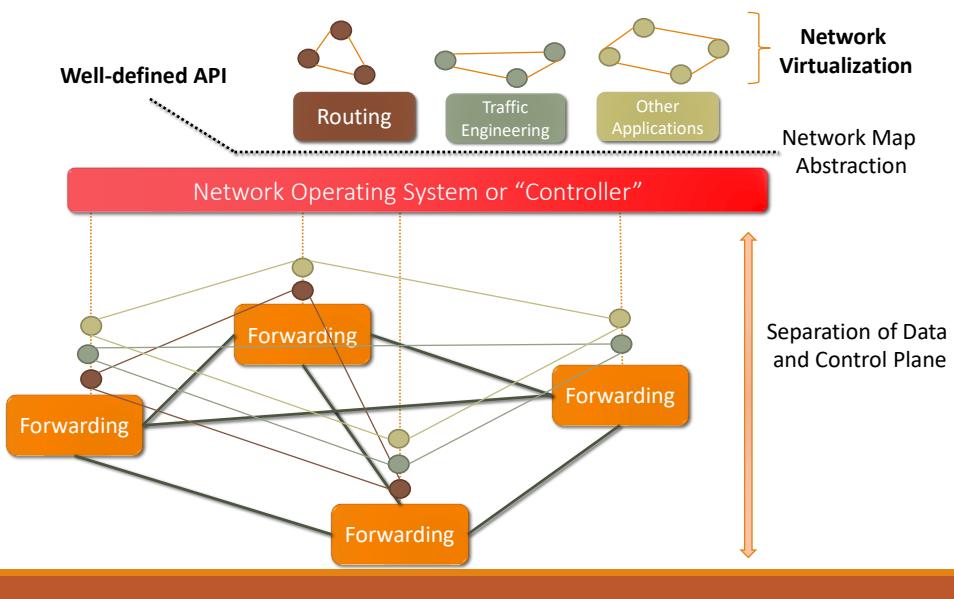


Analogy with IT industry: from closed box to SDN



17

Abstractions in the Control Plane



18

SDN Concept

- Separate Control plane and Data plane entities
 - Network intelligence and state are logically centralized
 - The underlying network infrastructure is abstracted from the applications
- Execute or run Control plane software on general purpose hardware
 - Decouple from specific networking hardware
 - Use commodity servers
- Have programmable data planes
 - Maintain, control and program data plane state from a central entity

An architecture to control, not just a networking device but an entire network

Network OS and OpenFlow

- **Network OS**

Distributed system that creates a consistent, up-to-date network view, runs on servers (controllers) in the network

- Uses an open protocol to:
 - Get state information **from** forwarding elements
 - Give control directives **to** forwarding elements

- **OpenFlow**

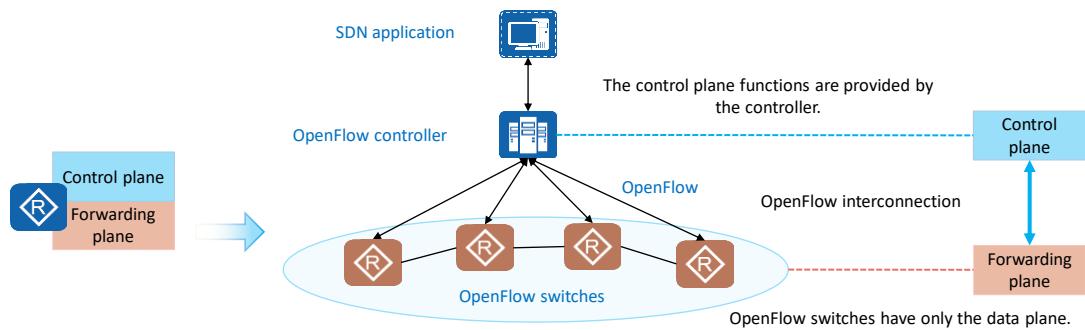
- is a protocol for remotely controlling the forwarding table of a switch or router
- is one element of SDN

Outline

- SDN motivations: Internet ossification, network complexity, barriers to innovation
- SDN approach
- OpenFlow
- Application examples
- SDN and cloud
- SDN and Network Function Virtualization

Network OS and OpenFlow

- SDN was developed by the Clean Slate Program at Stanford University as an innovative new network architecture. The core of SDN is to separate the control plane from the data plane of network devices to implement centralized control of the network control plane and provide good support for network application innovation.
- SDN has three characteristics in initial phase: **forwarding-control separation**, **centralized control**, and **open programmable interfaces**.



OpenFlow History

OpenFlow

- 2007, Stanford Univ.
- 2008, OpenFlow Consortium
- 2008, [Nicira Networks](#) released NOX platform.
- 2009, OpenFlow Spec 1.0
- 2009 MIT Tech. Review → SDN as one of 10 emerging technologies
- 2011 March, ONF ([Open Networking Foundation](#)) was born
- Facebook, Google, Microsoft, Yahoo → Data Center Operators
- Expand OpenFlow technologies to SDN
- 2012 ONF released OpenFlow 1.3
- 2013 ONF released OpenFlow 1.4
- 2015, ONF released OpenFlow 1.5
- 2017, extensions for optical transport and SPTN

SPTN: Super PTN - Packet Transport Network: MPLS-TP (Multi-Protocol Label Switching – Transport Profile) OpenFlow Protocol Extensions for SPTN

Examples of SDN Products and Solutions

- Open Source

	Solutions	OpenFlow version	
Controller	NOX	Support OpenFlow 1.3	C++ API
	POX	Python version of NOX, Support OpenFlow 1.1	Python API
	Floodlight	Support OpenFlow 1.3	BigSwitch joined OpenDaylight but left it on June 2013
	Ryu	Support OpenFlow 1.4	Python API
	OpenDayLight (ODL)	Support OpenFlow 1.3	2014.2
Switch	Open vSwitch	Support OpenFlow 1.3	
	Ericsson soft switch	Support OpenFlow 1.3	Compatible with Mininet Controller: NOX 1.3

Vendors

- NEC: released OpenFlow 1.3 switch and controller... 2013.9
- HP: released OpenFlow 1.3 data center switch ... 2013
- Centec Network, China: released Open SDN switch with OpenFlow1.3 support (implemented on OpenVswitch) ... 2013.4
- Brocade, OpenFlow 1.3 switch ... 2014.6~

Basic Concepts of OpenFlow

OpenFlow Controller



OpenFlow Protocol (SSL/TCP)

Control Path

OpenFlow

Data Path (Hardware)

OpenFlow

- **Definition**

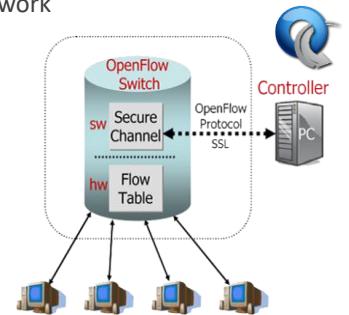
- A communication protocol that gives access to the forwarding plane of the network switch or router

- **Components**

- OpenFlow controller
 - Process packet match, instruction & action set, pipeline processing
- OpenFlow switch
 - Secure channel, flow table

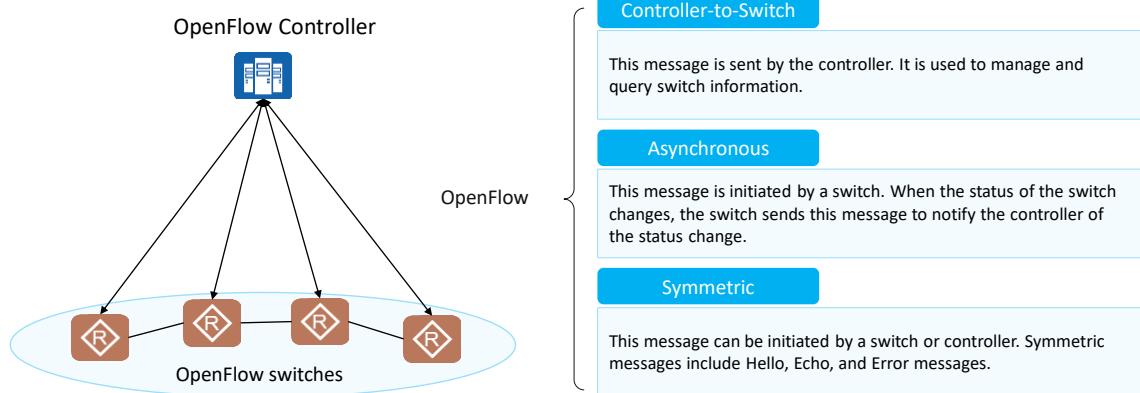
- **Features**

- Separation of control plane and data plane
 - The data path of an OpenFlow switch consists of a **Flow Table**, including **actions** associated with each flow entry
 - The control path consists of a controller which programs flow entry in the flow table
- OpenFlow is based on an Ethernet switch, with an internal flow-table, and a standardized interface to add and remove flow entries



Basic Concepts of OpenFlow

OpenFlow is an SBI protocol between a controller and a switch. It defines three types of messages: **Controller-to-Switch**, **Asynchronous**, and **Symmetric**. Each message contains more subtypes.



OpenFlow Protocol Messages

Protocol Layer

- OpenFlow control message relies on TCP protocol
- Controllers listen on [TCP port 6633/6653](#) to setup connection with switch
 - 6633/6653 became the official [IANA](#) port since 2013-07-18
- OpenFlow message structure
 - Version
 - Indicates the version of OpenFlow which this message belongs
 - Type
 - Indicates what type of message is present and how to interpret the payload (version dependent)
 - Message length
 - Indicates where this message will be end, starting from the first byte of header
 - Transaction ID (xid)
 - A unique value used to match requests to response

OpenFlow Message Structure

Bit Offset	0 ~ 7	8 ~ 15	16 ~ 23	24 ~ 31
0 ~ 31	Version	Type		Message Length
32 ~ 63			Transaction ID	
64 ~ ?			Payload	

6633 is historical, and 6653 is the newly allocated official IANA port

OpenFlow Protocol Messages - Examples

C: OpenFlow Controller AM: Asynchronous message CSM: Control/Switch Message
 S: OpenFlow Switch SM: Symmetric Message

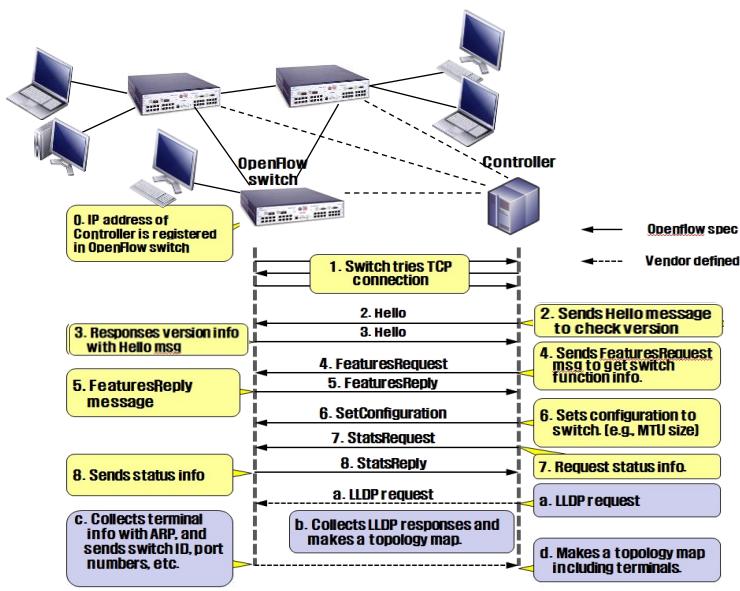
Category	Message	Type	Description
Meta Info. Configuration	Hello (SM)	C → S	following a TCP handshake, the controller sends its version number to the switch.
	Hello (SM)	S → C	the switch replies with its supported version number.
	Features Request (CSM)	C → S	the controller asks to see which ports are available.
	Set Config (CSM)	C → S	The controller sends a set config request message to set configuration parameters - e.g. Max bytes of new flow that datapath should send to the controller
	Features Reply (CSM)	S → C	the switch replies with a list of ports, port speeds, and supported tables and actions.
Flow Processing	Port Status	S → C	enables the switch to inform that controller of changes to port speeds or connectivity.
	Packet-In (AM)	S → C	a packet was received and it didn't match any entry in the switch's flow table, causing the packet to be sent to the controller.
	Packet-Out (CSM)	C → S	Instructs a switch to send a packet out to one or more switch ports.
	Flow-Mod (CSM)	C → S	instructs a switch to add a particular flow to its flow table.
	Flow-Expired (CSM)	S → C	a flow timed out after a period of inactivity.

enum ofp_type {/* Immutable messages. */OFPT_HELLO = 0, /* Symmetric message */OFPT_ERROR = 1, /* Symmetric message */OFPT_ECHO_REQUEST = 2, /* Symmetric message */OFPT_ECHO_REPLY = 3, /* Symmetric message */OFPT_EXPERIMENTER = 4, /* Symmetric message */63©2015; The Open Networking Foundation

OpenFlow Switch SpecificationVersion 1.5.1/* Switch configuration messages.
 /OFPT_FEATURES_REQUEST = 5, / Controller/switch message */OFPT_FEATURES_REPLY = 6, /* Controller/switch message */OFPT_GET_CONFIG_REQUEST = 7, /* Controller/switch message */OFPT_GET_CONFIG_REPLY = 8, /* Controller/switch message */OFPT_SET_CONFIG = 9, /* Controller/switch message *//* Asynchronous messages. */OFPT_PACKET_IN = 10, /* Async message */OFPT_FLOW_REMOVED = 11, /* Async message */OFPT_PORT_STATUS = 12, /* Async message *//* Controller command messages. */OFPT_PACKET_OUT = 13, /* Controller/switch message */OFPT_FLOW_MOD = 14, /* Controller/switch message */OFPT_GROUP_MOD = 15, /* Controller/switch message */OFPT_PORT_MOD = 16, /* Controller/switch message */OFPT_TABLE_MOD = 17, /* Controller/switch message *//* Multipart messages. */OFPT_MULTIPART_REQUEST = 18, /* Controller/switch message */OFPT_MULTIPART_REPLY = 19, /* Controller/switch message *//* Barrier messages. */OFPT_BARRIER_REQUEST = 20, /* Controller/switch message */OFPT_BARRIER_REPLY = 21, /* Controller/switch message *//* Controller role change request messages. */OFPT_ROLE_REQUEST = 24, /* Controller/switch message */OFPT_ROLE_REPLY = 25, /* Controller/switch message *//* Asynchronous message configuration. */OFPT_GET_ASYNC_REQUEST = 26, /* Controller/switch message */OFPT_GET_ASYNC_REPLY = 27, /* Controller/switch message */OFPT_SET_ASYNC = 28, /* Controller/switch message *//* Meters and rate limiters configuration messages. */OFPT_METER_MOD = 29, /* Controller/switch message *//* Controller role change event messages. */OFPT_ROLE_STATUS = 30, /* Async message *//* Asynchronous messages. */OFPT_TABLE_STATUS = 31, /* Async message *//* Request forwarding by the switch. */OFPT_REQUESTFORWARD = 32, /* Async message *//* Bundle operations (multiple messages as a single operation). */OFPT_BUNDLE_CONTROL = 33, /* Controller/switch message */OFPT_BUNDLE_ADD_MESSAGE = 34, /* Controller/switch message *//* Controller status async message. */OFPT_CONTROLLER_STATUS = 35, /* Async message */}

OpenFlow Communication

Connection Setup



The **Link Layer Discovery Protocol (LLDP)** is a vendor-neutral [link layer](#) protocol used by network devices for advertising their identity, capabilities, and neighbors on an [IEEE 802](#) local area network, principally [wired Ethernet](#).

Information gathered with LLDP can be stored in the device [management information database](#) (MIB) and queried with the [Simple Network Management Protocol](#) (SNMP) as specified in [RFC 2922](#). The topology of an LLDP-enabled network can be discovered by *crawling* the hosts and querying this database. Information that may be retrieved include:

System name and description

Port name and description

[VLAN](#) name

IP management address

System capabilities (switching, routing, etc.)

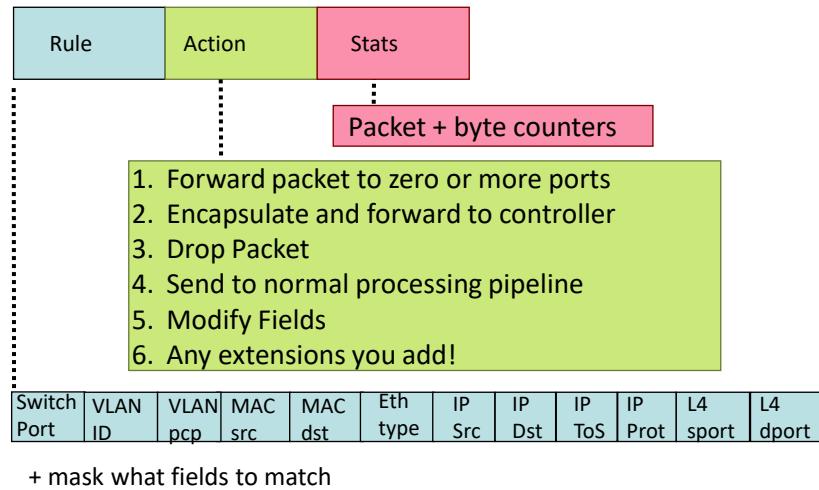
MAC/[PHY](#) information

[MDI power](#)

[Link aggregation](#)

OpenFlow Basics: Flow Table

OpenFlow switches forward packets based on flow tables.



31

Now I'll describe the API that tries to meet these goals.

OpenFlow Basics: Flow Table

Each flow entry includes the Match Fields, Priority, Counters, Instructions, Timeouts, Cookie, and Flags. The Match Fields and Instructions are key fields for packet forwarding.

The Match Fields is a field against which a packet is matched and can be customized.

The Instructions field indicates OpenFlow processing when a packet matches a flow entry.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags			
Flow table fields can be customized. The following table is an example.									
Ingress Port	Ether Source	Ether Dst	Ether Type	VLAN ID	VLAN Priority	IP Src	IP Dst	TCP Src Port	TCP Dst Port
3	MAC1	MAC2	0x8100	10	7	IP1	IP2	5321	8080

32

Match Fields: a field against which a packet is matched. (OpenFlow 1.5.1 supports 45 options). It can contain the inbound interface, inter-flow table data, Layer 2 packet header, Layer 3 packet header, and Layer 4 port number.

Priority: matching sequence of a flow entry. The flow entry with a higher priority is matched first.

Counters: number of packets and bytes that match a flow entry.

Instructions: OpenFlow processing when a packet matches a flow entry. When a packet matches a flow entry, an action defined in the Instructions field of each flow entry is executed. The Instructions field affects packets, action sets, and pipeline processing.

Timeouts: aging time of flow entries, including Idle Time and Hard Time.

Idle Time: If no packet matches a flow entry after Idle Time expires, the flow entry is deleted.

Hard Time: After Hard Time expires, a flow entry is deleted regardless of whether a packet matches the flow entry.

Cookie: identifier of a flow entry delivered by the controller.

Flags: This field changes the management mode of flow entries.

802.1Q VLAN tagging uses an *0x8100 EtherType* value

OpenFlow: Flow Table

match fields: to match against packets. These consist of the ingress port and packet headers, and optionally other pipeline fields such as metadata specified by a previous table.

priority: matching precedence of the flow entry.

counters: updated when packets are matched.

instructions: to modify the action set or pipeline processing.

timeouts: maximum amount of time or idle time before flow is expired by the switch.

cookie: opaque data value chosen by the controller. May be used by the controller to filter flow entries affected by flow statistics, flow modification and flow deletion requests. Not used when processing packets.

flags: flags alter the way flow entries are managed, for example the flag OFPFF_SEND_FLOW_Rem triggers flow removed messages for that flow entry.

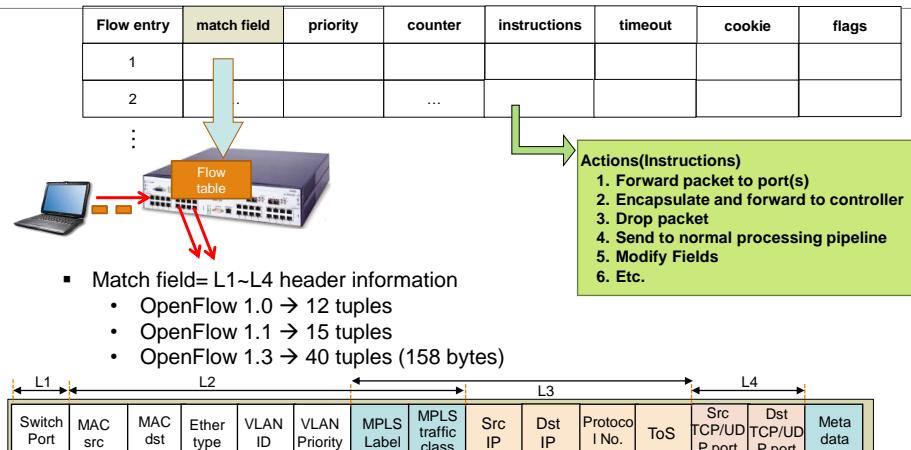
33

When the OFPFF_SEND_FLOW_Rem flag is set, the switch must send a flow removed message when the flow expires. The default is for the switch to not send flow removed messages for newly added flows.

When the OFPFF_CHECK_OVERLAP flag is set, the switch must check that there are no conflicting entries fails and an error code is returned.

When the OFPFF_EMERG_ flag is set, the switch must consider this flow entry as an emergency entry, and only use it for forwarding when disconnected from the controller with the same priority.

OpenFlow: Flow Table



OpenFlow: Flow Table

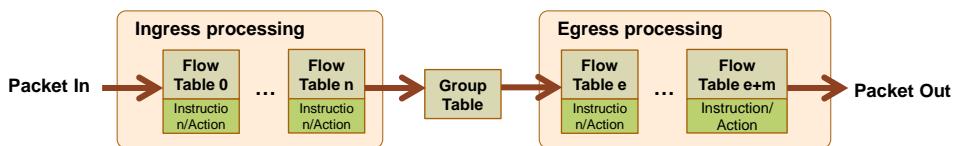
Operation Mode	Switch Port	MAC src	MAC dst	Ether type	VLAN ID	Src IP	Dst IP	Proto No.	TCP S_port	TCP D_port	Action	Counter
Switching	*	*	00:1f..	*	*	*	*	*	*	*	Port1	243
Flow Switching	Port3	00:20..	00:2f..	0800	vlan1	1.2.3.4	1.2.3.9	4	4666	80	Port7	123
Routing	*	*	*	*	*	*	1.2.3.4	*	*	*	Port6	452
VLAN Switching	*	*	00:3f..	*	vlan2	*	*	*	*	*	Port6 Port7 Port8	2341
Firewall	*	*	*	*	*	*	*	*	*	22	Drop	544
Default Route	*	*	*	*	*	*	*	*	*	*	Port1	1364

Wild card (*) means
“does not matter” – not
important field

OpenFlow Pipelining

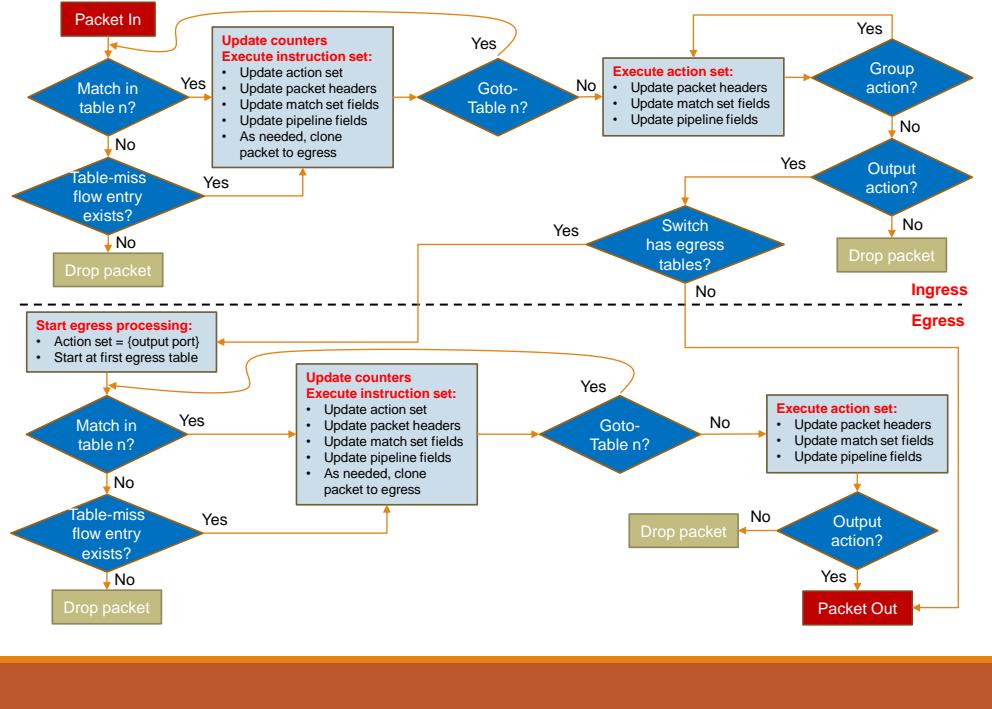
Pipelining

- The flow tables of a switch are sequentially numbered, starting at 0
- A packet is processed sequentially in multiple flow tables (version 1.1)
 - If a flow entry is found, the instruction set included in that flow entry is executed
 - Instructions may explicitly direct the packet to another flow table ("goto-table")
 - Pipeline processing can only go forward and not backward
- Two stage pipeline processing (version 1.5)
 - Ingress processing
 - Mandatory, performed before egress processing, use the rules specified in ingress tables
 - Egress processing
 - Optional, performed in the context of output port, use the rules specified in egress tables
 - Egress table can be configured during feature request/reply phase
- Useful to manage complicated processing
 - E.g., table 1 for VLAN processing, table 2 for multicast group processing



The OpenFlow pipeline of every OpenFlow Logical Switch contains one or more flow tables, each flow table containing multiple flow entries. The OpenFlow pipeline processing defines how packets interact with those flow tables (see Figure 2). An OpenFlow switch is required to have at least one ingress flow table, and can optionally have more flow tables. An OpenFlow switch with only a single flow table is invalid, in this case pipeline processing is greatly simplified. The flow tables of an OpenFlow switch are numbered in the order they can be traversed by packets starting at 0. Pipeline processing happens in two stages, ingress processing and egress processing. These separation of the two stages is indicated by the first egress table (see 7.3.2), all tables with a number lower than the first egress table must be used as ingress tables, and no table with a number higher than or equal to the first egress table can be used as an ingress table. Pipeline processing always starts with ingress processing at the first flow table: the packet must be first matched against flow entries of flow table 0 (see Figure 3). Other ingress flow tables may be used depending on the outcome of the match in the first table. If the outcome of ingress processing is to forward the packet to an output port, the OpenFlow switch may perform egress processing in the context of that output port. Egress processing is optional, a switch may not support any egress tables or may not be configured to use them. If no valid egress table is configured as the first egress table (see 7.3.2), the packet must be processed by the output port, and in most cases the packet is forwarded out of the switch. If a valid egress table is configured as the first egress table (see 7.3.2), the packet must be matched against flow entries of that flow table, and other egress flow tables may be used depending on the outcome of the match in that flow table. When processed by a flow table, the packet is matched against the flow entries of the flow table to select a flow entry (see 5.3). If a flow entry is found, the instruction set included in that flow entry is executed. These instructions may explicitly direct the packet to another flow table (using the Goto-Table Instruction, see 5.5), where the same process is repeated again. A flow entry can only direct a packet to a flow table number which is greater than its own flow table number, in other words pipeline processing can only go forward and not backward. Obviously, the flow entries of the last table of a pipe line stage can not include the Goto-Table instruction. If the matching flow entry does not direct packets to another flow table, the current stage of pipeline processing stops at this table, the packet is processed with its associated action set and usually forwarded (see 5.6). If a packet does not match a flow entry in a flow table, this is a table miss. The behavior on a table miss depends on the table configuration (see 5.4). The instructions included in the table-miss flow entry in the flow table can flexibly specify how to process unmatched packets, useful options include dropping them, passing them to another table or sending them to the controllers over the control channel via packet-in messages (see 6.1.2). There are few cases where a packet is not fully processed by a flow entry and pipeline processing stops without processing the packet's action set or directing it to another table. If no table-miss flow entry is present, the packet is dropped (see 5.4). If an invalid TTL is found, the packet may be sent to the controller (see 5.8). The OpenFlow pipeline and various OpenFlow operations process packets of a specific type in conformance with the specifications defined for that packet type, unless the present specification or the OpenFlow configuration specify otherwise. For example, the Ethernet header definition used by OpenFlow must conform to IEEE specifications, and the TCP/IP header definition used by OpenFlow must conform to RFC specifications. Additionally, packet reordering in an OpenFlow switch must conform to the requirements of IEEE specifications, provided that the packets are processed by the same flow entries, group buckets and meter bands.

Packet Processing Flowchart in OF Switch



Header match fields are match fields matching values extracted from the packet headers. Most header match fields map directly to a specific field in the packet header defined by a datapath protocol.

On receipt of a packet, an OpenFlow Switch performs the functions shown in the Figure. The switch starts by performing a table lookup in the first flow table, and based on pipeline processing, may perform table lookups in other flow tables. Packet header fields are extracted from the packet, and packet pipeline fields are retrieved. Packet header fields used for table lookups depend on the packet type, and typically include various protocol header fields, such as Ethernet source address or IPv4 destination address. In addition to packet headers, matches can also be performed against the ingress port, the metadata field and other pipeline fields. Metadata may be used to pass information between tables in a switch. The packet header fields and pipeline fields represent the packet in its current state, if actions applied in a previous table using the Apply-Actions instruction changed the packet headers or pipeline fields, those changes are reflected in the packet header fields and pipeline fields. A packet matches a flow entry if all the match fields of the flow entry are matching the corresponding header fields and pipeline fields from the packet. If a match field is omitted in the flow entry (i.e. value ANY), it matches all possible values in the header field or pipeline field of the packet. If the match field is present and does not include a mask, the match field is matching the corresponding header field or pipeline field from the packet if it has the same value. If the switch supports arbitrary bit masks on specific match fields, these masks can more precisely specify matches, the match field is matching if it has the same value for the bits which are set in the mask (see 7.2.3.5). The packet is matched against flow entries in the flow table and only the highest priority flow entry that matches the packet must be selected. The counters associated with the selected flow entry must be updated (see 5.9) and the instruction set included in the selected flow entry must be executed (see 5.5). If there are multiple matching flow entries with the same highest priority, the selected flow entry is explicitly undefined. This case can only arise when a controller writer never sets the OFPFF_CHECK_OVERLAP bit on flow mod messages and adds overlapping entries. IP fragments must be reassembled before pipeline processing if the switch configuration contains the OFPC_FRAG_REASM flag (see 7.3.2). This version of the specification does not define the expected behavior when a switch receives a mal-formed or corrupted packet either on an OpenFlow port (see 4.1) or in a Packet-Out message (see 6.1.1).

Instructions in OpenFlow

- Instructions are executed when a **packet matches** an entry in a table
- Instructions result in changes to the packet, action set and/or pipeline processing
- A switch is not required to support all instruction types, just those marked “Required Instruction”
- Optional Instruction: **Apply-Actions action (s)**: Applies the specific action(s) immediately, without any change to the Action Set. This instruction may be used to modify the packet between two tables or to execute multiple actions of the same type.
- Required Instruction: **Write-Actions action(s)**: Merges the specified set of action(s) into the current action set. If an action of the given type exists in the current set, overwrite it, otherwise add it. If a set-field action with a given field type exists in the current set, overwrite it, otherwise add it.
- Optional Instruction: **Write-Metadata metadata / mask**: Writes the masked metadata value into the metadata field. The mask specifies which bits of the metadata register should be modified

Instructions in OpenFlow

- Optional Instruction: **Stat-Trigger stat thresholds**: Generate an event to the controller if some of the flow statistics cross one of the stat threshold values.
- Required Instruction: **Goto-Table next-table-id**: Indicates the next table in the processing pipeline. The table-id must be greater than the current table-id. This instruction must be supported in all flow tables except the last one, OpenFlow switches with only a single flow table are not required to implement this instruction. The flow entries of the last table of the pipeline cannot include this instruction
 - When the instruction set does not contain a Goto-Table instruction, pipeline processing stops and the actions are executed

Apply-Actions instructions

copy TTL inwards: apply copy TTL inward actions to the packet

pop: apply all tag pop actions to the packet

push-MPLS: apply MPLS tag push action to the packet

push-PBB: apply PBB tag push action to the packet

push-VLAN: apply VLAN tag push action to the packet

copy TTL outwards: apply copy TTL outwards action to the packet

decrement TTL: apply decrement TTL action to the packet

set: apply all set-field actions to the packet

qos: apply all QoS actions, such as meter and set queue to the packet

group: if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list

output: if no group action is specified, forward the packet on the port specified by the output action

PBB: provider backbone bridging

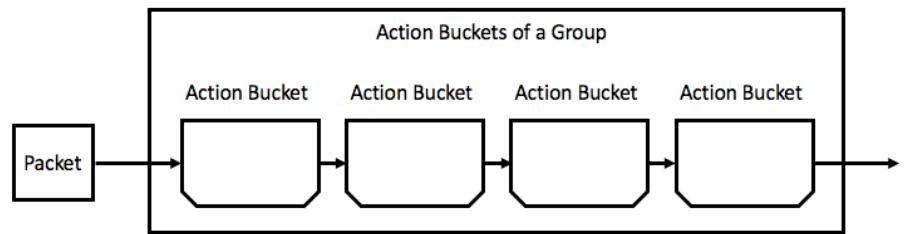
OpenFlow: Group table

Enables additional methods of forwarding

A group table consists of group entries

A group entry may consist of zero or more buckets

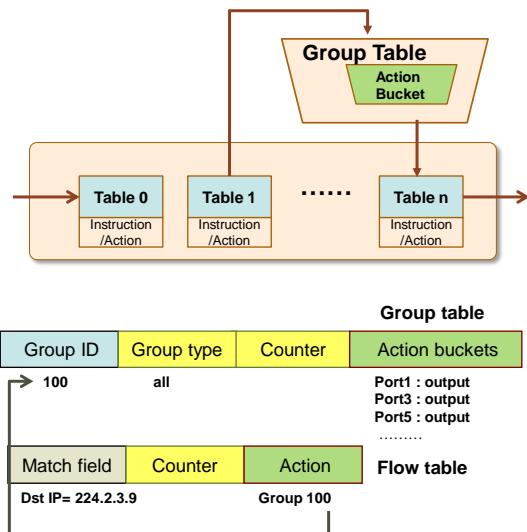
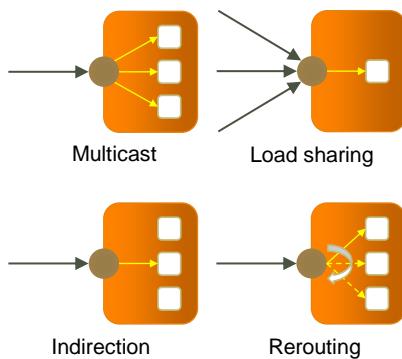
A bucket typically contains actions that modify the packet and an output action that forwards it to a port



OpenFlow Group Table

Group Table & Types (version 1.1)

- All: multicast
- Select: load sharing
- Indirect: simple indirection
- Fast-failover: rerouting



Each group entry is identified by its group identifier and contains:

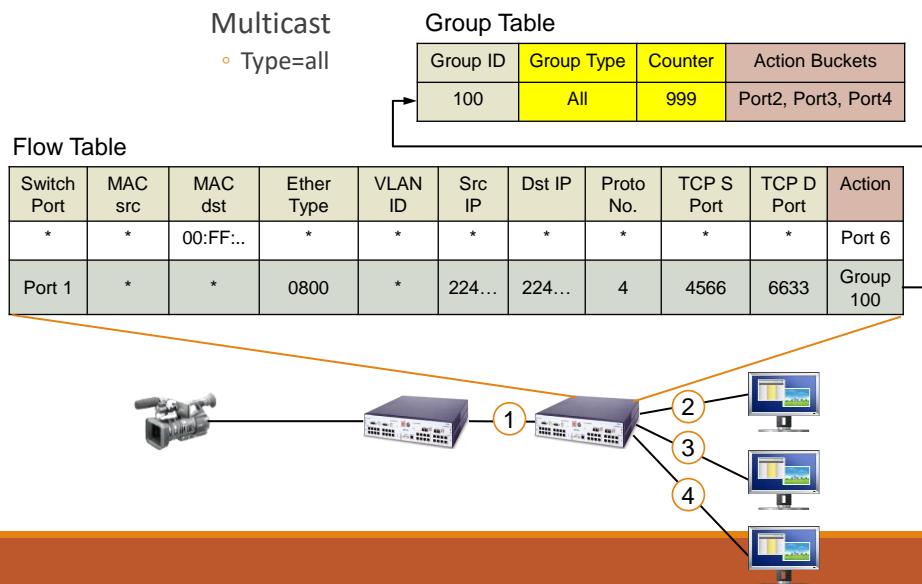
- **group identifier:** a 32 bit unsigned integer uniquely identifying the group on the OpenFlowswitch.

- **group type:** to determine group semantics
- **counters:** updated when packets are processed by a group.
- **action buckets:** an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters. The actions in a bucket are always applied as an action set

All—Multiple buckets are implemented for the handling of **multicast** and broadcast packets. Each incoming packet is replicated and processed by each bucket in the group.

Indirect—One bucket is implemented. An indirect group is typically referenced by multiple flow entries, thereby allowing each of these entities to have a centralized action that can be easily updated.

OpenFlow Group Table

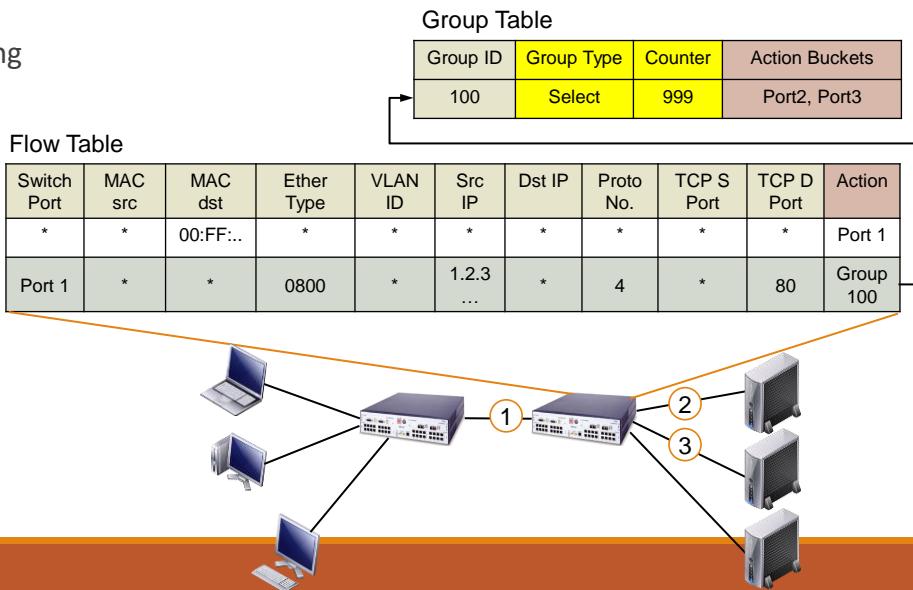


Required:all: Execute all buckets in the group. This group is used for multicast or broadcastforwarding. The packet is effectively cloned for each bucket; one packet is processed for eachbucket of the group. If a bucket directs a packet explicitly out the ingress port, this packet cloneis dropped. If the controller writer wants to forward out the ingress port, the group must includean extra bucket which includes an output action to theOFPP_IN_PORTreserved port.

OpenFlow Group Table

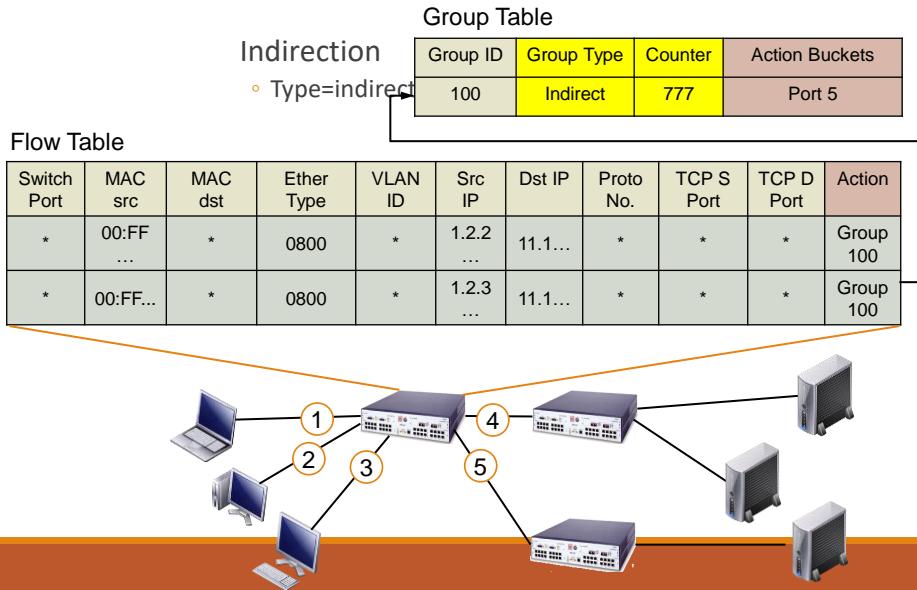
Load Balancing

- Type=select



Execute one bucket in the group. Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). All configuration and state for the selection algorithm are external to OpenFlow. The selection algorithm should implement equal load sharing and can optionally be based on bucket weights. When a port specified in a bucket in a select group goes down, the switch may restrict bucket selection to the remaining set (those with forwarding actions to live ports) instead of dropping packets destined to that port. This behavior may reduce the disruption of a downed link or switch.

OpenFlow Group Table



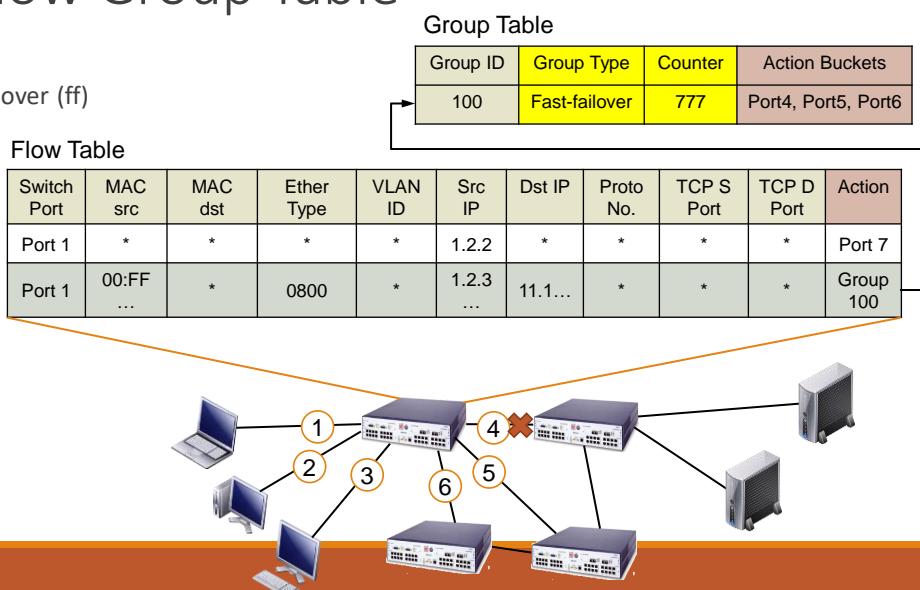
In informatica l'**indirezione** (detta anche riferimento **indiretto**) è la tecnica che consente di indicare un oggetto o un valore mediante un suo riferimento invece che direttamente.

This group supports only a single bucket. Allows multiple flow entries or groups to point to a common group identifier, supporting faster, more efficient convergence (e.g. next hops for IP forwarding). This group type is effectively identical to an all group with one bucket. This group is the simplest type of group, and therefore switches will typically support a greater number of them than other group types.

OpenFlow Group Table

Fast Failover

- Type=fast-failover (ff)



Execute the first live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. The buckets are evaluated in the order defined by the group, and the first bucket which is associated with a live port/group is selected. This group type enables the switch to change forwarding without requiring a round trip to the controller. If no buckets are live, packets are dropped. This group type must implement a liveness mechanism (see 6.7).

Ethertype 800: IPv4

OpenFlow Meter Table

Meter Table (ver 1.3)

- Counts packet rate of a matched flow
- QoS control → Rate-limit, DiffServ ...

Meter Table

Meter ID	Band Type	Rate	Counter	Argument
100	Drop (remark DSCP)	1000 kbps	1000	xxx

Flow Table

Switch Port	MAC src	MAC dst	Ether Type	Src IP	Dst IP	Proto No.	TCP S Port	TCP D Port	Inst. Meter	Action
Port 1	*	*	*	1.2.2	*	*	*	*	N/A	Port 7
Port 1	00:FF ...	*	0800	1.2.3 ...	11.1...	*	*	*	Meter 100	Port 2

A meter table consists of meter entries, defining per-flow meters.

meter identifier: a 32 bit unsigned integer uniquely identifying the meter

- meter bands: an unordered list of meter bands, where each meter band specifies the rate of the band and the way to process the packet
- counters: updated when packets are processed by a meter

Each meter band specifies a target rate for that band and a way packets should be processed if that rate is exceeded. The default meter band is always included in the meter and can not be set, it is equivalent to a band with target rate 0 that does nothing, it just lets packet through without doing anything

band type: defines how packets are processed
•rate: target rate for that band - used by the meter to select the meter band, usually the lowest rate at which the band can apply
•burst: defines the granularity of the meter band
•counters: updated when packets are processed by a meter band
•type specific arguments: some band types have optional arguments

Packet Forwarding in OpenFlow

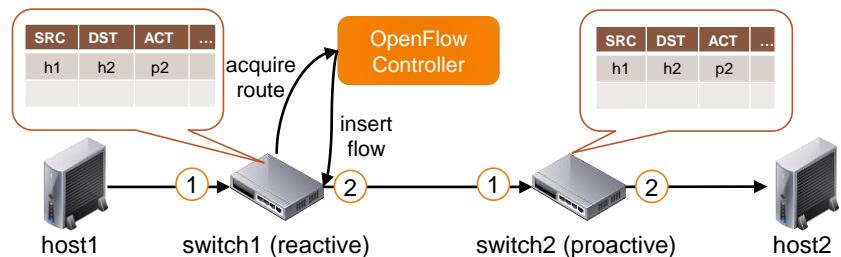
Packet Forwarding

Reactive flow insertion

- A non-matched packet reaches to OpenFlow switch, it is sent to the controller, based on the info in packet header, an appropriate flow will be inserted
- Always need to query the path from controller during packet arrival → slow
- Can reflect the current traffic status

Proactive flow insertion

- Flow can be inserted proactively by the controller to switches before packet arrives
- No need to communicate during packet arrival → fast packet forwarding
- Cannot reflect the current traffic status

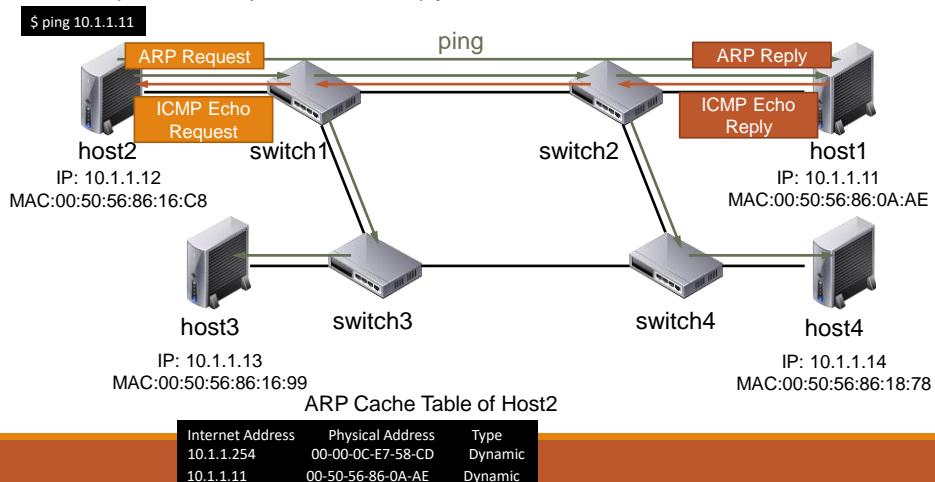


Outline

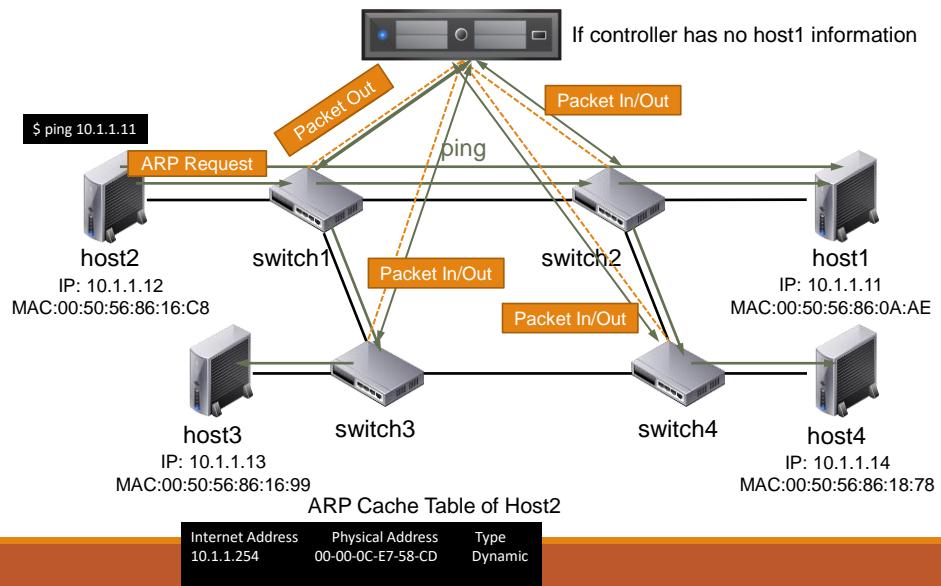
- SDN motivations: Internet ossification, network complexity, barriers to innovation
- SDN approach
- OpenFlow
- Application examples
- SDN and cloud
- SDN and Network Function Virtualization

Communication in Legacy Network

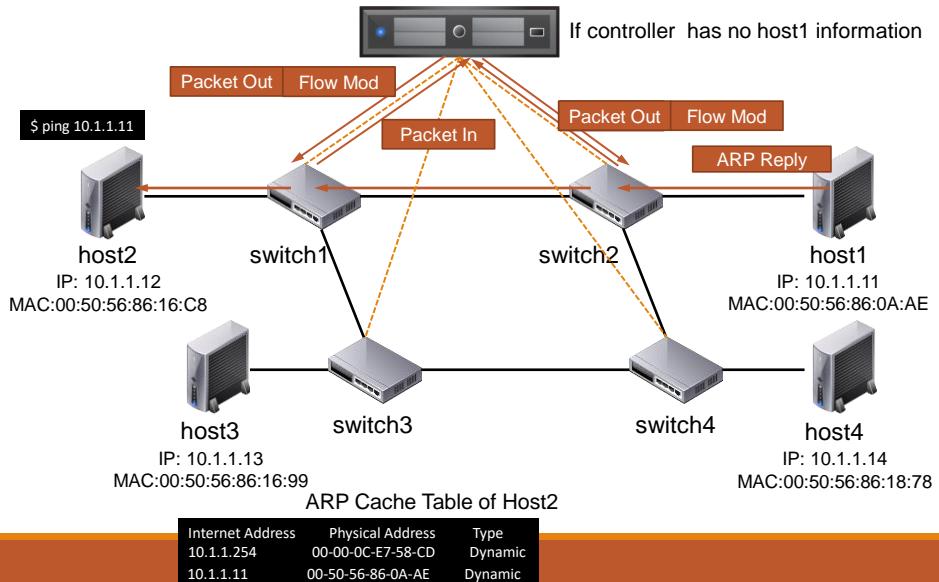
1. host2 tries communication to host1 by sending a ping ICMP packet
2. host2 broadcasts ARP Request packet
3. host1 replies ARP Request with ARP Reply
4. host2 creates entry to ARP Cache Table
5. host2 sends ICMP Echo request packet
6. host1 replies ICMP Echo request with ICMP Echo reply



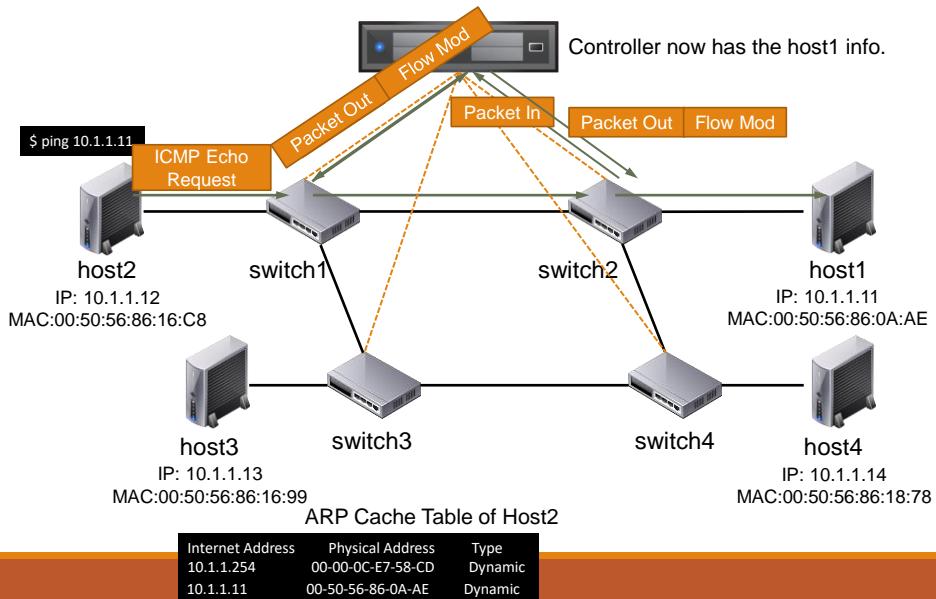
Communication in OpenFlow



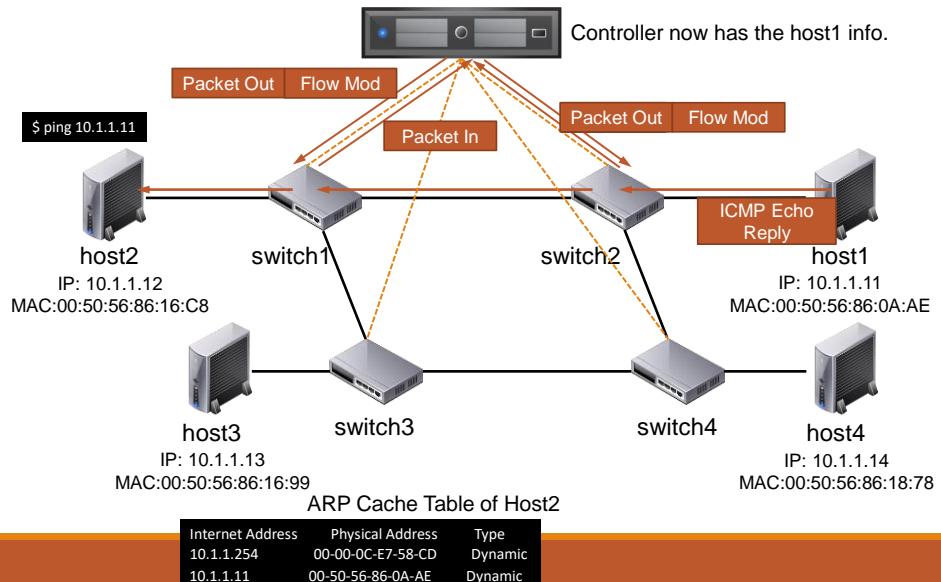
Communication in OpenFlow



Communication in OpenFlow



Communication in OpenFlow



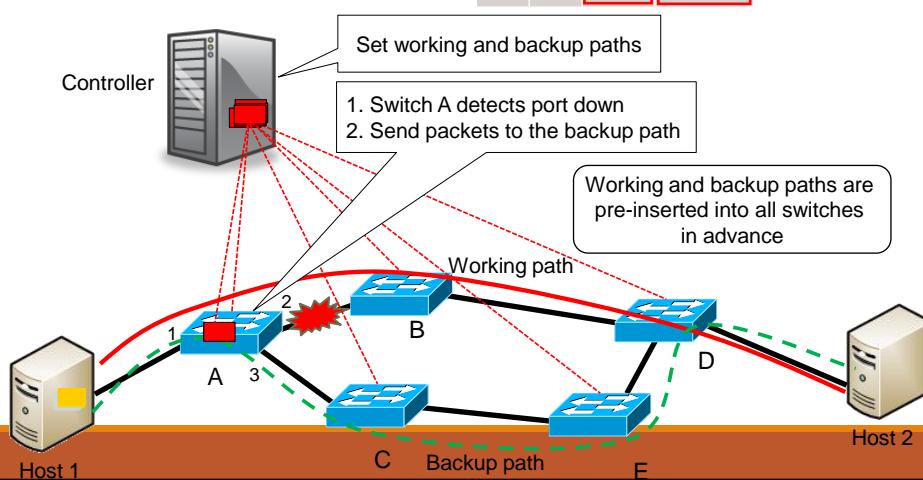
OpenFlow Failover

OpenFlow Failover

- Protection

Flow table of Switch A (group table combined)

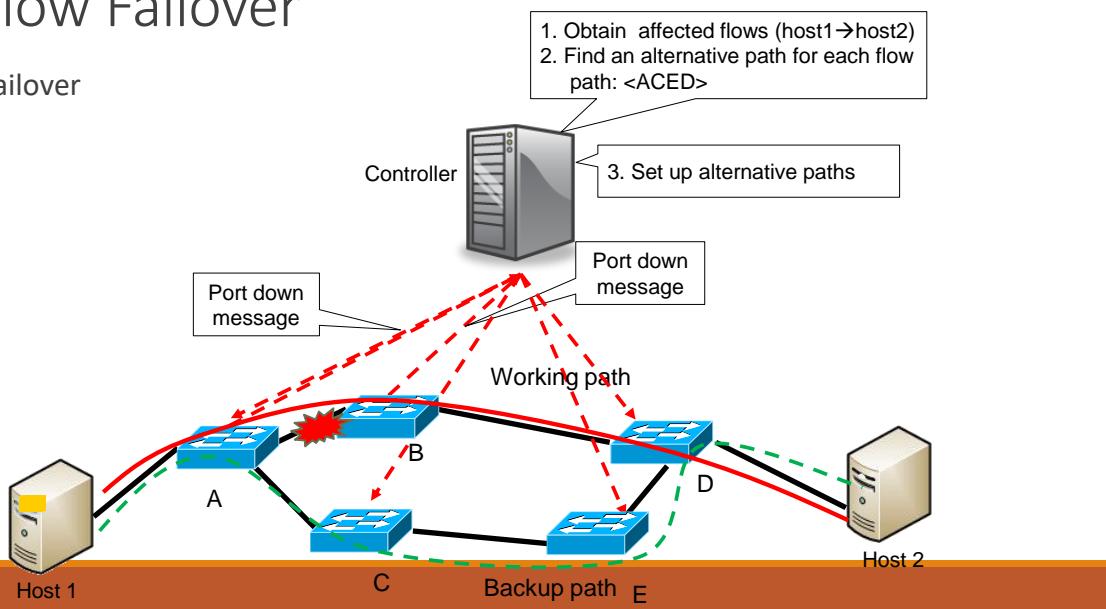
src	dst	Out port	Failover port
h1	h2	2	3



OpenFlow Failover

OpenFlow Failover

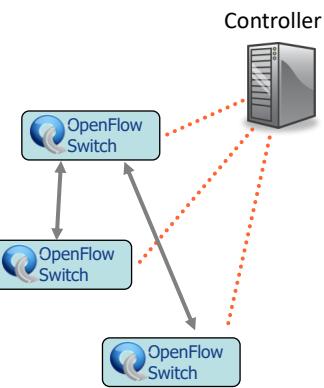
- Restoration



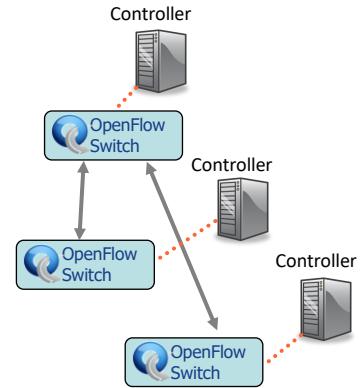
Centralized vs Distributed Control

Both models are possible with OpenFlow

Centralized Control



Distributed Control



Flow Routing vs. Aggregation

Both models are possible with OpenFlow

Flow-Based

- Every flow is individually set up by controller
- Exact-match flow entries
- Flow table contains one entry per flow
- Good for fine grain control, e.g. campus networks

Aggregated

- One flow entry covers large groups of flows
- Wildcard flow entries
- Flow table contains one entry per category of flows
- Good for large number of flows, e.g. backbone

Examples of SDN devices

Hardware support



59

Not only switches but other network components

Outline

- SDN motivations: Internet ossification, network complexity, barriers to innovation
- SDN approach
- OpenFlow
- Application examples
- SDN and cloud
- SDN and Network Function Virtualization

SDN and cloud

Cloud computing service providers face the issue of multi-tenancy at the network level

IP and Ethernet each have virtual network capability, but limited in terms of

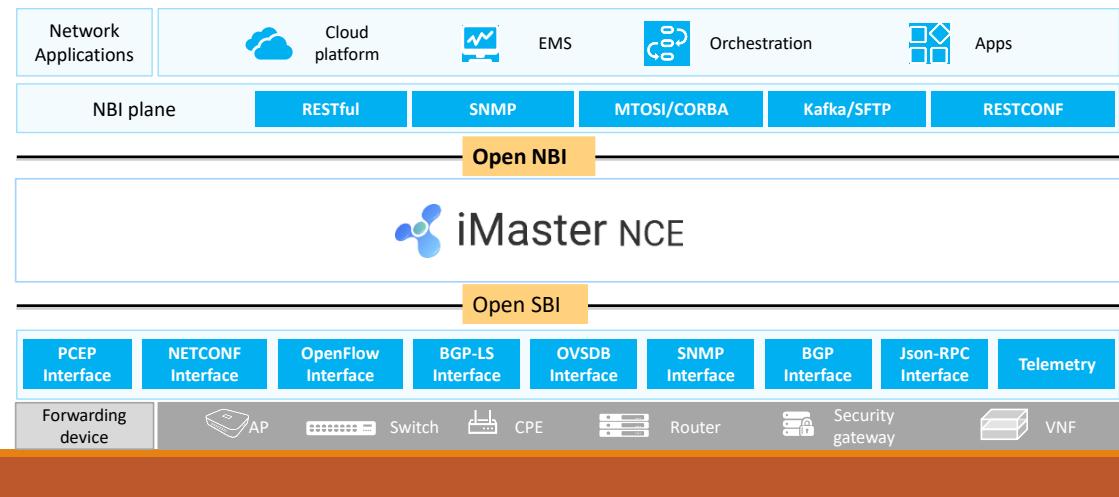
- how many tenants can be supported
- how isolated each tenant
- configuration and management complexity

SDN is increasingly accepted as the path to "cloud networking"

For example, AWS is the world's largest implementer of SDN. They utilize the massive scale of their global network, data centers, and servers to offer an amazing array of networking services.

Huawei SDN Network Architecture

Huawei SDN network architecture supports various SBIs and NBIs, including OpenFlow, OVSDB, NETCONF, PCEP, RESTful, SNMP, BGP, JSON-RPC, and RESTCONF interfaces.



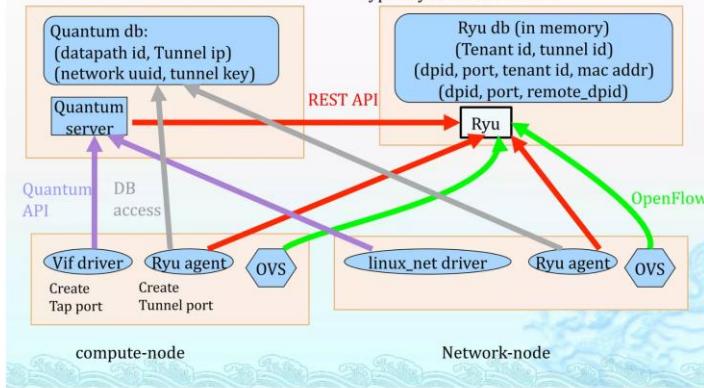
iMaster NCE collects network data through protocols such as SNMP and telemetry, performs intelligent big data analysis based on AI algorithms, and displays device and network status in multiple helping dimensions through dashboards and reports, O&M personnel quickly detect and handle device and network exceptions and ensuring the normal running of devices and networks.

OpenStack and SDN

How Ryu works with OpenStack

Quantum-node: somewhere where compute/network can communicate.
Typically on network-node

Ryu-node: somewhere where compute/network/quantum can communicate
Typically on network-node

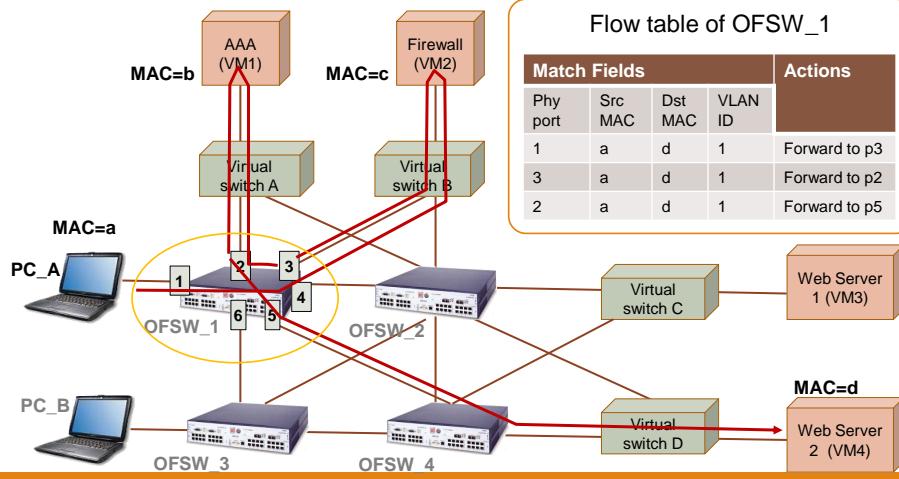


Outline

- SDN motivations: Internet ossification, network complexity, barriers to innovation
- SDN approach
- OpenFlow
- Application examples
- SDN and cloud
- SDN and Network Function Virtualization

Example

Example of Routing Control (hop-by-hop routing)



Origin of NFV

In October 2012, 13 top carriers (including AT&T, Verizon, VDF, DT, T-Mobile, BT, and Telefonica) released the first version of NFV White Paper at the SDN and OpenFlow World Congress. In addition, the Industry Specification Group (ISG) was founded to promote the definition of network virtualization requirements and the formulation of the system architecture.

In 2013, the ETSI NFV ISG conducted the first phase of research and completed the formulation of related standards. The ETSI NFV ISG defined NFV requirements and architecture and sorts out the standardization processes of different interfaces.

In 2015, NFV research entered the second phase. The main research objective is to build an interoperable NFV ecosystem, promote wider industry participation, and ensure that the requirements defined in phase 1 are met. In addition, the ETSI NFV ISG (Industry Specification Group) specified the collaboration relationships between NFV and SDN standards and open source projects. Five working groups are involved in NFV phase 2: IFA (architecture and interface), EVE (ecosystem), REL (reliability), SEC (security), and TST (test, execution, and open source). Each working group mainly discusses the deliverable document framework and delivery plan.

The ETSI NFV standard organization cooperates with the Linux Foundation to start the open source project OPNFV (NFV open source project, providing an integrated and open reference platform), integrate resources in the industry, and actively build the NFV industry ecosystem. In 2015, OPNFV released the first version, further promoting NFV commercial deployment.

NFV-related standard organizations include:

ETSI NFV ISG (Industry Specification Group): formulates NFV requirements and functional frameworks.

3GPP SA5 working group: focuses on technical standards and specifications of 3GPP NE virtualization management (MANO-related).

Open Platform for NFV (OPNFV): provides an open-source platform project that accelerates NFV marketization.

NFV Value

NFV aims to address issues such as complex deployment and O&M and service innovation difficulties due to large numbers of telecom network hardware devices. NFV brings the following benefits to carriers while reconstructing telecom networks:

- Shortened service rollout time
- Reduced network construction cost
- Improved network O&M efficiency
- Open ecosystem



67

Shortened service rollout time: In the NFV architecture, adding new service nodes becomes simple. No complex site survey or hardware installation is required. For service deployment, you only need to request virtual resources (compute, storage, and network resources) and software loading, simplifying network deployment. To update service logic, you simply need to add new software or load new service modules to complete service orchestration. Service innovations become simple.

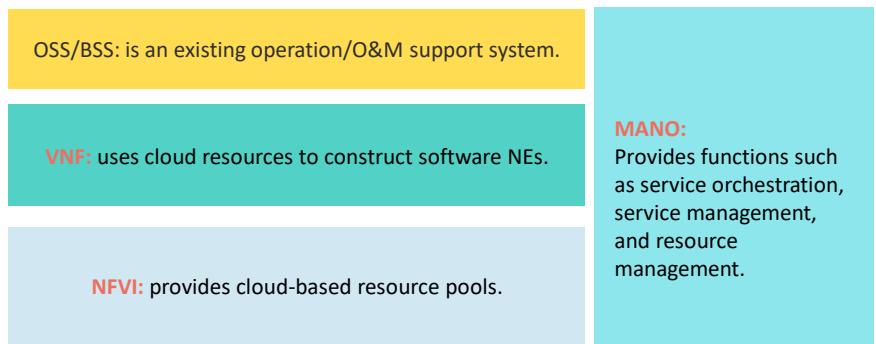
Reduced network construction cost: Virtualized NEs can be integrated into COTS devices to reduce the cost. Enhancing network resource utilization and lowering power consumption can lower overall network costs. NFV uses cloud computing technologies and universal hardware to build a unified resource pool. Resources are dynamically allocated on demand based on service requirements, implementing resource sharing and improving resource utilization. For example, automatic scale-in and scale-out can be used to solve the resource usage problem in the tidal effect.

Enhanced network O&M efficiency: Automated and centralized management improves the operation efficiency and reduces the O&M cost. Automation includes DC-based hardware unit management automation, MANO application service life management automation, NFV- or SDN-based coordinated network automation.

Open ecosystem: The legacy telecom network exclusive software/hardware model defines a closed system. NFV-based telecom networks use an architecture based on standard hardware platforms and virtual software. The architecture easily provides open platforms and open interfaces for third-party developers, and allows carriers to build open ecosystems together with third-party partners.

Introduction to the NFV Architecture

The NFV architecture includes the network functions virtualization infrastructure (NFVI), a virtualized network function (VNF), and management and orchestration (MANO). In addition, the NFV architecture needs to support the existing business support system (BSS) or operations support system (OSS).



Copyright © 2020 Huawei Technologies Co., Ltd. All rights reserved.

Each layer of the NFV architecture can be provided by different vendors, which improves system development but increases system integration complexity.

NFV implements efficient resource utilization through device normalization and software and hardware decoupling, reducing carriers' TCO, shortening service rollout time, and building an open industry ecosystem.

The NFVI consists of the hardware layer and virtualization layer, which are also called COTS ((Commercial) Off-the-Shelf component), and CloudOS in the industry.

COTS: universal hardware, focusing on availability and universality, for example, Huawei FusionServer series hardware server.

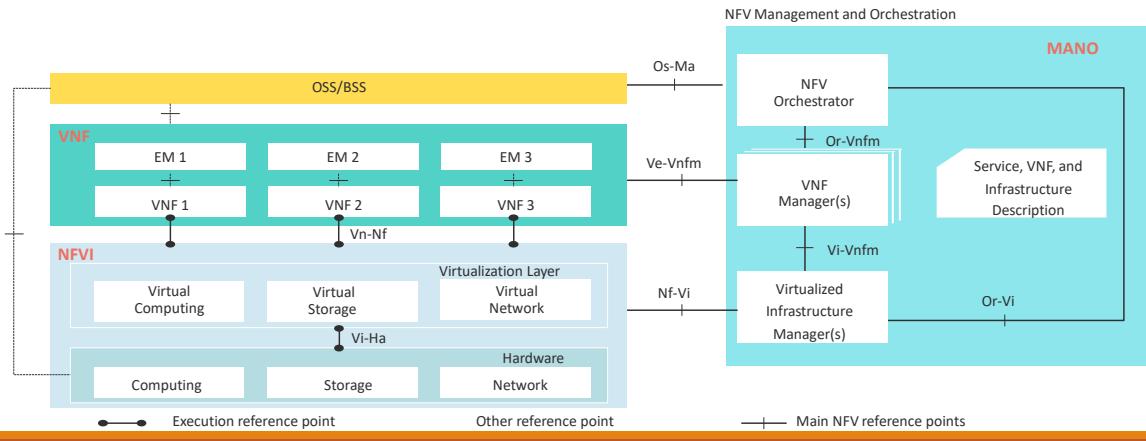
CloudOS: cloud-based platform software, which can be regarded as the operating system of the telecom industry. CloudOS virtualizes physical compute, storage, and network resources into virtual resources for upper-layer software to use, for example, Huawei FusionSphere.

VNF: A VNF can be considered as an app with different network functions and is implemented by software of traditional NEs (such as IMS, EPC, BRAS, and CPE) of carriers.

MANO: MANO is introduced to provision network services in the NFV multi-CT or multi-IT vendor environment, including allocating physical and virtual resources, vertically streamlining management layers, and quickly adapting to and interconnecting with new vendors' NEs. The MANO includes the Network Functions Virtualization Orchestrator (NFVO, responsible for lifecycle management of network services), Virtualized Network Function Manager (VNFM, responsible for lifecycle management of VNFs), and Virtualized Infrastructure Manager (VIM, responsible for resource management of the NFVI).

Standard NFV Architecture

ETSI defines the standard NFV architecture, which consists of the NFVI, VNF, and MANO. The NFVI includes the universal hardware layer and virtualization layer. The VNF is implemented using software, and the MANO implements management and orchestration of an NFV architecture.



Copyright © 2020 Huawei Technologies Co., Ltd. All rights reserved.

The NFV architecture includes the network functions virtualization infrastructure (NFVI),, a virtualized network function (VNF), and management and orchestration (MANO). In addition, the NFV architecture needs to support the existing business support system (BSS) or operations support system (OSS).

ETSI defines the standard NFV architecture, which consists of NFVI, VNF, and MANO components. NFVI consists of common hardware facilities and virtualization. VNFs use software to implement virtualized network functions, and MANOs manage and orchestrate the NFV architecture.

Functional Modules of the NFV Architecture

Main functional modules defined in the standard NFV architecture:

OSS or BSS: A management system for a Service provider. It is not a functional component in the NFV architecture, but the MANO must provide an interface for interoperation with the OSS or BSS.

MANO: NFV management and orchestration. The MANO includes the VIM, VNFM, and NFVO, and provides unified management and orchestration for VNFs and the NFVI.

VIM: NFVI management module that runs on an infrastructure site. The VIM provides functions such as resource discovery, virtual resource management and allocation, and fault handling.

VNFM: It controls the VNF lifecycle (including instantiation, configuration, and shutdown).

NFVO: It orchestrates and manages all the software resources and network services on an NFV network.

VNF: VNFs refer to VMs as well as service NEs and network function software deployed on the VMs.

EM (Element Management) – is an element management system for VNF. This is responsible for the functional management of VNF i.e. FCAPS (Fault, Configuration, Accounting, Performance and Security Management). This may manage the VNFs through proprietary interfaces. There may be one EMS per VNF or an EMS can manage multiple VNFs. EMS itself can be a VNF. EM does the management of functional components (for example issues related to mobile signalling).

NFVI: NFV infrastructure, including required hardware and software. The NFVI provides a running environment for VNFs.

Hardware layer: includes hardware devices that provide compute, network, and storage resources.

Virtualization layer: abstracts hardware resources to form virtual resources, such as virtual compute, storage, and network resources. The virtualization function is implemented by Hypervisor.

BSS: business support system are the components that a telecommunications service provider (or telco) uses to run its business operations towards customers. BSS deals with the taking of orders, payment issues, revenues, etc. It supports four processes: product management, order management, revenue management and customer management.

OSS: operation support system are computer systems used by telecommunications service providers to manage their networks (e.g., telephone networks). They support management functions such as network inventory, service provisioning, network configuration and fault management.Togther with business support systems (BSS), they are used to support various end-to-end telecommunication services. BSS and OSS have their own data and service responsibilities.

A hypervisor is a software layer between physical servers and OSs. It allows multiple OSs and applications to share the same set of physical hardware. It can be regarded as a meta operating system in the virtual environment and can coordinate all physical resources and VMs on the server. It is also called virtual machine monitor (VMM). The hypervisor is the core of all virtualization technologies. Mainstream hypervisors include KVM, VMware ESXi, Xen, and Hyper-V.

Functional Modules of the NFV Architecture

Main functional modules defined in the standard NFV architecture:

OSS or BSS	Management system for a service provider. It is not a functional component in the NFV architecture, but the MANO must provide an interface for interoperation with the OSS or BSS.
MANO	NFV management and orchestration. The MANO includes the VIM, VNFM, and NFVO, and provides unified management and orchestration for VNFs and the NFVI. <ul style="list-style-type: none">• VIM: NFVI management module that runs on an infrastructure site. The VIM provides functions such as resource discovery, virtual resource management and allocation, and fault handling.• VNFM: It controls the VNF lifecycle (including instantiation, configuration, and shutdown).• NFVO: It orchestrates and manages all the software resources and network services on an NFV network.
VNF	VNFs refer to VMs as well as service NEs and network function software deployed on the VMs.
NFVI	NFV infrastructure, including required hardware and software. The NFVI provides a running environment for VNFs. <ul style="list-style-type: none">• Hardware layer: includes hardware devices that provide compute, network, and storage resources.• Virtualization layer: abstracts hardware resources to form virtual resources, such as virtual compute, storage, and network resources. The virtualization function is implemented by Hypervisor^[1].

Copyright © 2020 Huawei Technologies Co., Ltd. All rights reserved.

BSS: business support system

OSS: operation support system

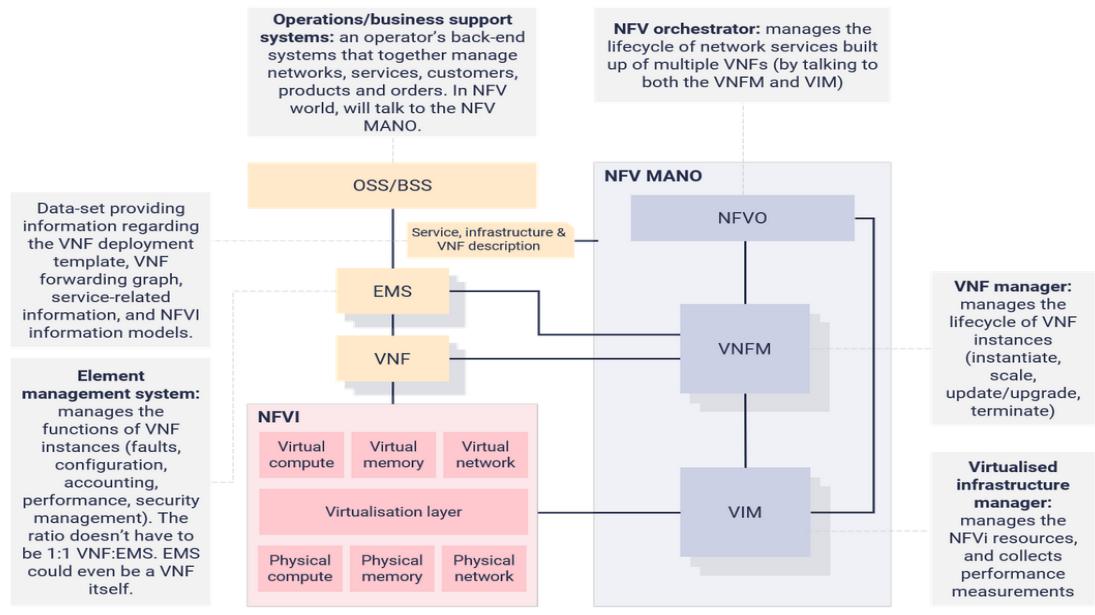
A hypervisor is a software layer between physical servers and OSs. It allows multiple OSs and applications to share the same set of physical hardware. It can be regarded as a meta operating system in the virtual environment, and can coordinate all physical resources and VMs on the server. It is also called virtual machine monitor (VMM). The hypervisor is the core of all virtualization technologies. Mainstream hypervisors include KVM, VMWare ESXi, Xen, and Hyper-V.

NFV Architecture Interfaces

Main interfaces of the standard NFV architecture:

Interface	Description
Vi-Ha	Is used between the virtualization layer and hardware layer. The virtualization layer meets basic hardware compatibility requirements.
Vn-Nf	Is used between a VM and the NFVI. It ensures that VMs can be deployed on the NFVI to meet performance, reliability, and scalability requirements. The NFVI meets VMs' OS compatibility requirements.
Nf-Vi	Is used between the virtualization layer management software and NFVI. It provides management of virtual computing, storage, and network systems of NFVI, virtual infrastructure configuration and connections, as well as system usage, performance monitoring, and fault management.
Ve-Vnfm	Is used between the VNFM and a VNF, implementing VNF lifecycle management, VNF configuration, VNF performance, and fault management.
OS-Ma	Manages lifecycles of network services and VNFs.
Vi-Vnfm	Is used for interaction between the service application management system or service orchestration system and virtualization layer management software.
Or-Vnfm	Sends configuration information to the VNFM, configures the VNFM, and connects the orchestrator and VNFM. It exchanges information with the NFVI resources allocated to VNFs and information between VNFs.
Or-Vi	Is used to send resource reservation and resource allocation requests required by the orchestrator and exchange virtual hardware resource configurations and status information.

Detailed ETSI NFV Reference Architecture



ETSI identified three main working domains:

1) Virtual Network Functions (VNFs)

These are individual functions of a network that have been virtualised. Possibilities are endless – firewalls, Evolved Packet Core, etc.

2) NFV infrastructure (NFVI)

The infrastructure required to run the VNFs. This is made up of hardware resources (computing servers and network switches), and virtual resources (“abstractions” of the hardware on which the VNFs run, known as “virtual machines” (VMs). A virtualisation layer (the “hypervisor”) exists to abstract between the two.

3) NFV management & orchestration (MANO)

MANO is the framework for management and orchestration of all the resources in the NFV environment. It is where the management of resources in the infrastructure layer takes place, and is also where resources are created and delegated and allocation of VNFs is managed.

Suggested Readings

- A. Manzalini, V. Vercellone, M. Ullio, «Software Defined Networking: sfide ed opportunità per le reti del futuro», Notiziario Tecnico Telecom Italia, n.1/2003
<http://www.telecomitalia.com/content/dam/telecomitalia/it/archivio/documenti/Innovazione/NotiziarioTecnico/2013/n1-2013/NT1-4-2013.pdf>
- N. McKeown et al. «OpenFlow: Enabling Innovation in Campus Networks», CCR 2008
<http://www.openflow.org/documents/openflow-wp-latest.pdf>
- For more information about OpenFlow, visit <https://www.opennetworking.org/>



Part 2 – OpenStack: Architecture and Components

Gianluca Reali

Università degli Studi di Perugia

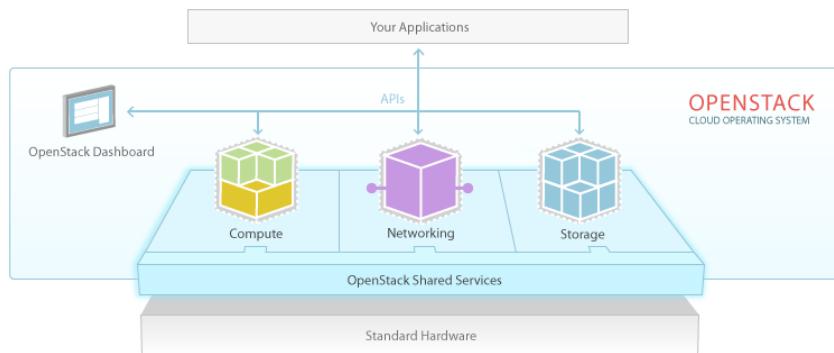
Ref. - Chap. 13 of the book “Introduction to Middleware” by

Letha Hughes Etzkorn

- www.opestack.org



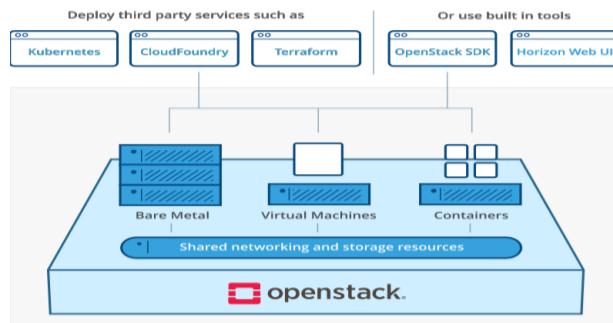
OpenStack



OpenStack is a cloud operating system that controls large **pools of compute, storage, and networking resources** throughout a datacenter. A **dashboard** that gives administrators control while empowering their users to provision resources through a **web interface**.



OpenStack

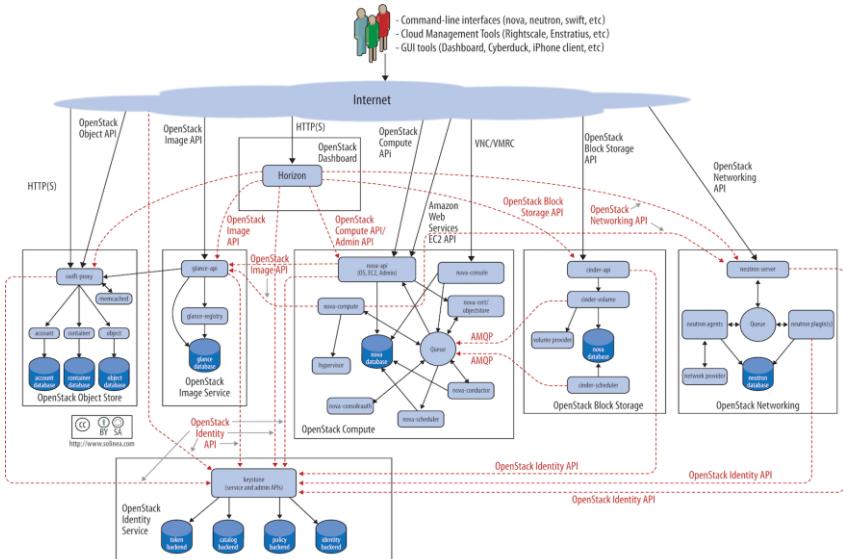


The OpenStack system consists of several key projects that you install separately. These projects work together depending on your cloud needs. These projects include **Compute, Identity Service, Networking, Image Service, Block Storage, Object Storage, Telemetry, Orchestration, and Database**. You can install any of these projects separately and configure them stand-alone or as connected entities.



OS Logical architecture

The following diagram shows the most common, but not the only possible, architecture for an OpenStack cloud:



<https://docs.openstack.org/install-guide/get-started-logical-architecture.html>

To design, deploy, and configure OpenStack, administrators must understand the logical architecture. OpenStack modules are one of the following types:

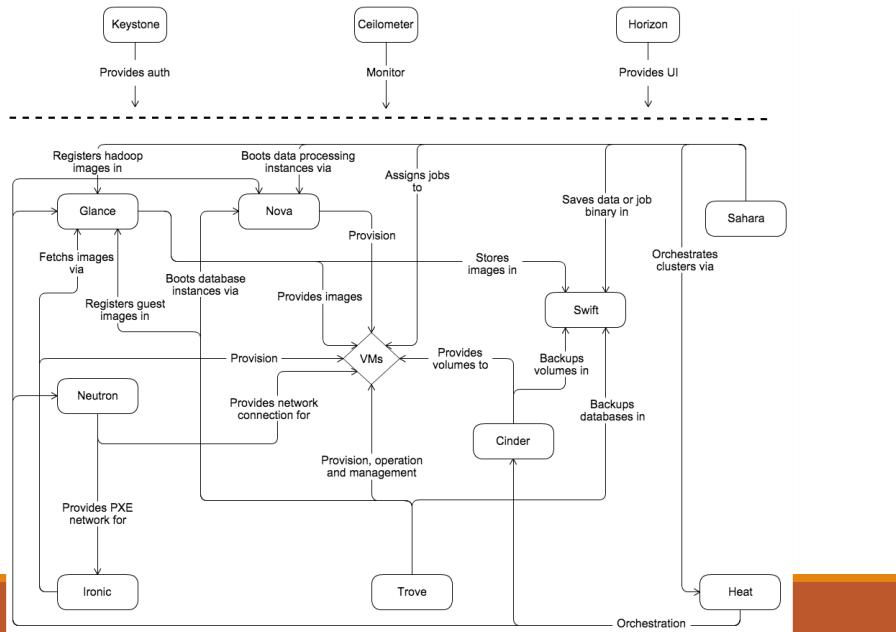
Daemon Runs as a background process. On Linux platforms, a daemon is usually installed as a service. Script Installs a virtual environment and runs tests. Command-line interface (CLI) Enables users to submit API calls to OpenStack services through commands. [OpenStack Logical Architecture](#) shows one example of the most common integrated services within OpenStack and how they interact with each other. End users can interact through the dashboard, CLIs, and APIs. All services authenticate through a common Identity service, and individual services interact with each other through public APIs, except where privileged administrator commands are necessary.

Advanced Message Queuing Protocol (AMQP) è uno standard aperto che definisce un protocollo a livello applicativo per il message-oriented middleware. AMQP è definito in modo tale da garantire funzionalità di messaggistica, accodamento, routing (con paradigmi punto-punto e pubblicazione-sottoscrizione), affidabilità e sicurezza.



OS Conceptual architecture

relationships among the OpenStack services



<https://docs.openstack.org/install-guide/get-started-conceptual-architecture.html>

Trove is Database as a Service for OpenStack. It's designed to run entirely on [OpenStack](#), with the goal of allowing users to quickly and easily utilize the features of a relational or non-relational

database without the burden of handling complex administrative tasks.

OpenStack bare metal provisioning a.k.a **Ironic** is an integrated OpenStack program which aims to provision bare metal machines instead of virtual machines, forked from the Nova baremetal driver.

The sahara project aims to provide users with a simple means to provision data processing frameworks (such as Apache Hadoop, Apache Spark and Apache Storm) on OpenStack. This is accomplished by specifying configuration parameters such as the framework version, cluster topology, node hardware details and more.



OS Conceptual architecture

- OpenStack consists of several independent parts, named the **OpenStack services**. All services authenticate through a **common Identity service**. Individual services interact with each other through **public APIs**, except where privileged administrator commands are necessary.
- Internally, OpenStack services are composed of **several processes**. All services have at least one **API process**, which listens for API requests, preprocesses them and passes them on to other parts of the service. With the exception of the Identity service, the actual work is done by distinct processes.



Message Oriented Middleware

- Openstack, as any other modern large-scale applications, is built as **distributed** network applications, with parts of the application in distinct processes and, possibly, in distinct parts of the world.
- All the same, the parts need to work together to behave as one reliable application. They need a way to communicate, and they must be able to tolerate failures.
- In principle, information could be shared through database or REST (e.g. HTTP), but it has some serious drawbacks.
 - A database is reliable, but it isn't designed to intermediate communication. Its focus is storing data, not moving it between processes.
 - REST helps you communicate efficiently, but it offers no reliability. If the party you're talking to is unavailable, the transmission is dropped.

Message Oriented Middleware

- The **store-and-forward messaging system** gives you efficient, reliable communication. Message brokers take responsibility for ensuring messages reach their destination, even if the destination is temporarily out of reach. Messaging APIs manage acknowledgments so that no messages are dropped in transit.
 - Remember Publisher-Subscriber pattern and queue-based Cloud Design Patterns



Message Oriented Middleware

- **Loosely Coupled Architecture:** OpenStack uses a [message queue](#) to coordinate operations and status information among services. The message queue service typically runs on the controller node.
- OpenStack projects use **AMQP**, an open standard for messaging middleware.
- The service's state is stored in a database. When deploying and configuring your OpenStack cloud, you can choose among several message broker and database solutions, such as RabbitMQ, MySQL, MariaDB, and SQLite.
- The default message queue service used is [RabbitMQ](#).
- **RabbitMQ** broker stays between internal components of each service in OpenStack (e.g. nova-conductor, nova-scheduler in Nova service) and allow them to communicate each other in the loosely coupled way.

The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

The AMQP specification is defined in several layers: (i) a type system, (ii) a symmetric, asynchronous protocol for the transfer of messages from one process to another, (iii) a standard, extensible message format and (iv) a set of standardised but extensible 'messaging capabilities.'



OpenStack Services/Projects

- **Dashboard /Horizon**

- Provides a web-based self-service portal to interact with underlying OpenStack services, such as launching an instance, assigning IP addresses and configuring access controls.

- **Compute /Nova**

- Manages the lifecycle of compute instances in an OpenStack environment. Responsibilities include spawning, scheduling and decommissioning of virtual machines on demand.

- **Networking /Neutron**

- Enables Network-Connectivity-as-a-Service for other OpenStack services, such as OpenStack Compute. Provides an API for users to define networks and the attachments into them. Has a pluggable architecture that supports many popular networking vendors and technologies.



OpenStack Services/Projects

Shared services

- **Identity service / Keystone**

- Provides an authentication and authorization service for other OpenStack services.
Provides a catalog of endpoints for all OpenStack services.

- **Image service / Glance**

- Stores and retrieves virtual machine disk images. OpenStack Compute makes use of this during instance provisioning.

- **Telemetry / Ceilometer**

- Monitors and meters the OpenStack cloud for billing, benchmarking, scalability, and statistical purposes.



OpenStack Services/Projects

Storage services

- **Object Storage / Swift**

- Stores and retrieves arbitrary unstructured data objects via a RESTful, HTTP based API. It is highly fault tolerant with its data replication and scale-out architecture. Its implementation is not like a file server with mountable directories. In this case, it writes objects and files to multiple drives, ensuring the data is replicated across a server cluster.

- **Block Storage / Cinder**

- Provides persistent block storage to running instances. Its pluggable driver architecture facilitates the creation and management of block storage devices.



OpenStack Services/Projects

Higher-level services

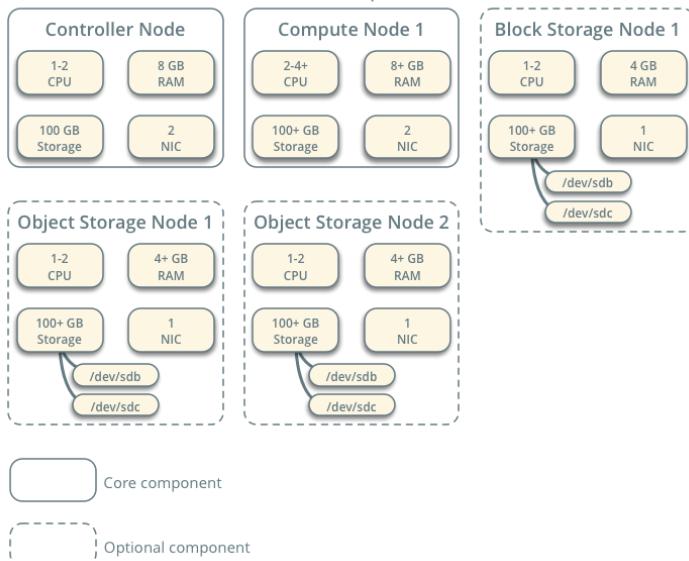
Orchestration / Heat

- Orchestrates multiple composite cloud applications by using either the native HOT template format or the AWS CloudFormation template format, through both an OpenStack-native REST API and a CloudFormation-compatible Query API.



Example architecture

Hardware Requirements





Example architecture

- **Controller:** The controller node runs the Identity service, Image service, management portions of Compute, management portion of Networking, various Networking agents, and the Dashboard. It also includes supporting services such as an SQL database, [message queue](#), and [NTP](#).

Optionally, the controller node runs portions of the Block Storage, Object Storage, Orchestration, and Telemetry services.

The controller node requires a minimum of two network interfaces.



Example architecture

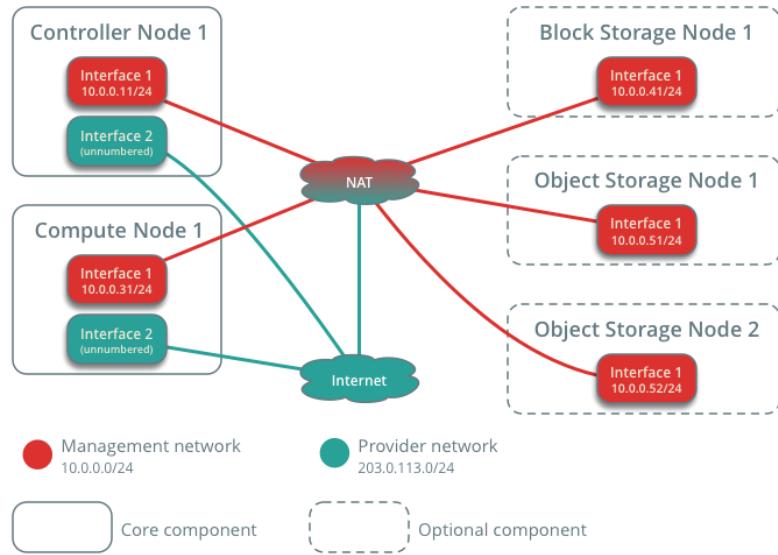
- **Compute:** The compute node runs the [hypervisor](#) portion of Compute that operates instances. By default, Compute uses the [KVM](#) hypervisor. The compute node also runs a Networking service agent that connects instances to virtual networks and provides [firewalling services to instances via security groups](#).
 - You can deploy more than one compute node. Each node requires a minimum of two network interfaces.
- **Block Storage:** The optional Block Storage node contains the disks that the Block Storage and Shared File System services provision for instances.
 - Production environments should implement a separate storage network to increase performance and security.
 - You can deploy more than one block storage node. Each node requires a minimum of one network interface.

Example architecture

- **Object Storage:** The optional Object Storage node contain the disks that the Object Storage service uses for storing accounts, containers, and objects.
 - Production environments should implement a separate storage network to increase performance and security.
 - Each node requires a minimum of one network interface. You can deploy more object storage nodes.



Network Layout





Networking Alternatives

Networking

You can choose one of the following virtual networking options.

- **Networking Option 1: Provider networks:** The provider networks option deploys the OpenStack Networking service in the simplest way possible with primarily layer-2 (bridging/switching) services and VLAN segmentation of networks. Essentially, it bridges virtual networks to physical networks and relies on physical network infrastructure for layer-3 (routing) services. Additionally, a DHCP service provides IP address information to instances.
- **Networking Option 2: Self-service networks:** The self-service networks option augments the provider networks option with layer-3 (routing) services that enable self-service networks using overlay segmentation methods such as VXLAN. Essentially, it routes virtual networks to physical networks using NAT. Additionally, this option provides the foundation for advanced services such as LBaaS and FWaaS.

19

Load Balancer as a Service (LBaaS)

Firewall as a Service (FWaaS)

How to use OpenStack

- Basic install gives you a project and login named demo, and a project and login named admin.
- You can, of course, add additional projects.

The screenshot shows the OpenStack Horizon dashboard at the URL `192.168.56.101/dashboard/project/instances/`. The top navigation bar has a dropdown for 'Project' set to 'demo'. The main content area is titled 'Compute / Instances' and displays a table of running instances. The table has columns: Instance Name, Image Name, IP Address, Size, Key Pair, Status, Availability Zone, Task, Power State, Time since created, and Actions. One instance is listed:

Instance Name	Image Name	IP Address	Size	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
10d0bf1672fb0f8163efffe0ca77	m1.nova	10.0.0.4	10.0.0.4	tryitout	Active	nova	None	Running	3 days, 15 hours	Create Snapshot

The left sidebar shows navigation links for 'Project' (with 'admin', 'alt_demo', and 'demo' listed), 'COMPUTE' (selected), 'Instances', 'Volumes', 'Images', 'Access & Security', 'NETWORK', 'Admin', 'Identity', 'Developer', and 'Help'.

20



How to use OpenStack (cont'd)

- An “**instance**” is what OpenStack calls a running virtual machine
- An **image** is a copy of an operating system.
- When an **instance** is started, it must be given some kind of OS to run
- OpenStack can run many different versions of Linux (Ubuntu, RedHat, Fedora, SUSE, Debian) and it can run Windows
- Several tested images are available or you can create your own **image**

How to use OpenStack (cont'd)

- For testing an instance of a cirros operating system can be used.
- The CirrOS OS is a test OS, that does very little but also doesn't require much memory to run:
 - cirros-0.3.2-x86_64-uec
- Using m1.nano size
- Using a previously created **keypair**



How to use OpenStack (cont'd)

- The **keypair** associated with an OpenStack instance consists of a **public key** and a **private key**
- You can use the **OpenStack dashboard** to **create** these, or you can create them other ways (such as with *ssh-keygen*)
- The public key must be registered with OpenStack
- You must **download** the private key to **keep it safe**:
 - You will use your private key later on to **access** your OpenStack instance



How to use OpenStack (cont'd)

The **flavor** tab provides flavors that correspond to choices about:

- Number of virtual CPUs
- How much disk memory (different kinds of disks)
- How much RAM



How to use OpenStack (cont'd)

The screenshot shows the OpenStack dashboard at the URL <http://192.168.56.103/dashboard/project/instances>. The interface is in English. The top navigation bar includes a back button, a search bar, and user information for 'admin'. The left sidebar has a 'Project' dropdown set to 'demo', followed by sections for COMPUTE (Overview, Instances, Volumes, Images), ACCESS & SECURITY, NETWORK, Admin, Identity, and Developer. The 'Instances' section is currently selected. The main content area displays a table of running instances:

Instance Name	Image Name	IP Address	Size	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
myveryowninstance	climbs-0.3.4-x86_64-uec	10.0.0.4	m1.nano	myveryownkeypair	Active	nova	None	Running	0 minutes	Create Snapshot
i	-	10.0.0.8	m1.nano	trytout	Active	nova	None	Running	2 days, 15 hours	Create Snapshot
anothernewinstance	-	10.0.0.7	m1.nano	trytout	Active	nova	None	Running	3 days, 1 hour	Create Snapshot

25



How to use OpenStack (cont'd)

- Overcommitting CPU and RAM: OpenStack allows you to overcommit CPU and RAM on compute nodes. This allows you to increase the number of instances running on your cloud at the cost of reducing the performance of the instances. The Compute service uses the following ratios by default:
 - CPU allocation ratio: 16:1
 - RAM allocation ratio: 1.5:1



OpenStack Partitioning

An OpenStack Cloud can be divided into three main hierarchical zones - **Regions**, **Availability Zones** and **Host Aggregates**.

- A Region is full OpenStack deployment, excluding the Keystone and Horizon.
- An Availability Zone (AZ) is a logical regional grouping of compute nodes. Different AZs can be configured across multiple locations , providing high availability.
- Host Aggregates are logical groups of compute nodes and the relating metadata. Only administrators are able to view or create host aggregates.

<https://docs.openstack.org/keystone/latest/admin/identity-concepts.html>

Region: An Identity service API v3 entity. Represents a general division in an OpenStack deployment. You can associate zero or more sub-regions with a region to make a tree-like structured hierarchy. Although a region does not have a geographical connotation, a deployment can use a geographical name for a region, such as us-east.

A Region is full OpenStack deployment, including its own API endpoints, networks and compute resources^[1], excluding the Keystone and Horizon. Each Region shares a single set of Keystone and Horizon services. Each Region shares a single set of Keystone and Horizon services. The default OpenStack region name is RegionOne.

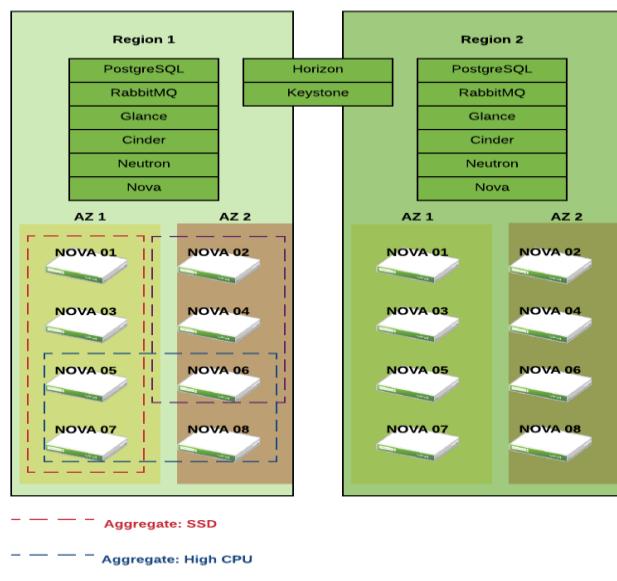
An aggregates metadata is commonly used to provide information for use with the Compute scheduler (for example, limiting specific flavors or images to a subset of hosts). Metadata specified in a host aggregate will limit the use of that host to any instance that has the same metadata specified in its flavor.

An Availability Zone (AZ) is a logical regional grouping of compute nodes. Different AZs can be configured across multiple locations , providing high availability. Unlike Host Aggregates, Availability Zones are exposed to end users

who can select a particular availability zone for VM placement while the instance is launched.



OpenStack Partitioning



<https://docs.openstack.org/keystone/latest/admin/identity-concepts.html>



OpenStack Identity Service: Keystone

Keystone is an OpenStack project that provides **Identity**, **Token**, **Catalog** and **Policy** services for use specifically by projects in the OpenStack family. It implements OpenStack's **Identity API**

- **Core use cases:**
Installation-wide authentication and authorization to OpenStack services
- **Core key capabilities:**
 - Authenticate user / password requests against multiple backends (SQL, LDAP, etc) (**Identity Service**)
 - Validate / manage tokens used after initial username/password verification (**Token Service**)
 - Endpoint registry of available services (**Service Catalog**)
 - Authorize API requests (**Policy Service**)
 - Policy service provides a rule-based (RBAC) authorization engine and the associated rule management interface.

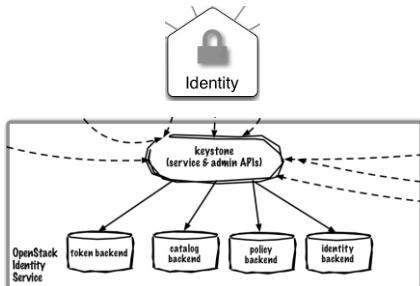


Image Source: <http://www.solinea.com/2013/04/17/openstack-summit-intro-to-openstack-architecture-grizzly-edition/>

29

Role-Based Access Control (RBAC)

Keystone Architecture

Services: Keystone is organized as a group of internal services exposed on one or many endpoints. Many of these services are used in a combined fashion by the frontend.

- **Identity Service**
- **Resource Service**
- **Assignment Service**
- **Token Service**
- **Catalog Service**
- **Policy Service**



The Resource Service (1/2)

The Resource service **provides data about projects and domains**.

- **Projects (Tenants)**: They represent the base unit of ownership in OpenStack, in that all resources in OpenStack should be owned by a specific project. A project itself must be owned by a specific domain, and hence all project names are not globally unique, but unique to their domain. If the domain for a project is not specified, then it is added to the default domain.
- **Domains**: They are a high-level container for projects, users and groups. Each is owned by exactly one domain. Each domain defines a namespace where an API-visible name attribute exists. Keystone provides a default domain, named 'Default'.

31

<https://docs.openstack.org/keystone/latest/admin/identity-concepts.html>

Domain: An Identity service API v3 entity. Domains are a collection of projects and users that define administrative boundaries for managing Identity entities. Domains can represent an individual, company, or operator-owned space. They expose administrative activities directly to system users. Users can be granted the administrator role for a domain. A domain administrator can create projects, users, and groups in a domain and assign roles to users and groups in a domain.

Project: A container that groups or isolates resources or identity objects. Depending on the service operator, a project might map to a customer, account, organization, or tenant.



The Resource Service (2/2)

In the Identity v3 API, the uniqueness of attributes is as follows:

- **Domain Name.** Globally unique across all domains.
- **Project Name.** Unique within the owning domain.
- **User Name.** Unique within the owning domain.
- **Group Name.** Unique within the owning domain.
- **Role Name.** Unique within the owning domain.

32

<https://docs.openstack.org/keystone/latest/admin/identity-concepts.html>



The Identity Service (1/3)

The **Identity service** is typically the first service a user interacts with. The Identity service **provides auth credential validation and data about users and groups**.

- **Users**

Users represent an individual API consumer. A user itself must be **owned by a specific domain**, and hence all user names are **not globally unique**, but only unique to their domain.

- **Groups**

Groups are containers representing a collection of users. A group itself must be **owned by a specific domain**, and hence all group names are **not globally unique**, but only unique to their domain.

33

<https://docs.openstack.org/keystone/latest/admin/identity-concepts.html>

Role: A personality with a defined set of user rights and privileges to perform a specific set of operations. The Identity service issues a token to a user that includes a list of roles. When a user calls a service, that service interprets the user role set, and determines to which operations or resources each role grants access.

Group: An Identity service API v3 entity. Groups are a collection of users owned by a domain. A group role, granted to a domain or project, applies to all users in the group. Adding or removing users to or from a group grants or revokes their role and authentication to the associated domain or project.



The Identity Service (2/3)

Once authenticated, an end user can use their identity to access other OpenStack resources through deployed services, based on user or group roles.

1. Users and services can locate other services by using the service catalog. As the name implies, a service catalog is a collection of available services and the relevant endpoints in an OpenStack deployment.
2. Each OpenStack service needs a service entry with corresponding endpoints stored in the Identity service. Each endpoint can be one of three types: **admin, internal, or public**.

34

Each service can have one or many endpoints and each endpoint can be one of three types: admin, internal, or public.

The admin API endpoint allows modifying users and tenants by default, while the public and internal APIs do not allow these operations.

In a production environment, different endpoint types might reside on separate networks exposed to different types of users for security reasons. For instance, the public API network might be visible from the Internet so customers can manage their clouds. The admin API network might be restricted to operators within the organization that manages cloud infrastructure. The internal API network might be restricted to the hosts that contain OpenStack services



The Identity Service (3/3)

The Identity service contains the following components:

- **Server:** A centralized server provides authentication and authorization services using a RESTful interface.
- **Modules:** Entities that intercept service requests, extract user credentials, and send them to the centralized server for authorization.
- **Drivers:** Components integrated to the centralized server. They are used for accessing identity information in repositories external to OpenStack (for example, SQL databases or LDAP servers).

35

The Lightweight Directory Access Protocol (LDAP) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network. Directory services play an important role in developing intranet and Internet applications by allowing the sharing of information about users, systems, networks, services, and applications throughout the network. As examples, directory services may provide any organized set of records, often with a hierarchical structure, such as a corporate email directory. Similarly, a telephone directory is a list of subscribers with an address and a phone number.



The Assignment Service (1/1)

To assign a user to a project, you must assign a **role** to a **user-project pair**. For example:

```
openstack role add --user USER_NAME --project PROJECT_NAME ROLE_NAME
```

The Assignment service **provides data about roles and role assignments**.

- **Roles** dictate the level of authorization the end user can obtain. Roles can be granted at either the domain or project level. A role can be assigned at the individual user or group level. Role names are unique within the owning domain.
- **Role Assignments** is therefore a 3-tuple that has a **Role**, a **Resource** and an **Identity**



Keystone Authentication

Keystone allows different kinds of authentication; these include:

- Username/password
- Token-based
 - Universally Unique Identifier (UUID) tokens
 - Public Key Infrastructure (PKI)/ Public Key Infrastructure Compressed (PKIZ) tokens
 - Fernet tokens
 - JSON Web Signature – JWS - tokens

37

One way to authenticate is to employ username/password. However, you would have to supply

them in every separate command to an API, which means that each command sent is a chance for

your username and password to be stolen. It also means that you have to store the username and password

locally, or you will have to re-enter them (type them in) for every separate command! It can be

dangerous to store them locally, since someone might steal them, but re-entering them every time is

just too much in a situation where you have multiple commands.

An alternative is to use tokens. A token is valid only for a limited time, which means that even if a

token is stolen when it is being sent across the network in a command to an API, it is a short time until

the thief can no longer use it. Similarly, if you store (cache) a token locally, even if it is stolen in the

case when your local account is hacked, it is a short time until the thief can no longer use it.



The Token Service

The Token service validates and manages tokens used for authenticating requests once a user's credentials have already been verified

38

The token type issued by keystone is configurable through the /etc/keystone/keystone.conf file. Currently, the only supported token provider is fernet.



Token Types and Authentication

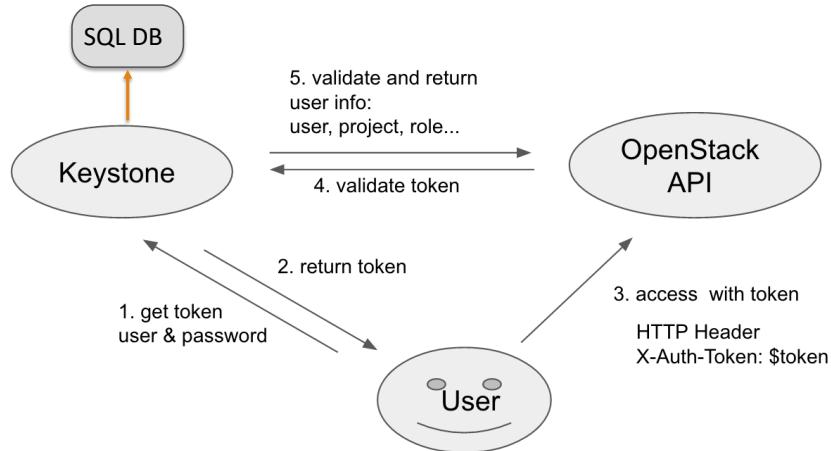
When configured to use **UUID tokens**, Keystone generates tokens in UUID 4 format. UUID tokens are 128 bits (16 bytes) long. e.g. 12ea1917-4641-49dc-ad68-d0dc1eb30842

The way that UUID tokens work on Keystone is as follows:

- **First stage—user acquires a token:**
 - The user sends a username/password to Keystone to request a token
 - Keystone generates a UUID token (in UUID 4 format) and sends it back to the user
 - Keystone keeps a copy of this token, together with user identification information, in its database
- **Subsequent stages—user uses the token:**
 - User sends a request to some other OpenStack component services (such as Nova or Glance) including the token
 - The OpenStack component service sends the token to Keystone for authentication
 - If the token is in the Keystone database and has not expired and has not been revoked, then Keystone tells the other OpenStack component service, okay go ahead. Otherwise Keystone tells the other OpenStack component service, nope, I don't recognize that token.



Token Types and Authentication



Authentication by UUID token



Token Types and Authentication

- The **size of tokens is a major issue**, since performance degrades when huge amounts of token data are transmitted across a network and also when huge amounts of token data are stored in a persistent token database, since authentication requests require searching for the token in persistent storage, and can take a very long time.
- This can result in the need to repeatedly flush the persistent token database. Note that some token formats are larger than others, which exacerbates the problem if used in this way.
- Keystone must periodically prune tokens from its token database, those expired, and any users whose user privileges have been revoked.



Token Types and Authentication

- One problem with the way Keystone uses UUID tokens is that every user of any OpenStack component service must send a validation message to Keystone. This can make a lot of network traffic to Keystone.
- Another important problem is the management of the token repository
- Another option is for Keystone to use **Public Key Infrastructure (PKI) tokens**.

When using PKI tokens, Keystone acts as a Certification Authority. At the time Keystone is installed, a **Certificate Authority private key**, a **Certificate Authority certificate**, a **Signing Private Key**, and a **Signing Certificate** are generated.

PKI stands for Public Key Infrastructure. Tokens are documents, cryptographically signed using the X509 standard. In order to work correctly token generation requires a public/private key pair. The public key must be signed in an X509 certificate, and the certificate used to sign it must be available as a Certificate Authority (CA) certificate. These files can be generated either using the keystone-manage utility, or externally generated. The files need to be in the locations specified by the top level Identity service configuration file /etc/keystone/keystone.conf as specified in the above section. Additionally, the private key should only be readable by the system user that will run the Identity service.

When using Public Key Infrastructure (PKI) tokens with the identity service, users must have access to the signing certificate and the certificate authority's (CA) certificate for the token issuer in order to validate tokens. This extension provides a simple means of retrieving these certificates from an identity service.

Certificates are a public resource and can be shared. Typically when validating a certificate we would only require the issuing certificate authority's certificate however PKI tokens are distributed without including the original signing certificate in the message so this must be retrievable as well.



Token Types and Authentication

When you're using either PKI or PKIZ tokens (we'll look at PKIZ tokens shortly), then the entire normal Keystone validation response is included as part of the token. This interface information in the Keystone response is called **service catalog**.

```
{  
  "endpoints": [  
    {  
      "id": "bd130670250c4cb0bba1750a54b66be0",  
      "interface": "internal",  
      "region": "RegionOne",  
      "region_id": "RegionOne",  
      "url": "http://146.229.233.30:8773/services/Cloud"  
    },  
    {  
      "id": "d265e0d952e2448a9b526d11358a5d9e",  
      "interface": "public",  
      "region": "RegionOne",  
      "region_id": "RegionOne",  
      "url": "http://146.229.233.30:8773/services/Cloud"  
    }  
  ]  
}
```

43



Token Types and Authentication

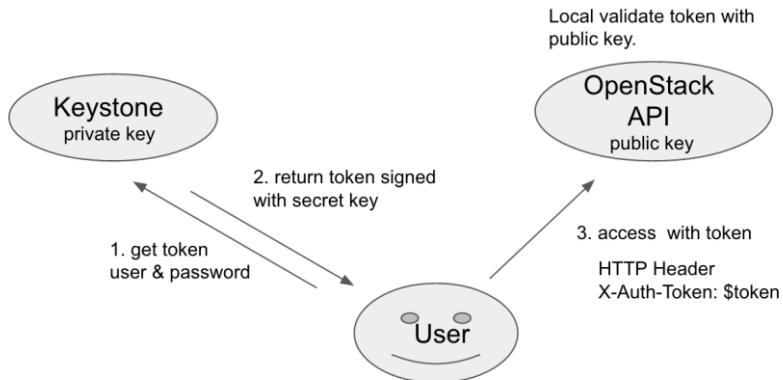
- This information is now converted into **Cryptographic Message Syntax (CMS)** and signed with the Certificate Authority Signing Key. The recipient will of course have to convert it back from CMS and verify the signature before processing it further.
- When a user sends a request including a PKI token to an OpenStack component service via its API, the OpenStack component service/API has a local copy of the Signing Certificate, the Certificate Authority Certificate, and the Certification Revocation List (CRL). If the OpenStack API has not previously downloaded this information from Keystone, then it does it immediately before processing the user's request.

44

The **Cryptographic Message Syntax (CMS)** is the IETF's standard for cryptographically protected messages. It can be used by cryptographic schemes and protocols to digitally sign, digest, authenticate or encrypt any form of digital data.



Token Types and Authentication



Authentication by PKI/PKIZ token

Token Types and Authentication

- These PKI tokens can be humongous. A basic token with a single endpoint is approximately 1700 bytes.
- The PKI token sizes increase proportionally as regions and services are added to the catalog, and sometimes can be over 8KB.
- PKIZ tokens try to fix this problem by compression— PKIZ tokens are just PKI tokens compressed using zlib compression, and except for that PKIZ works the same as PKI.



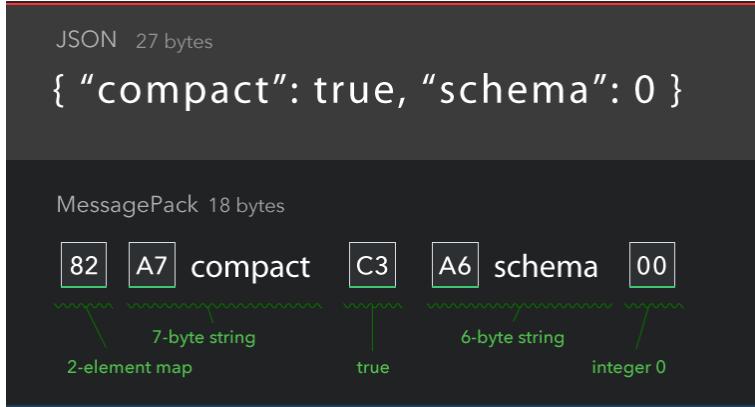
Token Types and Authentication

Fernet tokens are ephemeral, i.e. non-persistent.

- Fernet tokens contain a limited amount of identity and authorization data in a [MessagePacked](#) payload.
- MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller, although less readable by humans. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves.



Token Types and Authentication



<https://msgpack.org/>



Token Types and Authentication

Fernet tokens use **symmetric (secret) Key Encryption**. With Fernet tokens, the data in a message (its payload) is encrypted using the **Advanced Encryption Standard (AES)**, then the **whole Fernet token** is signed using a **SHA256 HMAC key**, and finally the whole is **base 64 URL encoded**. The two keys are commonly referred to as **fernet key**.

- Keys are held in a key repository that keystone passes to a library that handles the encryption and decryption of tokens.

49

A fernet key is used to encrypt and decrypt fernet tokens. Each key is actually composed of two smaller keys: a 128-bit AES encryption key and a 128-bit SHA256 HMAC signing key. The keys are held in a key repository that keystone passes to a library that handles the encryption and decryption of tokens.

Base64 is a group of binary-to-text encoding schemes that represent binary data (more specifically, a sequence of 8-bit bytes) in an ASCII string format by translating the data into a radix-64 representation. The term Base64 originates from a specific MIME content transfer encoding. Each non-final Base64 digit represents exactly 6 bits of data. Three 8-bit bytes (i.e., a total of 24 bits) can therefore be represented by four 6-bit Base64 digits.



Token Types and Authentication

A **key repository** is required by keystone in order to create fernet tokens. The different types are as follows:

- **Primary key:** There is only one primary key in a key repository. The primary key is allowed to encrypt and decrypt tokens. This key is always named as the highest index in the repository.
- **Secondary key:** A secondary key was at one point a primary key, but has been demoted in place of another primary key. It is only allowed to decrypt tokens. Keystone needs to be able to decrypt tokens that were created with old primary keys.
- **Staged key:** The staged key is a special key that shares some similarities with secondary keys. There can only ever be one staged key in a repository and it must exist. Just like secondary keys, staged keys have the ability to decrypt tokens. Unlike secondary keys, staged keys have never been a primary key. In fact, they are opposites since the staged key will always be the next primary key. This helps clarify the name because they are the next key “staged” to be the primary key. This key is always named as 0 in the key repository.

50

A fernet key is used to encrypt and decrypt fernet tokens. Each key is actually composed of two smaller keys: a 128-bit AES encryption key and a 128-bit SHA256 HMAC signing key. The keys are held in a key repository that keystone passes to a library that handles the encryption and decryption of tokens.

Token Types and Authentication

- The fernet keys have a natural lifecycle.
- Each key starts as a staged key, is promoted to be the primary key, and then demoted to be a secondary key.
- New tokens can only be encrypted with a primary key. Secondary and staged keys are never used to encrypt token.
- As an operator, this gives you the chance to perform a key rotation on one keystone node, and distribute the new key set over a span of time. This does not require the distribution to take place in an ultra short period of time
- The key repository is specified using the key_repository option in the keystone configuration file.

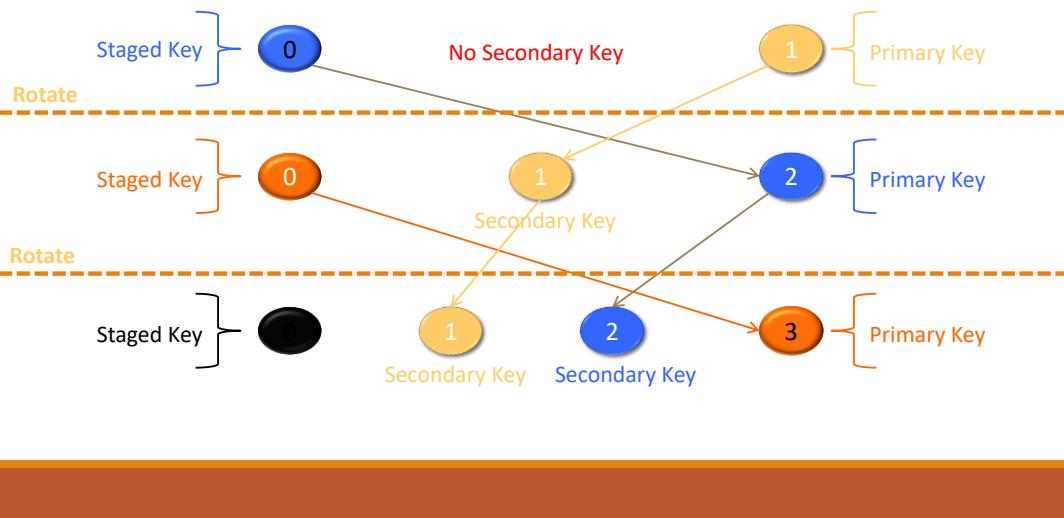
```
[fernet_tokens]  
key_repository= /etc/keystone/fernet-keys/
```

51



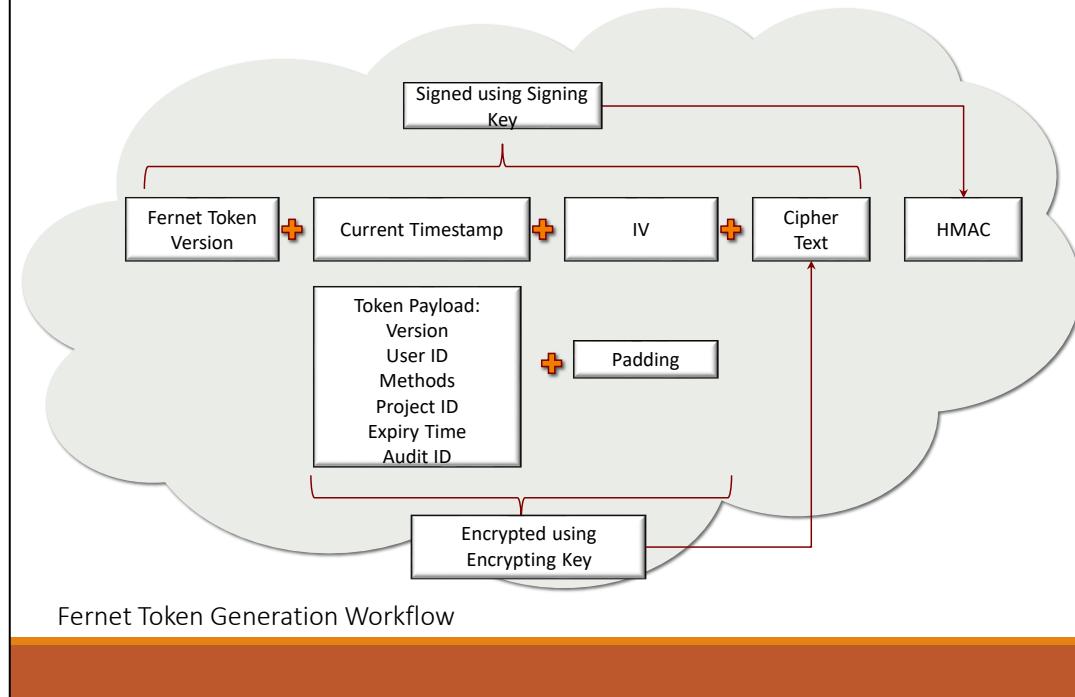
Token Types and Authentication

Fernet Key Rotation





Token Types and Authentication



Identity:

Checks if User exist in User Domain

Check if User is enabled

Retrieves **User ID**

Matches Password

Resource:

Checks if Domain or Project exist

Check if Domain or Project is enabled

Retrieves **Project ID** and **Domain ID**

Catalog:

Retrieves **Services** associated with User's Project

Retrieves the list of **endpoints** for all the services

In cryptography, an **HMAC** (sometimes expanded as either keyed-hash message

authentication code or hash-based message authentication code) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. As with any MAC, it may be used to simultaneously verify both the data integrity and the authenticity of a message.

HMAC can provide message authentication using a shared secret instead of using digital signatures with asymmetric cryptography. It trades off the need for a complex public key infrastructure by delegating the key exchange to the communicating parties, who are responsible for establishing and using a trusted channel to agree on the key prior to communication



Token Types and Authentication

Fernet Token Contents

Field	Info about Contents	Size
0x80	Fernet token version	8 bits
Current timestamp	In UTC	64 bits
Initialization vector	Random number	128 bits
Payload/ciphertext		Variable size, a multiple of 128 bits, padded as necessary
	HMAC	Signed using Signing Key —SHA256

Payload/Ciphertext Format

Field	Info about Contents
• Version	Possible values: <ul style="list-style-type: none">• unscoped=0• domain scoped=1• project scoped=2• trust scoped=3• federated unscoped=4• federated domain scoped=6• federated project scoped=5
User Identifier	UUID
Methods	oauth1, password, token, external
Project identifier	UUID
Expiration time	In UTC
Audit Identifier	Serves as a Token Identifier (URL safe random number)

Different authentication methods

fernet tokens can expire just like any other keystone token formats

54

<https://docs.openstack.org/keystone/latest/admin/tokens-overview.html>

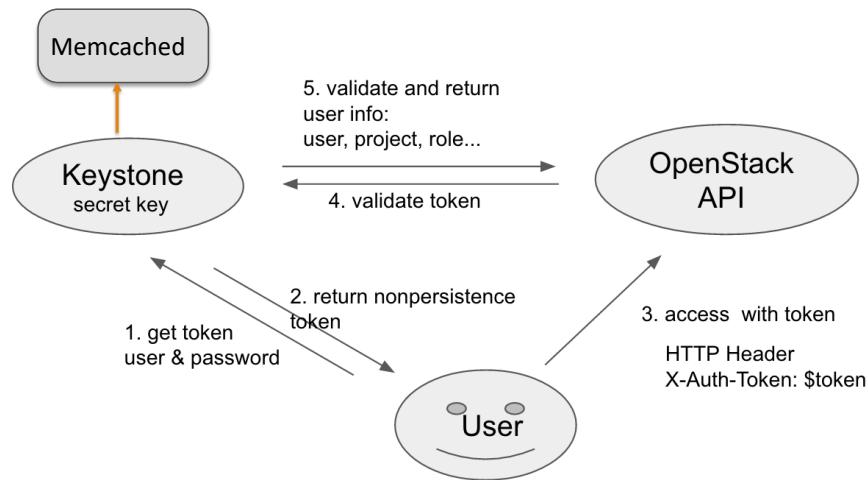
<https://docs.openstack.org/api-ref/identity/v3-ext/#os-trust-api>

A trust represents a user's (the *trustor*) authorization to delegate roles to another user (the *trustee*), and optionally allow the trustee to impersonate the trustor. After the trustor has created a trust, the trustee can specify the trust's id attribute as part of an authentication request to then create a token representing the delegated authority of the trustor.

The audit_ids attribute is a list that contains no more than two elements. Each id in the audit_ids attribute is a randomly (unique) generated string that can be used to track the token.



Token Types and Authentication



Authentication by Fernet token

55

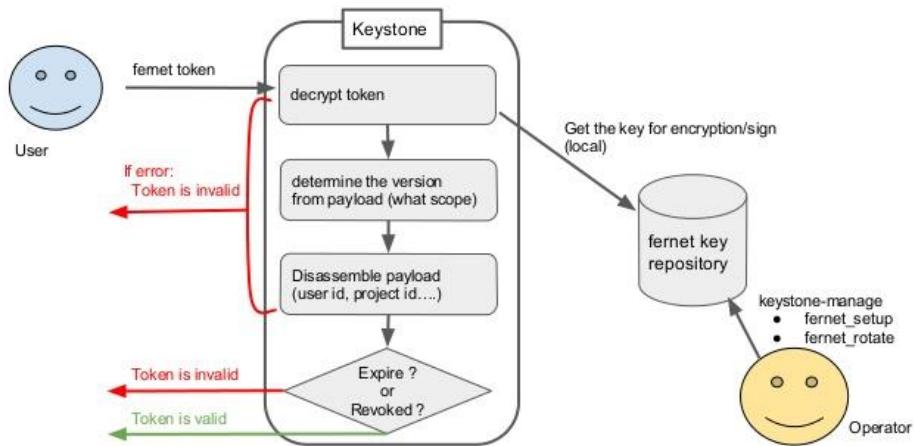
Memcached is an open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.



è solo un recap di cose già dette prima

Token Types and Authentication

Fernet Token Validation





Token Types and Authentication

- Even though fernet tokens operate very similarly to UUID tokens, they do not require persistence or leverage the configured token persistence driver in any way.
 - The keystone token database no longer suffers bloat as a side effect of authentication. Pruning expired tokens from the token database is no longer required when using fernet tokens. Because fernet tokens do not require persistence, they do not have to be replicated. As long as each keystone node shares the same key repository, fernet tokens can be created and validated instantly across nodes.
- The arguments for using fernet over PKI and PKIZ remain the same as UUID, in addition to the fact that fernet tokens are much smaller than PKI and PKIZ tokens. PKI and PKIZ tokens still require persistent storage and can sometimes cause issues due to their size. Fernet tokens are kept under a 250 byte limit. PKI and PKIZ tokens typically exceed 1600 bytes in length.



JSON Web Signature (JWS) token

- Implemented in the Stein release. **JWS tokens are signed**, meaning the **information used to build the token ID** is **not opaque** to users and can **it can be decoded by anyone**.
- JWS tokens are **ephemeral**, or non-persistent, which means they won't bloat the database or require replication across nodes.
- Tokens are **signed with private keys** and **validated with public keys**. The JWS token provider implementation only supports the ES256 JSON Web Algorithm (JWA), which is an Elliptic Curve Digital Signature Algorithm (ECDSA) using the P-256 curve and a SHA-256 hash algorithm.

JSON web token (JWT), pronounced "jot", is an open standard ([RFC 7519](#)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. Because of its relatively small size, a JWT can be sent through a URL, through a POST parameter, or inside an HTTP header, and it is transmitted quickly. A JWT contains all the required information about an entity to avoid querying a database more than once. The recipient of a JWT also does not need to call a server to validate the token.



JSON Web Signature (JWS) token

- A deployment might consider using JWS tokens as **opposed to fernet tokens** if there are security concerns about key sharing. Remember that key distribution is only required in multi-node keystone deployments. If you only have one keystone node serving requests in your deployment, key distribution is unnecessary.
- Note that a major difference between the two providers is that JWS tokens are not opaque and can be decoded by anyone with the token ID. Fernet tokens are opaque in that the token ID is ciphertext.
- Despite the JWS token payload being readable by anyone, keystone reserves the right to make backwards incompatible changes to the token payload itself, which is not an API contract. It is recommended validating the token against keystone's authentication API to inspect its associated metadata.

A deployment might consider using JWS tokens as opposed to fernet tokens if there are security concerns about sharing symmetric encryption keys across hosts. Note that a major difference between the two providers is that JWS tokens are not opaque and can be decoded by anyone with the token ID. Fernet tokens are opaque in that the token ID is ciphertext. Despite the JWS token payload being readable by anyone, keystone reserves the right to make backwards incompatible changes to the token payload itself, which is not an API contract. We only recommend validating the token against keystone's authentication API to inspect its associated metadata.



The Catalog Service (1/2)

- The OpenStack keystone service **catalog** allows API clients to dynamically discover and navigate through cloud services.
- The service catalog may differ from deployment-to-deployment, user-to-user, and project-to-project.
- The Catalog service provides an endpoint registry used for endpoint discovery

60



The Catalog Service (2/2)

```
File Edit Tabs Help
studente@fondamenti: ~ $ openstack catalog list
WARNING: Failed to import plugin clustering.

+-----+-----+-----+
| Name | Type | Endpoints |
+-----+-----+-----+
| glance | image | RegionOne
|         |       |   public: http://controller:9292
|         |       | RegionOne
|         |       |   internal: http://controller:9292
|         |       | RegionOne
|         |       |   admin: http://controller:9292
| keystone | identity | RegionOne
|           |       |   public: http://controller:5000/v3/
|           |       | RegionOne
|           |       |   admin: http://controller:5000/v3/
|           |       | RegionOne
|           |       |   internal: http://controller:5000/v3/
| heat | orchestration | RegionOne
|       |       |   internal: http://controller:8004/v1/0bf1e67c865b4dc9a909a300a5a0497c
|       |       | RegionOne
|       |       |   public: http://controller:8004/v1/0bf1e67c865b4dc9a909a300a5a0497c
|       |       | RegionOne
|       |       |   admin: http://controller:8004/v1/0bf1e67c865b4dc9a909a300a5a0497c
| aodh | alarming | RegionOne
|       |       |   public: http://controller:8042
|       |       | RegionOne
|       |       |   internal: http://controller:8042
|       |       | RegionOne
|       |       |   admin: http://controller:8042
| placement | placement | RegionOne
|           |       |   internal: http://controller:8778
|           |       | RegionOne
|           |       |   public: http://controller:8778
|           |       | RegionOne
|           |       |   admin: http://controller:8778
| neutron | network | RegionOne
|           |       |   public: http://controller:9696
+-----+-----+-----+
```



The Policy Service

- The Policy service provides a rule-based authorization engine and the associated rule management interface.
- Each OpenStack service defines the access policies for its resources in an associated policy file. A resource, for example, could be API access, the ability to attach to a volume, or to fire up instances. The policy rules are specified in JSON format and the file is called policy.json.

62

Each OpenStack service, Identity, Compute, Networking, and so on, has its own role-based access policies. They determine which user can access which objects in which way, and are defined in the service's policy.json file.

Whenever an API call to an OpenStack service is made, the service's policy engine uses the appropriate policy definitions to determine if the call can be accepted. Any changes to policy.json are effective immediately, which allows new policies to be implemented while the service is running.

A policy.json file is a text file in JSON (Javascript Object Notation) format. Each policy is defined by a one-line statement in the form "<target>" : "<rule>".

The Policy Service

```
/usr/lib/python2.7/dist-packages/cinder/tests/unit/policy.json  
/usr/lib/python2.7/dist-packages/glance/tests/etc/policy.json  
/usr/lib/python2.7/dist-packages/gnocchi/rest/policy.json  
/usr/lib/python2.7/dist-packages/neutron/tests/etc/policy.json  
/usr/lib/python2.7/dist-packages/neutron_lib/tests/etc/no_policy.json  
/usr/lib/python2.7/dist-packages/neutron_lib/tests/etc/policy.json  
/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/keystone_policy.json  
/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/no_default_policy.json  
/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/nova_policy.json  
/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/with_default_policy.json  
/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/cinder_policy.json  
/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/glance_policy.json  
/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/keystone_policy.json  
/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/neutron_policy.json  
/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/nova_policy.json
```

63

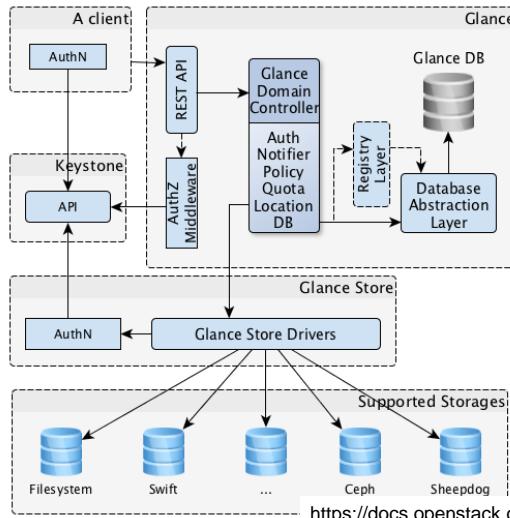
Try:

```
$ locate policy.json
```



OpenStack Glance: Image Service

Glance is an OpenStack project that provides a service where users can upload and discover data assets that are meant to be used with other services. This currently includes **images and metadata** definitions. It implements OpenStack's **Images API v2**



- **Core Use Cases:**
 - Glance provisions and manages images of Virtual Machines (VMs).
- **Key Capabilities:**
 - Glance image services include discovering, registering, and retrieving virtual machine (VM) images.
 - Glance has a RESTful API that allows querying of VM image metadata as well as retrieval of the actual image.

OpenStack Glance has a **client-server architecture** that provides the user with a REST API. A **Glance Domain Controller** manages the internal server operations that is divided into three main components: **AuthN**, **AuthZ**, and **Glance Store**. All the file (Image data) operations are performed using **glance_store** library, which is responsible for interacting with various storage backends. Glance uses a central database (**Glance DB**) that is shared amongst all the components.

Following components are present in the Glance architecture:

A client - any application that makes use of a Glance server.

REST API - Glance functionalities are exposed via REST.

Database Abstraction Layer (DAL) - an application programming interface (API) that interacts with the Glance Domain Controller.

Glance Domain Controller - middleware that implements the main Glance functionalities.

Glance Store - used to organize interactions between Glance and various data stores.

Registry Layer - optional layer that is used to organise secure communication between Glance and its external clients.

Authn (authentication) primarily deals with user identity: who is this person? Is she who she claims to be?

Authz (authorization) performs the authentication token validation and retrieves actual user information.

The **domain model** contains the following layers:

[Authorization](#)

[Property protection](#)

[Notifier](#)

[Policy](#)

[Quota](#)

[Location](#)

[Database](#)

Authorization

The first layer of the domain model provides a verification of whether an image itself or its property can be changed. An admin or image owner can apply the changes. The information about a user is taken from the request context and is compared with the image owner. If the user cannot apply a change, a corresponding error message appears.

Property protection

The second layer of the domain model is optional. It becomes available if you set the `property_protection_file` parameter in the Glance configuration file.

There are two types of image properties in Glance:

Core properties, as specified in the image schema

Meta properties, which are the arbitrary key/value pairs that can be added to an image

The property protection layer manages access to the meta properties through Glance's public API calls. You can restrict the access in the property protection configuration file.

Notifier

On the third layer of the domain model, the following items are added to the message queue:

Notifications about all of the image changes

All of the exceptions and warnings that occurred while using an image

Policy

The fourth layer of the domain model is responsible for:

Defining access rules to perform actions with an image. The rules are defined in the `etc/policy.yaml` file.

Monitoring of the rules implementation.

Quota

On the fifth layer of the domain model, if a user has an admin-defined size quota for all of his uploaded images, there is a check that verifies whether this quota

exceeds the limit during an image upload and save:

If the quota does not exceed the limit, then the action to add an image succeeds.

If the quota exceeds the limit, then the action does not succeed and a corresponding error message appears.

Location

The sixth layer of the domain model is used for interaction with the store via the glance_store library, like upload and download, and for managing an image location. On this layer, an image is validated before the upload. If the validation succeeds, an image is written to the glance_store library.

This sixth layer of the domain model is responsible for:

Checking whether a location URI is correct when a new location is added

Removing image data from the store when an image location is changed

Preventing image location duplicates

Database

On the seventh layer of the domain model:

The methods to interact with the database API are implemented.

Images are converted to the corresponding format to be recorded in the database. And the information received from the database is converted to an Image object.

Glance Components

- A **client** - any application that makes use of a Glance server.
- **REST API** - Glance capabilities are exposed via REST.
- **Database Abstraction Layer (DAL)** - an application programming interface (API) that unifies the communication between Glance and databases.
- **Glance Domain Controller** - middleware that implements the main Glance functions such as authorization, notifications, policies, database connections.
- **Glance Store** - used to organize interactions between Glance and various data stores.

Glance Components

- **Registry Layer** - optional layer that is used to organise secure communication between the domain and the DAL by using a separate service.
- **Database:** Stores image metadata and you can choose your database depending on your preference. Most deployments use MySQL or SQLite.



Glance General Features

- VM images made available through Glance can be stored in a variety of locations from simple filesystems to object-storage systems like the OpenStack Swift project.
- Images are uniquely identified by way of a URI that matches the following signature:
 - <Glance Server Location>/v2/images/<ID>
 - /var/lib/glance/images in case of filesystem backend.
- When you add an image to the Image service, you can specify its disk and container formats.
- You can set your image's disk format to one of the following:
 - raw, vhd, vhdx, vmdk, vdi, iso, ploop, qcow2, aki, ari, ami.

67

The raw image format is the simplest one, and is natively supported by both KVM and Xen hypervisors. You can think of a raw image as being the bit-equivalent of a block device file, created as if somebody had copied, say, /dev/sda to a file using the **dd** command.

You can set your image's disk format to one of the following:

raw

This is an unstructured disk image format

vhd

This is the VHD disk format, a common disk format used by virtual machine monitors from VMware, Xen, Microsoft, VirtualBox, and others

vhdx

This is the VHDX disk format, an enhanced version of the vhd format which supports larger disk sizes among other features.

vmdk

Another common disk format supported by many common virtual machine monitors

vdi

A disk format supported by VirtualBox virtual machine monitor and the QEMU emulator

iso

An archive format for the data contents of an optical disc (e.g. CDROM).

ploop

A disk format supported and used by Virtuozzo to run OS Containers

qcow2

A disk format supported by the QEMU emulator that can expand dynamically and supports Copy on Write

aki

This indicates what is stored in Glance is an Amazon kernel image

ari

This indicates what is stored in Glance is an Amazon ramdisk image

ami

This indicates what is stored in Glance is an Amazon machine image



Glance General Features

- The **container format** indicates whether the virtual machine image is in a file format that also contains metadata about the actual virtual machine.
- It is safe to simply specify **bare** as the container format if you are unsure.
- You can set your image's container format to one of the following:
 - **bare, ovf, aki, ari, ami, ova, docker.**

68

The container format refers to whether the virtual machine image is in a file format that also contains metadata about the actual virtual machine.

You can set your image's container format to one of the following:

bare

This indicates there is no container or metadata envelope for the image

ovf

This is the OVF container format. **Open Virtualization Format (OVF)** is an [open standard](#) for packaging and distributing [virtual appliances](#) or, more generally, [software](#) to be run in [virtual machines](#).

aki

This indicates what is stored in Glance is an Amazon kernel image

ari

This indicates what is stored in Glance is an Amazon ramdisk image

ami

This indicates what is stored in Glance is an Amazon machine image

ova

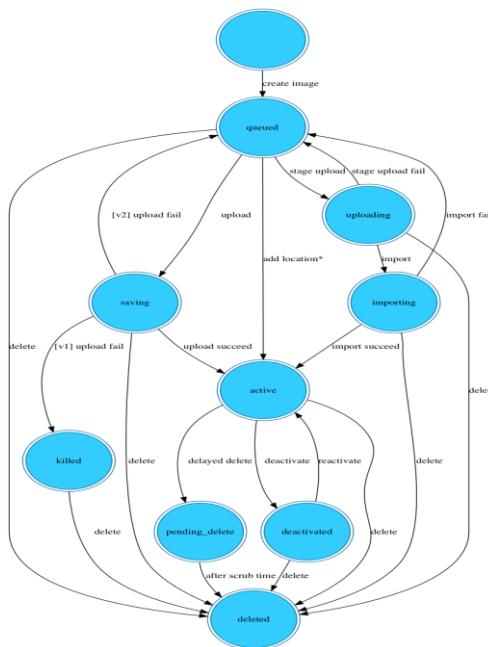
This indicates what is stored in Glance is an OVA tar archive file

docker

This indicates what is stored in Glance is a Docker tar archive of the container filesystem



Glance Image Statuses



69

queued: The image identifier has been reserved for an image in the Glance registry. No image data has been uploaded to Glance and the image size was not explicitly set to zero on creation.

saving: Denotes that an image's raw data is currently being uploaded to Glance. When an image is registered with a call to POST /images and there is an x-image-meta-location header present, that image will never be in the saving status (as the image data is already available in some other location).

uploading: Denotes that an import data-put call has been made. While in this status, a call to PUT /file is disallowed. (Note that a call to PUT /file on a queued image puts the image into saving status. Calls to PUT /stage are disallowed while an image is in saving status. Thus it's not possible to use both upload methods on the same image.)

importing: Denotes that an import call has been made but that the image is not yet ready for use.

active: Denotes an image that is fully available in Glance. This occurs when the image data is uploaded, or the image size is explicitly set to zero on creation.

deactivated: Denotes that access to image data is not allowed to any non-admin user. Prohibiting downloads of an image also prohibits operations like image export and image cloning that may require image data.

killed: Denotes that an error occurred during the uploading of an image's data, and that the image is not readable.

deleted: Glance has retained the information about the image, but it is no

longer available to use. An image in this state will be removed automatically at a later date.

pending_delete: This is similar to deleted, however, Glance has not yet removed the image data. An image in this state is not recoverable.

Glance Image Statuses

Status	Description
queued	The Image service reserved an image ID for the image in the catalog but did not yet upload any image data.
saving	The Image service is in the process of saving the raw data for the image into the backing store.
active	The image is active and ready for consumption in the Image service.
killed	An image data upload error occurred.
deleted	The Image service retains information about the image but the image is no longer available for use.
pending_delete	Similar to the deleted status. An image in this state is not recoverable.
deactivated	The image data is not available for use.
uploading	Data has been staged as part of the interoperable image import process. It is not yet available for use. (<i>Since Image API 2.6</i>)
importing	The image data is being processed as part of the interoperable image import process, but is not yet available for use. (<i>Since Image API 2.6</i>)



Glance Image Visibility

Visibility	Description
public	<p>Any user may read the image and its data payload. Additionally, the image appears in the default image list of all users.</p>
community	<p>Any user may read the image and its data payload, but the image does not appear in the default image list of any user other than the owner.</p>
shared	<p>The image is associated with an image members list. Only the owner and users in the image members list may read the image or its data payload. The image appears in the default image list of the owner. It also appears in the default image list of members who have accepted to be in the image member list. If you do not specify a visibility value when you create an image, it is assigned this visibility by default. Non-owners, however, will not have access to the image until they are added as image member list.</p>
private	<p>Only the owner image may read the image or its data payload. Additionally, the image appears in the owner's default image list.</p>

Visibility Semantics

Here are the semantics for visibility:

1.visibility is orthogonal to ownership

2.semantics for each of the values

1. **public**: all users:

1. have this image in default image-list
2. can see image-detail for this image
3. can boot from this image

2. **private**: users with tenantId == tenantId(owner) *only*:

1. have this image in the default image-list
2. see image-detail for this image
3. can boot from this image

3. **shared**:

1. users with tenantId == tenantId(owner)
 1. have this image in the default image-list
 2. see image-detail for this image
 3. can boot from this image
2. users with tenantId in the member-list of the image
 1. can see image-detail for this image

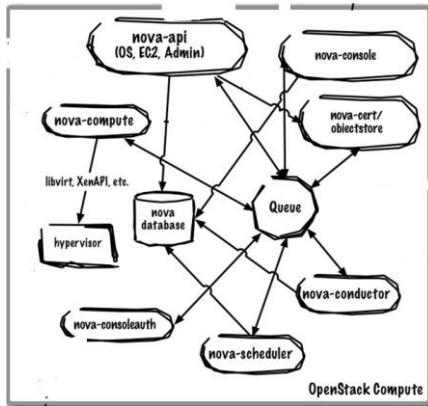
2. can boot from this image
 1. users with tenantId in the member-list with member_status == 'accepted'
 1. have this image in their default image-list
- 1. community:**
1. all users
 1. can see image-detail for this image
 2. can boot from this image
 2. users with tenantId in the member-list of the image with member_status == 'accepted'
 1. have this image in their default image-list

1.NOTE: it's possible for an image to have 'visibility' == 'shared' but have an empty member-list



OpenStack Nova

Nova is an OpenStack project that provides users with computing resources.
It implements OpenStack's **Compute API**



- **Core Use Cases:**

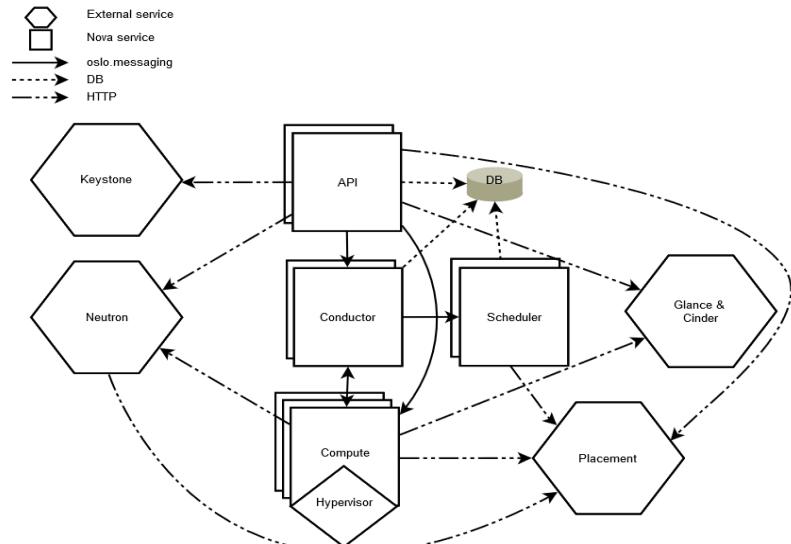
Nova provisions and manages Virtual Machines (VMs).

- **Key Capabilities:**

OpenStack Nova can work with several different hypervisors.



Nova Components



<https://docs.openstack.org/nova/latest/install/get-started-compute.html>

<https://docs.openstack.org/placement/latest/>

The placement API service was introduced in the 14.0.0 Newton release within the nova repository and extracted to the [placement repository](#) in the 19.0.0 Stein release. This is a REST API stack and data model used to track resource provider inventories and usages, along with different classes of resources. For example, a resource provider can be a compute node, a shared storage pool, or an IP allocation pool. The placement service tracks the inventory and usage of each provider. For example, an instance created on a compute node may be a consumer of resources such as RAM and CPU from a compute node resource provider, disk from an external shared storage pool resource provider and IP addresses from an external IP pool resource provider.

Two processes, nova-compute and nova-scheduler, host most of nova's interaction with placement.

Nova Components

- DB: sql database for data storage.
- API: component that receives HTTP requests, converts commands and communicates with other components via the **oslo.messaging** queue or HTTP.
- Scheduler: decides which host gets each instance.
- Compute: manages communication with hypervisor and virtual machines.
- Conductor: handles requests that need coordination (build/resize), acts as a database proxy, or handles object conversions.
- Placement: tracks resource provider inventories and usages (external service).

Nova Components

- Nova **SQL database**: Stores most build-time and run-time states for a cloud infrastructure, including:
 - Available instance types
 - Instances in use
 - Available networks
 - Projects

➤ Theoretically, OpenStack Compute can support any database that SQLAlchemy supports. Common databases are SQLite3 for test and development work, MySQL, MariaDB, and PostgreSQL.
- **nova-compute service**: A worker daemon that creates and terminates virtual machine instances through hypervisor APIs, for example for Xen, QEMU, Vmware.
- **nova-scheduler service**: Takes a virtual machine instance request from the queue and determines on which compute server host it runs.



Nova Components

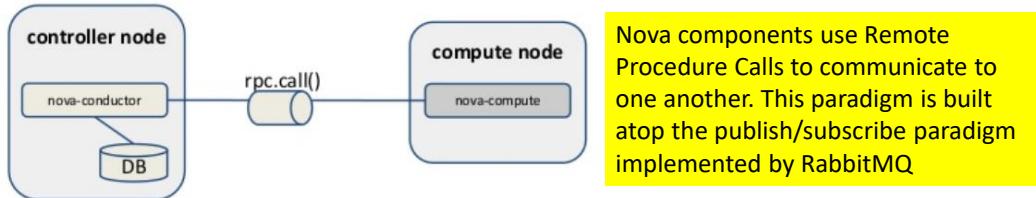
Nova consists of several components, which can typically run on different servers:

- **Message Queue**—central hub for passing messages between components. Uses Advanced Message Queuing Protocol (AMQP). By default uses **RabbitMQ**, can use Apache Qpid
- **Nova API**—handles OpenStack RESTful API, EC2, and admin interfaces.
 - **nova-api service**: Accepts and responds to end user compute API calls. Provides an endpoint, initiates running and instances, enforces some policies.
 - **nova-api-metadata** service: Accepts metadata requests from instances.
- **nova-conductor** — Mediates interactions between the nova-compute service and the database. It eliminates direct accesses to the cloud database made by the nova-compute service. Conceptually, it implements a new layer on top of nova-compute. The nova-conductor module scales horizontally.



Nova Components

Nova-conductor – compute exchange



Nova components use Remote Procedure Calls to communicate to one another. This paradigm is built atop the publish/subscribe paradigm implemented by RabbitMQ.

- Stateless RPC server. Acts as a proxy to the database.
- Eliminates remote DB access (security)
- Horizontal scalability; spawn multiple worker threads operating in parallel (performance)
- Hides DB implementation from the Nova Compute (upgrades)
- Beneficial for operations that cross multiple compute nodes (migration, resizes).

77

The nova-conductor service enables OpenStack to function without compute nodes accessing the database. Conceptually, it implements a new layer on top of nova-compute. It should not be deployed on compute nodes, or else the security benefits of removing database access from nova-compute are negated. Just like other nova services such as nova-api or nova-scheduler, it can be scaled horizontally. You can run multiple instances of nova-conductor on different machines as needed for scaling purposes.

The Nova conductor does not expose a REST API, but communicates with the other Nova components via RPC calls (based on RabbitMQ). The conductor is used to handle long-running tasks like building an instance or performing a live migration.

From <https://docs.openstack.org/nova/queens/reference/rpc.html>

Nova components (the compute fabric of OpenStack) use Remote Procedure Calls (RPC hereinafter) to communicate to one another; however such a paradigm is built atop the publish/subscribe paradigm so that the following benefits can be achieved:

- Decoupling between client and servant (such as the client does not need to know where the servant's reference is).
- Full a-synchronism between client and servant (such as the client does not need

the servant to run at the same time of the remote call).

- Random balancing of remote calls (such as if more servants are up and running, one-way calls are transparently dispatched to the first available servant).

- RabbitMQ broker stays between internal components of each service in OpenStack (e.g. nova-conductor, nova-scheduler in Nova service) and allow them to communicate each other in the loosely coupled way.

- RabbitMQ runs on controller as a process that will get the message from Invoker (API or scheduler) through `rpc.call()` or `rpc.cast()`.

Nova Message Exchange

OpenStack communicates between internal components using a **Message Queue and Advanced Message Queuing Protocol (AMQP)**. This uses a **publish/subscribe** message paradigm Message-Oriented Middleware.

- Messages can be distributed from one producer to many consumers, or from many producers to many consumers. For example, all consumers that subscribe to a particular topic (channel) will receive the message.
- An AMQP message broker handles communication between any two Nova components and messages are queued.
- This means that the Nova components are only loosely coupled.



Placement Service

- The placement API service was introduced in the 14.0.0 Newton release within the nova repository and extracted to the [placement repository](#) in the 19.0.0 Stein release.
- In a fully installed system, OpenStack services can register as **resource providers**. Each provider offers a certain set of resource classes, which is called an **inventory**.
- This is a REST API stack and data model used to track resource provider inventories and usages, along with different classes of resources.
 - For example, a resource provider can be a compute node, a shared storage pool, or an IP allocation pool.
- **The placement service tracks the inventory and usage of each provider.** For example, an instance created on a compute node may be a consumer of resources such as RAM and CPU from a compute node resource provider, disk from an external shared storage pool resource provider and IP addresses from an external IP pool resource provider.
- To link consumers and usage information, Placement uses **allocations**. An allocation represents a usage of resources by a specific consumer,

Before the Stein release the placement code was in Nova alongside the compute REST API code (nova-api). Now the placement API is a separate service and thus should be registered under a *placement* service type in the service catalog. Clients of placement, such as the resource tracker in the nova-compute node, will use the service catalog to find the placement endpoint.

The placement service provides an HTTP API <<https://docs.openstack.org/api-ref/placement/>> used to track resource provider inventories and usages, along with different classes of resources. For example, a resource provider can be a compute node, a shared storage pool, or an IP allocation pool.

A list of all *resource classes is* known to Placement. Resource classes are types of resources that Placement manages, like IP addresses, vCPUs, disk space or memory. In a fully installed system, OpenStack services can register as *resource providers*. Each provider offers a certain set of resource classes, which is called an *inventory*. A compute node, for instance, would typically provide CPUs, disk space and memory.

For each resource provider, Placement also maintains *usage data* which keeps track of the current usage of the resources.

To link *consumers* and usage information, Placement uses *allocations*. An allocation represents a usage of resources by a specific consumer,

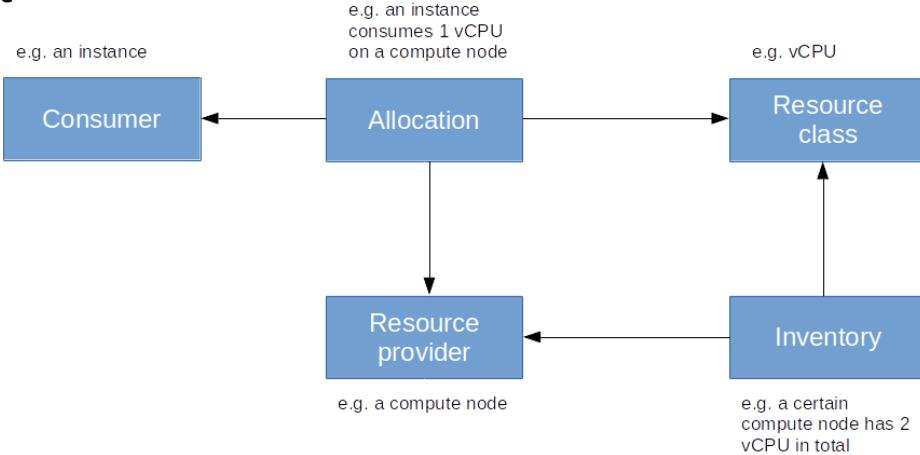
Relation between Nova and Placement. As we have mentioned above, compute nodes represent resource providers in Placement, so Nova needs to register resource provider records for the compute nodes it manages and providing the inventory information. When a new instance is created, the Nova scheduler will request information on inventories and current usage from Placement to determine the compute node on which the instance will be placed, and will subsequently update the allocations to register itself as a consumer for the resources consumed by the newly created

instance.



Placement Service

Example





OpenStack Cells

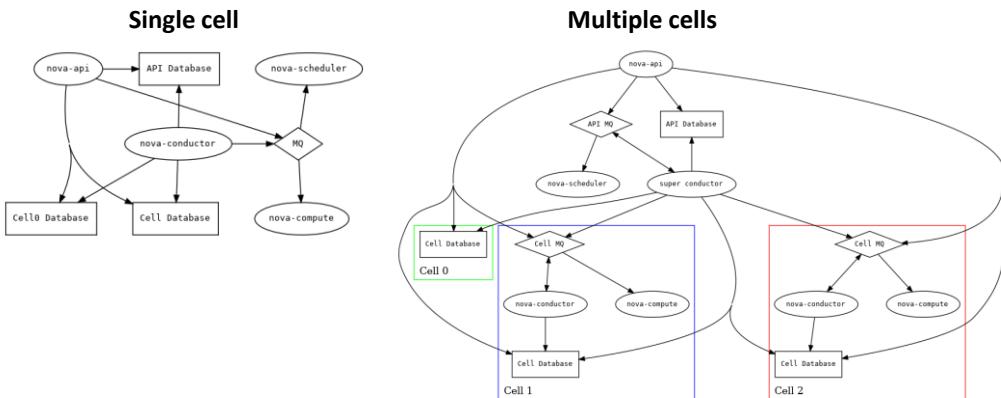
- Historically, Nova has depended on a single logical database and message queue that all nodes depend on for communication and data persistence. This becomes an issue for deployers as scaling and providing fault tolerance for these systems is difficult.
- An experimental feature in Nova called “cells”, is used by some large deployments to partition compute nodes into smaller groups, coupled with a database and queue.
- When this functionality is enabled, the hosts in an OpenStack Compute cloud are partitioned into groups called cells. Cells are configured as a tree.
- The purpose of the cells functionality in nova is to allow larger deployments to **shard** their many compute nodes into cells. All nova deployments are by definition cells deployments, even if most will only ever have a single cell. This means a multi-cell deployment will not be radically different from a “standard” nova deployment.

The idea behind this is that the set of your compute nodes are partitioned into cells. Every compute node is part of a cell, and in addition to these regular cells, there is a cell called cell0 (which is usually not used and only holds instances which could not be scheduled to a node). The Nova database schema is split into a global part which is stored in a database called the API database and a cell-local part. This cell-local database is different for each cell, so each cell can use a different database running (potentially) on a different host. A similar sharding applies to message queues. When you set up a compute node, the configuration of the database connection and the connection to the RabbitMQ service determine to which cell the node belongs. The compute node will then use this database connection to register itself with the corresponding cell database, and a special script (nova-manage) needs to be run to make these hosts visible in the API database as well so that they can be used by the scheduler.



OpenStack Cells

The top-level cell should have a host that runs a nova-api service, but no nova-compute services. Each child cell should run all of the typical nova-* services in a regular Compute cloud except for nova-api. You can think of cells as a normal Compute deployment in that each cell has its own database server and message queue broker.



<https://docs.openstack.org/nova/latest/admin/cells.html>

MQ: Message Queue

A “cell database” which is used by API, conductor and compute services, and which houses the majority of the information about instances.

A “cell0 database” which is just like the cell database, but contains only instances that failed to be scheduled. This database mimics a regular cell, but has no compute nodes and is used only as a place to put instances that fail to land on a real compute node (and thus a real cell).

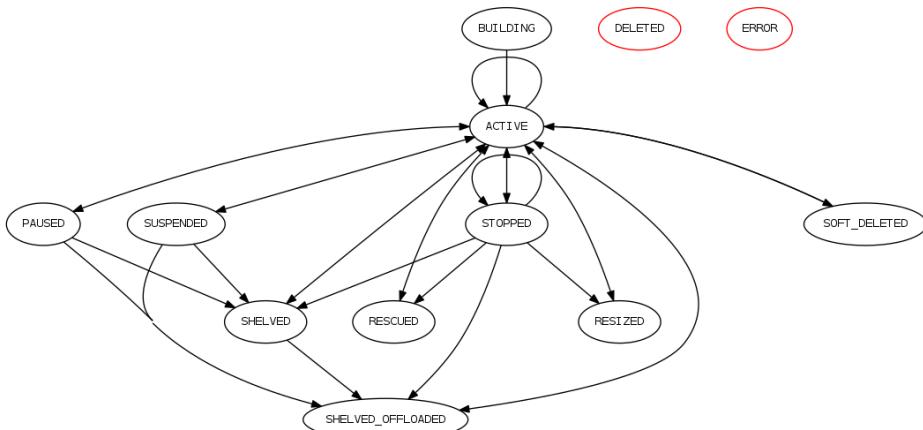
A message queue which allows the services to communicate with each other via RPC.

In larger deployments, we can opt to shard the deployment using multiple cells. In this configuration there will still only be one global API database but there will be a cell database (where the bulk of the instance information lives) for each cell, each containing a portion of the instances for the entire deployment within, as well as per-cell message queues and per-cell nova-conductor instances. There will also be an additional nova-conductor instance, known as a super conductor, to handle API-level operations.



NON VANNO SAPUTI A MEMORIA, solo il concetto generale
di questi stati transitori tra ACTIVE & ERROR

Virtual Machine States and Transitions



All states are allowed to transition to DELETED and ERROR.

<https://docs.openstack.org/nova/rocky/reference/vm-states.html>

<https://docs.openstack.org/nova/rocky/reference/vm-states.html>

Virtual Machine (Server) States and Transitions

- **INITIALIZED**: VM is just created in the database, but has not been built. (was BUILDING)
- **ACTIVE**: VM is running with the specified image.
- **RESCUED**: VM is running with the rescue image.
- **PAUSED**: VM is paused with the specified image.
- **SUSPENDED**: VM is suspended with the specified image, with a valid memory snapshot.
- **STOPPED**: VM is not running, and the image is on disk.
- **SOFT_DELETED**: VM is no longer running on compute, but the disk image remains and can be brought back. The server is marked as deleted but will remain in the cloud for some configurable amount of time. While soft-deleted, an authorized user can restore the server back to normal state. When the time expires, the server will be deleted permanently
- **HARD_DELETED**: From quota and billing's perspective, the VM no longer exists. VM will eventually be destroyed running on compute, disk images too.
- **RESIZED**: The VM is stopped on the source node but running on the destination node. The VM images exist at two locations (src and dest, with different sizes). The user is expected to confirm the resize or revert it.

<https://docs.openstack.org/nova/rocky/reference/vm-states.html>

Virtual Machine (Server) States and Transitions

- **ERROR:** some unrecoverable error happened. Only delete is allowed to be called on the VM.
- **SHELVED:** The server is in shelved state. Depends on the shelve offload time, the server will be automatically shelved off loaded.
- **SHELVED_OFFLOADED:** The shelved server is offloaded (removed from the compute host) and it needs unshelved action to be used again.

<https://docs.openstack.org/nova/rocky/reference/vm-states.html>

Shelving is useful if you have an instance that you are not using, but would like retain in your list of servers. For example, you can stop an instance at the end of a work week, and resume work again at the start of the next week. All associated data and resources are kept; however, anything still in memory is not retained. If a shelved instance is no longer needed, it can also be entirely removed.

You can run the following shelving tasks:

- Shelve an instance - Shuts down the instance, and stores it together with associated data and resources (a snapshot is taken if not volume backed). Anything in memory is lost.

\$ nova shelve SERVERNAME

Note By default, the **nova shelve** command gives the guest operating system a chance to perform a controlled shutdown before the instance is powered off. The shutdown behavior is configured by the `shutdown_timeout` parameter that can be set in the `nova.conf` file. Its value stands for the overall period (in seconds) a guest operation system is allowed to complete the shutdown. The default timeout is 60 seconds. See [Description of Compute configuration options](#) for details.

The timeout value can be overridden on a per image basis by means of

`os_shutdown_timeout` that is an image metadata setting allowing different types of operating systems to specify how much time they need to shut down cleanly.

- Unshelve an instance - Restores the instance.

```
$ nova unshelve SERVERNAME
```

- Remove a shelved instance - Removes the instance from the server; data and resource associations are deleted. If an instance is no longer needed, you can move the instance off the hypervisor in order to minimize resource usage.

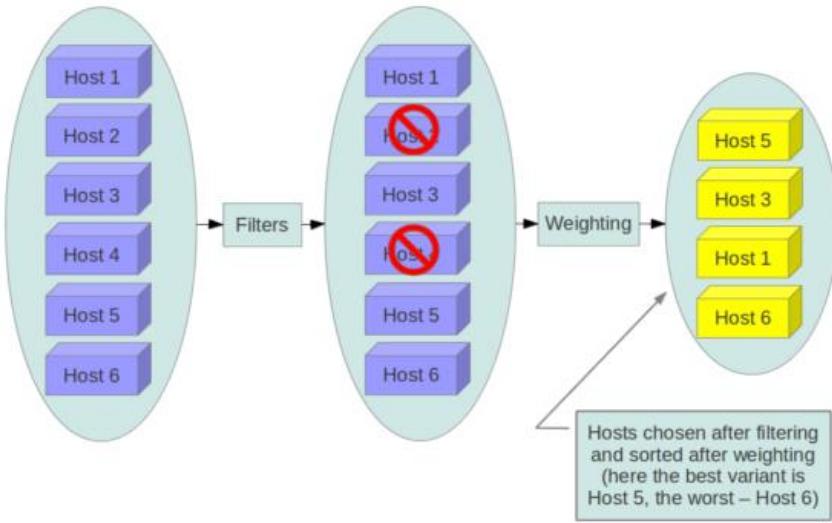
```
$ nova shelve-offload SERVERNAME
```

Task_state

- task_state should represent a transition state, and is precisely associated with one compute API, indicating which task the VM is currently running
- None: no task is currently in progress, BUILDING, IMAGE_SNAPSHOTTING, IMAGE_BACKINGUP, UPDATING_PASSWORD, PAUSING, UNPAUSING, SUSPENDING, RESUMING, DELETING, STOPPING, STARTING, RESCUING, UNRESCUING, REBOOTING, REBUILDING, POWERING_ON, POWERING_OFF, RESIZING, RESIZE_REVERTING, RESIZE_CONFIRMING, SCHEDULING, BLOCK_DEVICE_MAPPING, NETWORKING, SPAWNING, RESIZE_PREP, RESIZE_MIGRATING, RESIZE_MIGRATED, RESIZE_FINISH



Nova Scheduler



87

<https://docs.openstack.org/nova/latest/user/filter-scheduler.html>

The **Filter Scheduler** supports *filtering* and *weighting* to make informed decisions on where a new instance should be created. This Scheduler supports working with Compute Nodes only.

During its work Filter Scheduler iterates over all found compute nodes, evaluating each against a set of filters. The list of resulting hosts is ordered by weighers. The Scheduler then chooses hosts for the requested number of instances, choosing the most weighted hosts. For a specific filter to succeed for a specific host, the filter matches the user request against the state of the host plus some extra magic as defined by each filter (described in more detail below).

If the Scheduler cannot find candidates for the next instance, it means that there are no appropriate hosts where that instance can be scheduled.

The Filter Scheduler has to be quite flexible to support the required variety of *filtering* and *weighting* strategies. If this flexibility is insufficient you can implement *your own filtering algorithm*.

There are many standard filter classes which may be used (`nova.scheduler.filters`):

`AllHostsFilter` - does no filtering. It passes all the available hosts.

`ImagePropertiesFilter` - filters hosts based on properties defined on the instance's image. It passes hosts that can support the properties specified on the image used by the instance.

`AvailabilityZoneFilter` - filters hosts by availability zone. It passes hosts matching the availability zone specified in the instance properties. Use a comma to specify multiple zones. The filter will then ensure it matches any zone specified.

`ComputeCapabilitiesFilter` - checks that the capabilities provided by the host compute service satisfy any extra specifications associated with the instance type. It passes hosts that can create the specified instance type.

Nova Scheduler

- The **nova-scheduler** handles scheduling for virtual machine instances. The **filter scheduler** is the default scheduler for scheduling virtual machine instances. It supports filtering and weighting to make informed decisions on where a new instance should be created.
 - When it receives a resource request, it first applies filters to determine which host is eligible—the host is either accepted or rejected by a filter.



Nova Scheduler

Example filters include (among others):

- AvailabilityZoneFilter—in the requested availability zone
- ComputeFilter—passes all hosts that are running and enabled
- DiskFilter—is there sufficient disk space?
- CoreFilter—are there sufficient CPU cores available?
- RamFilter—does the host have sufficient RAM available?

Nova Scheduler

- By default, virtual machine instances are spread evenly across all available hosts, and hosts for new instances are chosen randomly from a set of the N best available hosts.
- Various options can change how this works, for example, one option in the nova.conf (configuration) file allows virtual machine instances to be stacked on one host until that host's resources are used up (set ram_weight_-multiplier in the config file to a negative number). Another option (scheduler_host_subset_size) selects the value N used to select the N best available hosts.



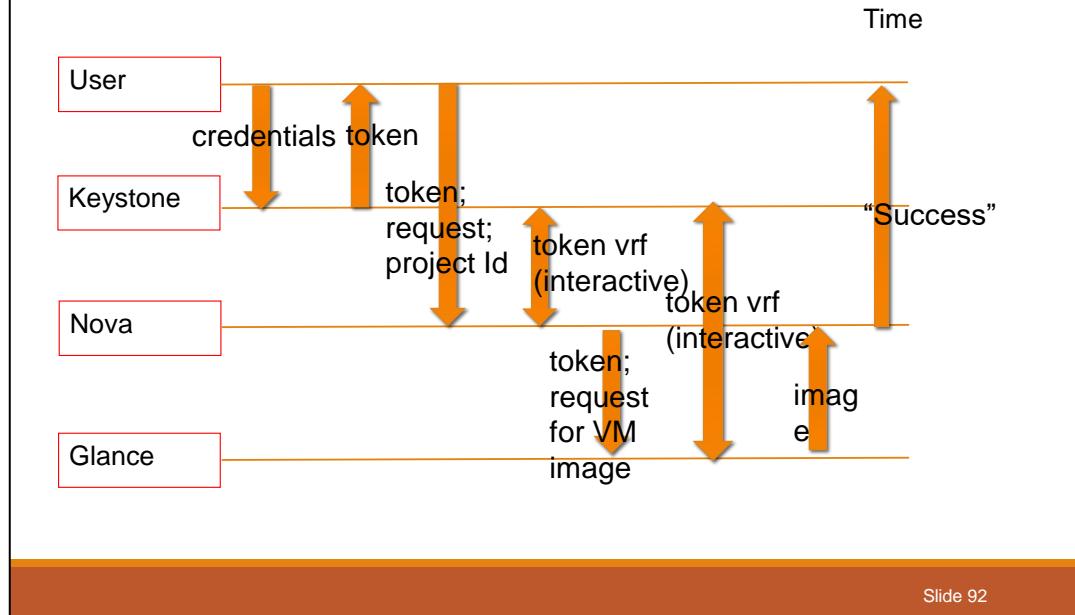
Nova Scheduler

- The filter scheduler weights each host in the list of available hosts. By default, it uses the **RAM Weigher** to determine which host to place the instance on. Using the RAM Weigher, the hosts with large quantities of RAM will be selected first, until you exceed a maximum number of VMs per node.
- If you disable the RAM Weigher, then VMs will be randomly distributed among available hosts (note that each host already was checked by a filter to determine that it had sufficient RAM to run the VM).



Basic---Launching an Instance

Creating/Running a VM without Networks (Neutron), without Persistent Storage (Cinder)

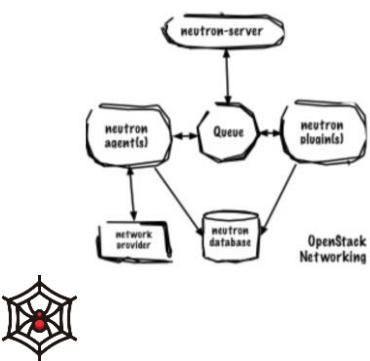


Triggered by a POST request to the /servers/API endpoint.



OpenStack Neutron

Neutron is an OpenStack project that provides “network connectivity as a service” between interface devices (e.g., vNICs) managed by other OpenStack services (e.g., Nova). It implements the OpenStack’s **Neutron API**



- **Core Use Cases:**

OpenStack Networking mainly interacts with OpenStack Compute to provide networks and connectivity for its instances.

- **Key Capabilities:**

Neutron allows users to configure their own network topology. It also allows use of advanced network services, including services intended to improve security and quality of service

OpenStack Neutron components communicate with each other using AMQP and a message queue.

<https://docs.openstack.org/neutron/zed/>

OpenStack Networking (neutron) allows you to create and attach interface devices managed by



Neutron Features

The following are basic concepts in Neutron:

- **Network**—A virtual object that can be created. It provides an independent network for each tenant in a multitenant environment. A network is equivalent to a switch with virtual ports which can be dynamically created and deleted.
- **Port**—A connection port. A router or a VM connects to a network through a port.
- **Subnet**—An address pool that contains a group of IP addresses. Two different subnets communicate with each other through a router.
- **Router**—A virtual router that can be created and deleted. It performs routing selection and data forwarding.

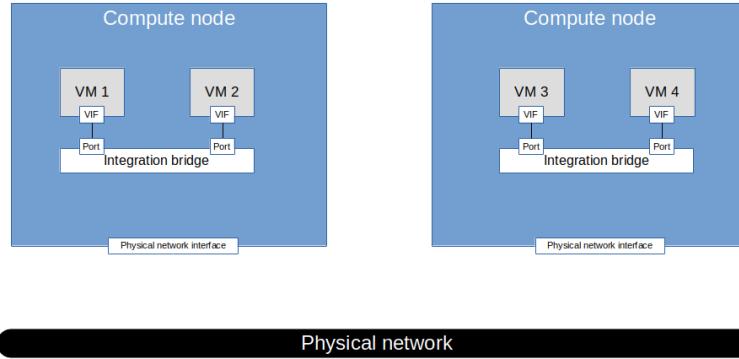
Neutron provides actual layer 2 connectivity to compute instances. Networks in Neutron are layer 2 networks, and if two compute instances are assigned to the same virtual network, they are connected to an actual virtual Ethernet segment and can reach each other on the Ethernet level.

Assume that we are given two virtual machines, call them VM1 and VM2, on the same physical compute node. Our hypervisor will attach a **virtual interface (VIF)** to each of these virtual machines. In a physical network, you would simply connect these two interfaces to ports of a switch to connect the instances. In our case, we can use a **virtual switch / bridge** to achieve this.

OpenStack is able to leverage several bridging technologies. First, OpenStack can of course use the Linux bridge driver to build and configure virtual switches. In addition, Neutron comes with a driver that uses Open vSwitch (OVS).



Neutron Features

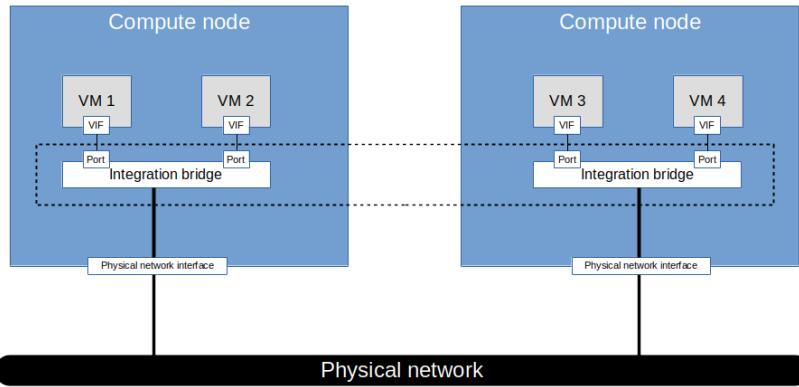


To connect the VMs running on the same host, Neutron could use (and it actually does) an OVS bridge to which the virtual machine networking interfaces are attached. This bridge is called the **integration bridge**. In a typical physical network, this bridge is also connected to a DHCP agent, routers and so forth.

Neutron Features

- But even for the simple case of VMs on the same host, we are not yet done. To operate a cloud at scale, you will need some approach to isolate networks.
- If, for instance, the two VMs belong to different tenants, you do not want them to be on the same network. To do this, Neutron uses **VLANs**. So the ports connecting the integration bridge to the individual VMs are tagged, and there is one VLAN for each Neutron network.

Neutron Features



We need to move on and connect the VMs that are attached to the same network on different hosts. To do this, we will have to use some **virtual networking technology** to connect the integration bridges on the different hosts

Neutron Features

- It is possible simply connect each integration bridge to a physical network device which in turn is connected to the physical network. With this setup, called a **flat network** in Neutron, all virtual machines are effectively connected to the same Ethernet segment. Consequently, there can only be one flat network per deployment.
- The second option we have is to use VLANs to partition the physical network. Neutron would assign a global VLAN ID to each virtual network (which in general is different from the VLAN ID used on the integration bridge) and tag the traffic with the corresponding VLAN ID.
- Finally, we could use tunnels to connect the integration bridges across the hosts. Neutron supports the most commonly used tunneling protocols (**VXLAN**, GRE, Geneve).

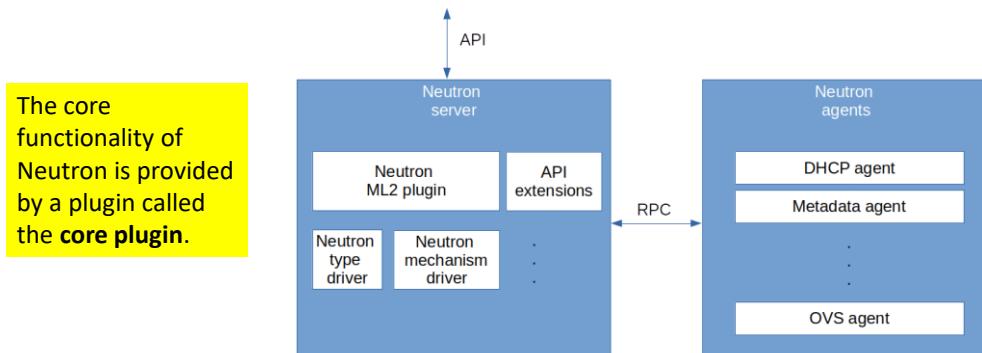



Neutron Architecture

- The Neutron architecture includes the neutron server and the neutron agents
- The main software entity for Neutron is the **neutron-server** daemon.
 - This is basically a python program that starts two critical components:
 - **Neutron REST service**
 - **Neutron Plugins** - core and service plugins
 - Accepts and routes **API requests** to the appropriate OpenStack Networking plug-in for action
- The **Neutron RPC service** is also started so that the neutron servers can communicate with the agents. And the RPC service actually loads the Neutron Plugin.



Neutron Architecture



The **Neutron server** on the left hand side provides the **Neutron API endpoint**. Then, there are the components that provide the actual functionality behind the API.

The Neutron API can be extended by API extensions. These extensions (which are again Python classes which are stored in a special directory and loaded upon startup) can be action extensions (which provide additional actions on existing resources), resource extensions (which provide new API resources) or request extensions that add new fields to existing requests.

Neutron comes with agents for additional functionality like DHCP, a metadata server or IP routing. In addition, there are agents running on the compute node to manipulate the network stack there, like the OVS agent or the Linux bridging agent, which correspond to the chosen mechanism drivers.



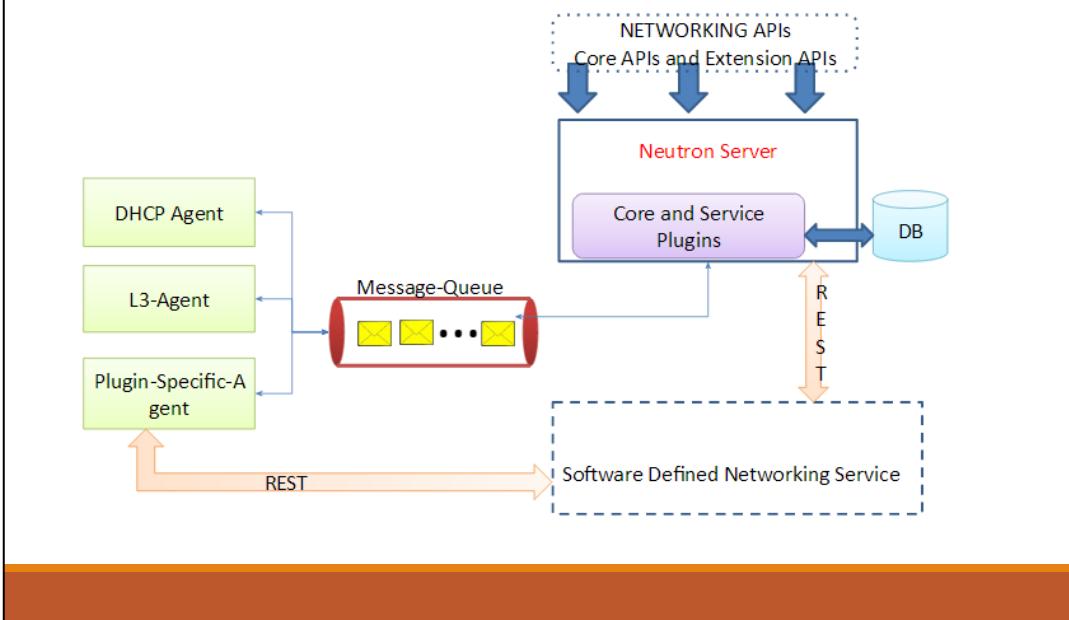
Neutron Architecture

Neutron has the following components:

- **Neutron plugins** (`neutron-*-plugin`). maintains configuration data and relationships between routers, networks, subnets, and ports in the Neutron database.
- **Plugins** are pluggable python classes that are invoked while responding to API requests. Neutron plugins are classified into **Core** and **Service plugins**. Core plugin primarily deals with L2 connectivity and IP Address management. On the other hand, Service plugins support services such as Routing (L3), firewall and load-balancing services etc.
- **Note: plugin code is executed as part of Neutron Server on the Controller node.**



Neutron Architecture



OpenStack Networking ships with plugins and agents for Cisco virtual and physical switches, NEC OpenFlow products, Open vSwitch, Linux bridging, and the VMware NSX product.

Plugins are pluggable python classes that are invoked while responding to API requests. Neutron plugins are classified into Core and Service plugins. Core plugin primarily deals with L2 connectivity and IP Address management. On the other hand, Service plugins support services such as Routing (L3), firewall and load-balancing services etc.

Each plug-in that Networking uses has its own concepts. While not vital to operating the VNI and

OpenStack environment, understanding these concepts can help you set up Networking. All Networking

installations use a core plug-in and a security group plug-in (or just the No-Op security group plug-in).

Additionally, Firewall-as-a-Service (FWaaS) is available

Agents

- Provides layer 2/3 connectivity to instances
- Handles physical-virtual network transition
- Handles metadata, etc.



Neutron Features

Neutron Plugins

- Plugins in Neutron allow **extension and/or customization** of the pre-existing functionality in Neutron. Networking vendors can write plugins that ensures **smooth inter-operability between OpenStack Neutron and vendor-specific software and hardware**. With this approach a rich set of physical and virtual networking resources can be made available to the virtual machines instances.

103

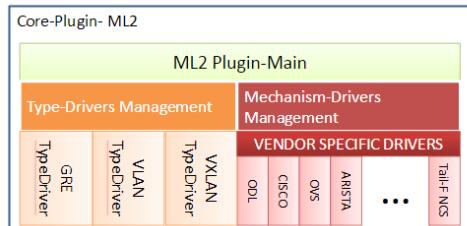
The DHCP agent provides DHCP services for virtual networks.
Configuration in /etc/neutron/dhcp_agent.ini

The metadata agent provides configuration information such as credentials to instances.
Configuration in /etc/neutron/metadata_agent.ini



ML2 Plugin and Drivers

- The ML2 or Modular Layer 2 plugin is bundled with OpenStack. It is an important **core plugin** because it supports wide variety of L2 technologies. More importantly the ML2 plugin allows multiple vendor technologies to co-exist.



- The ML2 plugin supports **Layer 2 technologies** such as VLAN, VXLAN and GRE etc. These technologies are referred to as Type drivers. And these technologies in turn can be implemented using various methods like Open vSwitch or via network hardware. ML2 Plugin allows different implementation methods using mechanism drivers.
- Although ML2 is the sole core plugin, it can support multiple **type drivers and mechanism drivers**. This model provides the flexibility and choice needed for tenants as well as leverage the hardware and software networking resources effectively.

Type Driver: which tells it what type of L2 technology to use when implementing the networking. For example, there is an option of using VLANs, VXLAN, or GRE Tunneling.

Mechanism Driver: which specifies what driver to use to implement the technology.

The ML2 plug-in uses the Linux bridge mechanism to build layer-2 (bridging and switching) virtual networking infrastructure for instances.

Configuration in

/etc/neutron/plugins/ml2/ml2_conf.ini

/etc/neutron/plugins/ml2/linuxbridge_agent.ini

ML2 Plugin

- The ML2 plugin allows OpenStack Networking to **simultaneously utilize the variety of layer 2 networking technologies** found in complex **real-world data centers**.
- Each available network type is managed by an ml2 **TypeDriver**.
 - TypeDrivers maintain any needed **type-specific network state**, and perform provider network validation and tenant network allocation. The ml2 plugin currently includes drivers for the **local**, **flat**, **vlan**, **gre** and **vxlan** network types.
- Each networking mechanism is managed by an ml2 **MechanismDriver**, responsible for taking the information established by the TypeDriver and ensuring that it is properly applied given the specific networking mechanisms enabled.

105

The Modular Layer 2 (ML2) neutron plug-in is a framework allowing OpenStack Networking to simultaneously use the variety of layer 2 networking technologies found in complex real-world data centers.

The ML2 framework distinguishes between the two kinds of drivers that can be configured:

- Type drivers

Define how an OpenStack network is technically realized. Example: VXLAN

Each available network type is managed by an ML2 type driver. Type drivers maintain any needed

type-specific network state. They validate the type specific information for provider networks and

are responsible for the allocation of a free segment in project networks.

- Mechanism drivers

Define the mechanism to access an OpenStack network of a certain type. Example: Open vSwitch

mechanism driver.

The mechanism driver is responsible for taking the information established by the type driver and

ensuring that it is properly applied given the specific networking mechanisms that have been enabled.

Mechanism drivers can utilize L2 agents (via RPC) and/or interact directly with external devices

or controllers.

Multiple mechanism and type drivers can be used simultaneously to access different ports of the same virtual network

To enable mechanism drivers in the ML2 plug-in, edit the /etc/neutron/plugins/ml2/ml2_conf.ini file on the neutron server.



ML2 Plugin

type driver / mech driver	Flat	VLAN	VXLAN	GRE
Open vSwitch	yes	yes	yes	yes
Linux bridge	yes	yes	yes	no
SRIOV	yes	yes	no	no
MacVTap	yes	yes	no	no
L2 population	no	no	yes	yes

vSphere supports **Single Root I/O Virtualization (SR-IOV)**. You can use SR-IOV for networking of virtual machines that are latency sensitive or require more CPU resources.

SR-IOV is a specification that allows a single Peripheral Component Interconnect Express (PCIe) physical device under a single root port to appear as multiple separate physical devices to the hypervisor or the guest operating system.

Macvtap is a new device driver meant to simplify virtualized bridged networking. It replaces the combination of the tun/tap and bridge drivers with a single module based on the macvlan device driver. A macvtap endpoint is a character device that largely follows the tun/tap ioctl interface and can be used directly by kvm/qemu and other hypervisors that support the tun/tap interface. The endpoint extends an existing network interface, the lower device, and has its own mac address on the same ethernet segment. Typically, this is used to make both the guest and the host show up directly on the switch that the host is connected to.

L2 population is a special mechanism driver that optimizes BUM (Broadcast, unknown destination address, multicast) traffic in the overlay networks VXLAN and GRE. It needs to be used in conjunction with either the Linux bridge or the Open vSwitch mechanism driver and cannot be used as standalone mechanism driver. For more information, see the *Mechanism drivers* section below.

Specialized

- Open source
- External open source mechanism drivers exist as well as the neutron integrated reference implementations. Configuration of those drivers is not part of this document. For example:
 - OpenDaylight
 - OpenContrail
- Proprietary (vendor)
- External mechanism drivers from various vendors exist as well as the neutron integrated reference implementations.



no

Vendor Specific Mechanism Drivers

The set of drivers included in the main Neutron distribution and supported by the Neutron community include:

[Open vSwitch](#)
[Cisco UCS/Nexus](#)
[Cisco Nexus1000v](#)
[Linux Bridge](#)
[Modular Layer 2](#)
[Nicira Network Virtualization Platform \(NVP\)](#)
[Ryu OpenFlow Controller](#)
[NEC OpenFlow](#)
[Big Switch Controller Plugin](#)
[Cloudbase Hyper-V](#)
[MidoNet](#)
[Brocade Neutron Plugin](#)
[PLUMgrid](#)
[Mellanox Neutron Plugin](#)
[Embrane Neutron Plugin](#)

- [IBM SDN-VE](#)
- [CPLANE NETWORKS](#)
- [Nuage Networks](#)
- [OpenContrail](#)
- [Lenovo Networking](#)
- [Avaya Neutron Plugin](#)

Additional plugins are available from other sources:

- [Extreme Networks Plugin](#)
- [Ruijie Networks Plugin](#)
- [Juniper Networks Neutron Plugin](#)
- [Calico Neutron Plugin \(docs\)](#)
- [BNC Plugin \(docs\)](#)



Service Plugins

- Historically, Neutron supported the following advanced services:
 - **FWaaS** (*Firewall-as-a-Service*): runs as part of the L3 agent.
 - **VPNaas** (*VPN-as-a-Service*): derives from L3 agent to add VPNaas functionality.
- **Service plugins** implement different additional services, some of them already mentioned, including load balancing (Load Balancing as a Service or **LBaas**), VPNs (**VPNaas**), firewalls (**FWaaS**), metering, Trunk.



no, se vuoi leggi ma nulla di nuovo

Neutron Agents

- While the Neutron server acts as the centralized controller, the actual networking related commands and configuration are executed on the Compute nodes and **agents** are the entities that implement the actual networking changes on these nodes. Agents receive messages and instructions from the Neutron server (via plugins or directly) on the message bus.
- Since Agents are responsible for the implementation of networking, they are **closely associated with specific technologies** and the corresponding plugins.
- **Plugin agent** (neutron-*-agent)—Processes data packets on virtual networks. The choice of plug-in agents depends on Neutron plug-ins. A plug-in agent interacts with the Neutron server and the configured Neutron plug-in through a message queue.
- **DHCP agent** (neutron-dhcp-agent)—Provides DHCP services for tenant networks.
- **L3 agent** (neutron-l3-agent)—Provides Layer 3 forwarding services to enable inter-tenant communication and external network access by "routers" that connect to L2 networks.
 - It is responsible for providing layer 3 and NAT forwarding to gain external access for virtual machines on tenant networks.



Nutron API Concepts

The Neutron v2 API manages three kind of entities:

- **Network**, representing isolated virtual Layer-2 domains; a network can also be regarded as a virtual (or logical) switch.
- **Subnet**, representing IPv4 or IPv6 address blocks from which IPs to be assigned to VMs on a given network are selected.
- **Port**, representing virtual (or logical) switch ports on a given network.
- **Router**, representing a virtual device taking care of L3 operations including floating IP.

All entities support the basic CRUD operations with POST/GET/PUT/DELETE verbs, and have an auto-generated unique identifier.

110

CRUD: Create Read Update Delete



Network Configuration

- To configure rich network topologies, you can create and configure **networks and subnets** and instruct other OpenStack services like Compute to attach virtual devices to ports on these networks.
- OpenStack Compute (**Nova**) is a consumer of OpenStack Networking to provide connectivity for its instances.
- There are two types of network, **project (or self-service)** and **provider** networks. It is possible to share any of these types of networks among projects as part of the network creation process.

Provider networks

- **Provider networks** offer layer-2 connectivity to instances with optional support for DHCP and metadata services. These networks connect, or map, to existing layer-2 networks in the data center, typically using VLAN (802.1q) tagging to identify and separate them.
- Provider networks generally offer simplicity, performance, and reliability at the cost of flexibility. By default only administrators can create or update provider networks because they require configuration of physical network infrastructure.

Self Service networks

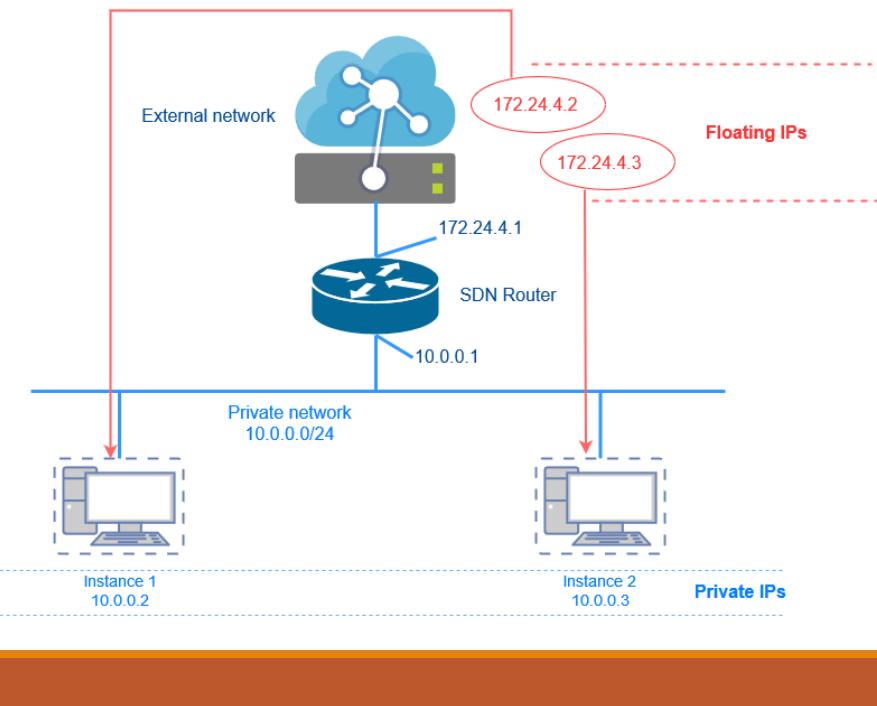
- Self-service networks primarily enable general (non-privileged) projects to manage networks without involving administrators. These networks are entirely virtual and require virtual routers to interact with provider and external networks such as the Internet. Self-service networks also usually provide DHCP and metadata services to instances.
- In most cases, self-service networks use overlay protocols such as VXLAN or GRE because they can support many more networks than layer-2 segmentation using VLAN tagging (802.1q).

Self Service networks

- IPv4 self-service networks typically use **private IP address** ranges and interact with provider networks via source NAT on virtual routers. **Floating IP** addresses enable access to instances from provider networks via destination NAT on virtual routers. **IPv6** self-service networks always use public IP address ranges and interact with provider networks via virtual routers with static routes.
- The Networking service implements routers using a layer-3 agent that typically resides at least one network node. Contrary to provider networks that connect instances to the physical network infrastructure at layer-2, self-service networks must traverse a layer-3 agent.



Floating IPs



Instance 1 and instance 2 have IP addresses that come from the private network (behind a NAT router). Everything that is behind the NAT router cannot be addressed directly so the 10.0.0.2 and 10.0.0.3 IP addresses cannot be accessed directly from the Internet.

That is why in OpenStack Software Defined Networking, to make instances accessible, these instances need to be allocated floating IP addresses.

Floating IPs are IP addresses exposed at the external side of the NAT router which is the SDN router. So, for external traffic to reach instance 1 and instance 2, the external traffic would go directly to the floating IP addresses of the instances. In the above example, traffic arrives 172.24.4.2 and 172.24.4.3 will be forwarded to 10.0.0.2 and 10.0.0.3 respectively.

Used for communication with networks outside the cloud, including the Internet
A floating IP address and a private IP address can be used at the same time on a NIC (Network Interface Card)

NOT automatically allocated to instances by default, they need to be attached to instances manually

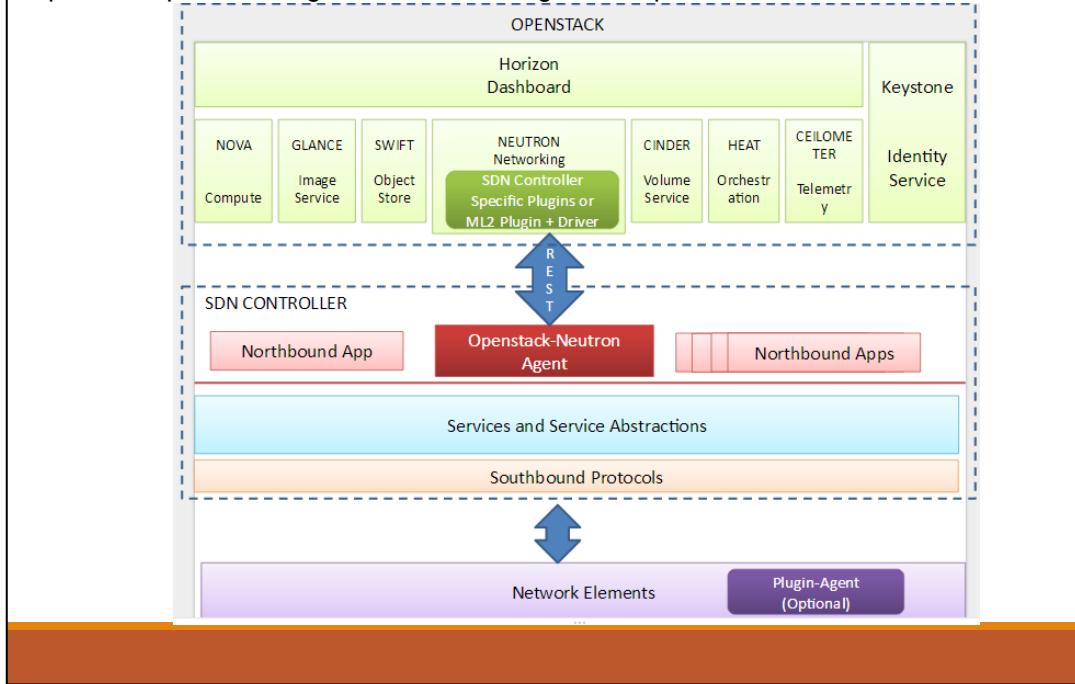
nah, già detto prima

OpenStack SDN

- [Softwared-defined networking](#) was introduced to both overcome the deficiencies of Neutron and to provide support for multiple network virtualization technologies (a centralized control plane creating isolated tenant [virtual networks](#)) and approaches. With the integration on SDN, Neutron is expected to support the dynamic nature of large-scale, high-density, multi-tenant cloud environments.
- OpenStack Neutron, with its plugin architecture, provides the ability to integrate SDN controllers into the OpenStack. This integration of SDN controllers into Neutron using plugins provides centralized management, and also facilitates the network programmability of OpenStack networking using APIs.
- SDN controllers like [OpenDaylight](#), [Ryu](#), and [Floodlight](#) use either specific plugins or the ML2 plugin with the corresponding mechanism drivers, to allow communication between Neutron and the SDN controller.

OpenStack SDN

<https://wiki.openstack.org/wiki/Neutron/OFAgent/ComparisonWithOVS>



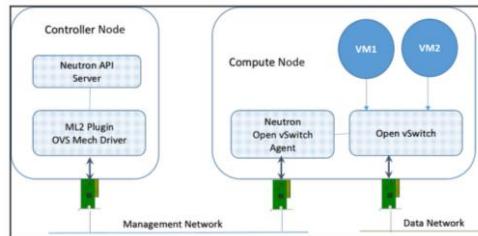
With this integration of OpenStack Neutron and SDN controllers, the changes to the network and network elements are also triggered from the OpenStack user, which are translated into Neutron APIs, and handled by neutron plugins and corresponding agents running in SDN controllers. For example, OpenDaylight interacts with Neutron by using the ML2 plugin present on the network node of Neutron via the REST API using northbound communication.

"Livello di dettaglio eccessivo, ma vabbè ormai ve lo dico"
[min 49 se ti interessa, del 09/04]



OpenStack SDN

Open Vswitch Plugin and Driver



- The **Neutron API server** receives commands via **REST API clients**, such as Horizon.
- The **API server** invokes the **ML2 Plugin** to process the request.
- The **ML2 Plugin** passes the request to the configured **OVS mechanism driver**.
- The **OVS mechanism driver** constructs an RPC message and directs it **towards the OVS agent** on the Compute Node over the Management Network Interface.
- The **OVS agent** in the Compute Node communicates with the **local OVS** instance to program it according to the instructions on the received command.
- Currently the OVS agent integrates the **RYU controller**.

From Newton on, by default Neutron uses the native interface of OVSDB and OpenFlow. The use of this interface allows Neutron to call ryu.base.appmanager during operation, and by default the native interface will have the Ryu controller listen on 127.0.0.1:6633 which can be however modified to point to another address. The integration among Ryu and OpenStack is very basic: however, it is impossible to load applications through the appmanager that is loaded by Neutron: therefore, the possible scenarios that can take place are infinite, because it simply needs the administrator to run the SDN application that he or she wants in order to manage the cloud platform in addition to or in a different way with respect to the default Neutron mechanisms.

Prior to OpenStack Newton, the neutron-openvswitch-agent used "ovs-ofctl" ofinterface driver by default to communicate with the Open vSwitch. From Newton on instead, the default implementation for ofinterface has become the novel "native", that mostly eliminates spawning ovs-ofctl to slightly improve the networking performance. This is an alternative OpenFlow implementation, implemented using Ryu SDN controller ofproto python library from Ryu SDN Framework. This solution indeed uses Ryu to inject OpenFlow rules when requested (e.g. a new virtual instance is being created or an old one is deleted) instead of building a "ovs-ofctl add/del-flow". The consequences are:

- The implemented OpenFlow rules are switched to OpenFlow 1.3, rather than OpenFlow 1.0;
- The OvS-agent acts as an OpenFlow controller perfectly integrated with the OpenStack platform;
- The OvS of the node is configured to connect to the controller.

90CHAPTER 4. NEUTRON NETWORKINGAmong the benefits, it is also possible to state:

- Reduction of the overhead related to the invocation of the ovs-ofctl command (and its associated rootwrap);
- Easier dealing with a future use of OpenFlow asynchronous messages(e.g. Packet-In, Port-Status, etc.)⇒Introduction of a SDN controllerinside the OpenStack platform, integration among the platform and theframework⇒Possibility to directly manage the OpenStack networkingby managing Ryu;
- Simpler XenAPI integration.



OpenStack Data Storage

- **Ephemeral**

- Each running VM receives some ephemeral storage, used to store the operating system of the image, and local data.
- The lifetime of the data in this storage is the lifetime of the VM.

- **Block Storage**

- It is performed by **Cinder**, which attaches storage volumes to a VM. A storage volume has its own file system. The server running Cinder can have storage volumes located on its physical disks, or can be located on other physical disks.

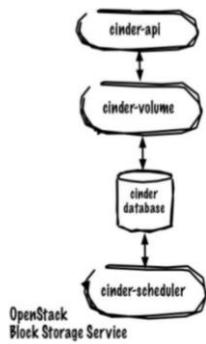
- **Object Storage**

- It is performed using **Swift**. It stores data as binary objects that are retrieved and written using HTTP commands (GET, PUT, etc.). Swift stores objects on object servers. Swift is best used for storing unstructured data, such as email, images, audio, and video, etc.



OpenStack Volume Storage: Cinder

- Cinder provides block storage service



- **Core Use Cases:**

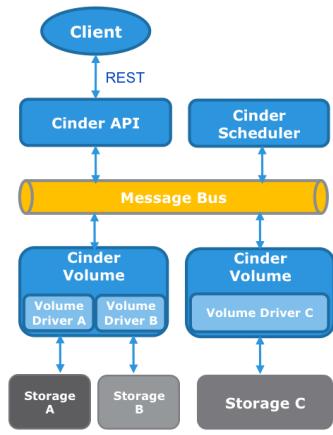
To implement services and libraries to provide on demand, self-service access to Block Storage resources.

- **Key Capabilities:**

It is designed to present storage resources to end users that can be consumed by the OpenStack Compute Project (Nova). The short description of Cinder is that it virtualizes the management of block storage devices and provides end users with a self service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device.



OpenStack Cinder



- All features of Cinder are exposed via a REST
- API that can be used to build more complicated logic or automation with Cinder. This can be consumed directly or via **various SDKs**.
- The **Block Storage API and scheduler services typically run on the controller nodes**. Depending upon the drivers used, the volume service can run on controller nodes, compute nodes, or standalone storage nodes.

Cinder Components

The Block Storage service consists of the following components:

- **cinder-api**
 - Accepts API requests, and routes them to the cinder-volume for action.
- **cinder-volume**
 - **manages Block Storage devices, specifically the back-end devices themselves.** The cinder-volume service responds to read and write requests sent to the Block Storage service to maintain state. It can interact with a variety of storage providers through a driver architecture.
 - Interacts directly with the other services and processes, such as the cinder-scheduler, through a message queue
- **cinder-scheduler daemon**
 - Selects the optimal storage provider node on which to create the volume. A similar component to the nova-scheduler. Depending upon your configuration, this may be simple round-robin scheduling to the running volume services, or it can be more sophisticated through the use of the Filter Scheduler.

Back-end Storage Devices - the Block Storage service requires some form of back-end storage that the service is built on. The **default implementation** is to use LVM on a local volume group named “cinder-volumes.” In addition to the base driver implementation, the Block Storage service also provides the means to add support for other storage devices to be utilized such as external Raid Arrays or other storage appliances. These back-end storage devices may have custom block sizes when using KVM or QEMU as the hypervisor.

Users and Tenants (Projects) - the Block Storage service can be used by many different cloud computing consumers or customers (tenants on a shared system), using role-based access assignments. Roles control the actions that a user is allowed to perform. In the default configuration, most actions do not require a particular role, but this can be configured by the system administrator in the cinder policy file that maintains the rules.

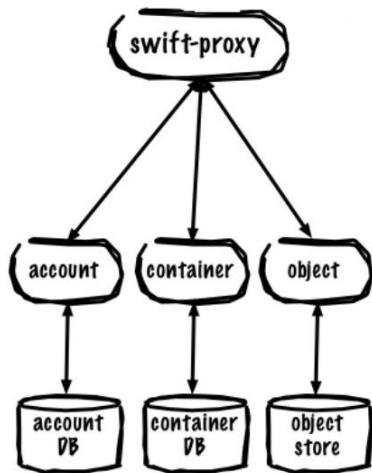
The Filter Scheduler is the default and enables filters on things like Capacity, Availability Zone, Volume Types, and Capabilities as well as custom filters.

Cinder Components

- **cinder-backup daemon**
 - The cinder-backup service provides backing up volumes of any type to a backup storage provider. Like the cinder-volume service, it can interact with a variety of storage providers through a driver architecture. The installation is optional.
- **Messaging queue**
 - Routes information between the Block Storage processes.



OpenStack Block Storage: Swift



This service provides a simple storage service for applications using [RESTful interfaces](#), providing maximum data availability and storage capacity.

Swift is composed of three major parts:

- **swift-proxy**,
- **storage servers**
 - account
 - container
 - object
- **consistency servers**

Swift Architecture: Proxy Server

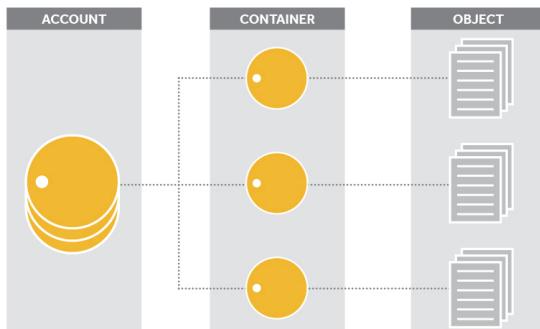
- The **proxy server** is an HTTP server that implements swift's RESTful API.
- As the **only** system in the swift cluster that **communicates with clients**, the proxy is responsible for coordinating with the storage servers and replying to the client with appropriate messages.



Swift Architecture: Storage Servers

In Swift 3 categories of things to store exist: **accounts**, **containers** and **objects**.

An **account** is a user account. An account contains **containers** (the equivalent of Amazon S3's **buckets**). Each container can contain user-defined key and values (just like a hash table or a dictionary): values are what Swift call **objects**.



A **storage URL** in Swift for an object looks like this:

`https://swift.example.com/v1/account/container/object`

A foundational premise of Swift is that requests are made via HTTP using a RESTful API. All requests sent to Swift are made up of at least three parts:

- **HTTP verb (e.g., GET, PUT, DELETE)**

- **Authentication information**

- **Storage URL**

- Optional: any data or metadata to be written

The HTTP verb provides the action of the request. I want to PUT this object into the cluster. I want to GET this account information out of the cluster, etc.

The authentication information allows the request to be fulfilled.

A storage URL in Swift for an object looks like this:

`https://swift.example.com/v1/account/container/object`

The storage URL has two basic parts: cluster location and storage location. This is because the storage URL has two purposes. It's the cluster address where the request should be sent and it's the location in the cluster where the requested action should take place.

Using the example above, we can break the storage URL into its two main parts:

- Cluster location: `swift.example.com/v1/`

- Storage location (for an object): `/account/container/object`

A storage location is given in one of three formats:

- `/account`

- The account storage location is a uniquely named storage area that contains the metadata (descriptive information) about the account itself as well as the list of containers in the account.
- Note that in Swift, an account is not a user identity. When you hear account, think storage area.

• /account/container

- The container storage location is the user-defined storage area within an account where metadata about the container itself and the list of objects in the container will be stored.

• /account/container/object

Swift Architecture: Storage Servers

The swift storage servers provide the on-disk storage for the cluster. There are **three types of storage servers in swift: account, container, and object**. Each of these servers provide an **internal RESTful API**.

- **Accounts server:** manages the accounts defined in the object storage service; Additionally, the account server provides a listing of the containers within an account
- **Container server:** manages the mapping of containers and folders within the service;
- **Object server:** manages objects and files on storage nodes.

Account Layer

The account server process handles requests regarding metadata for the individual accounts or the list of the containers within each account. This information is stored by the account server process in SQLite databases on disk.

Container Layer

The container server process handles requests regarding container metadata or the list of objects within each container. It's important to note that the list of objects doesn't contain information about the location of the object, simply that it belongs to a specific container. Like accounts, the container information is stored as SQLite databases.

Object Layer

The object server process is responsible for the actual storage of objects on the drives of its node. Objects are stored as binary files on the drive using a path that is made up in part of its associated partition and the operation's timestamp. The timestamp is important as it allows the object server to store multiple versions of an object. The object's metadata (standard and custom) is stored in the file's extended attributes (xattrs) which means the data and metadata are stored together and copied as a single unit.



Swift Architecture: Storage Servers

- The account and container servers provide **namespace partitioning and listing functionality**. They are implemented as databases (SQLite) on disk, and as other entities in Swift, they are replicated to multiple availability zones within the swift cluster.
- Swift is designed for **multi-tenancy**. Users are generally given access to a single swift account within a cluster, and they have complete control over that unique namespace.



Consistency Management

In theoretical computer science, the **CAP theorem** states that it is impossible for a **distributed data store** to **simultaneously** provide more than two out of the following three guarantees:

Consistency: Every read receives the most recent write or an error

Availability: Every request receives a (non-error) response, regardless it contains the most recent write

Partition tolerance: The system continues to operate despite any number of communication breakdowns between nodes in the system

Consistency Services

A key aspect of Swift is that it acknowledges that failures happen and is built to work around them. When account, container or object server processes are running on node, it means that data is being stored there. That means consistency services will also be running on those nodes to ensure the integrity and availability of the data.

Consistency Management

- Swift aims to **availability** and **partition tolerance** and dropped **consistency**. Hence it is always possible to get data, they will be dispersed on many places, but it is possible to receive an old version of them (or no data at all) in some odd cases (as server overload or failure).
- This compromise is made to allow maximum availability and scalability of the storage platform.
- For this reason, some mechanisms are built into Swift to minimize the potential data inconsistency window: they are responsible for data replication and consistency.



Consistency Management

- Storing data on disk and providing a RESTful API is quite easy. The hard part is handling failures. Swift's **consistency servers** are responsible for finding and correcting errors caused by both data corruption and hardware failures.
- Swift introduces three background processes to solve the problem of data consistency: **Auditor**, **Updater**, and **Replicator**.

The two main consistency services are auditors and replicators. There are also a number of specialized services that run in support of individual server process, e.g., the account reaper that runs where account server processes are running.

Consistency Management

- **Auditors** continually *scan the disks* to ensure that the data stored on disk has not suffered any bit or file system corruption. If an error is found, the corrupted object is moved to a quarantine area, and replication is responsible for replacing the data with a known good copy.
- **Updaters** ensure that *account and container listings are correct*. The *object updater* is responsible for keeping the object listings in the containers correct, and the *container updaters* are responsible for keeping the account listings up-to-date
- **Replicators** ensure that the *data stored in the cluster is where they should be* and that enough copies of the data exist in the system. Generally, replicators are responsible for repairing any corruption or degraded durability in the cluster.

Auditor is responsible for data auditing. It checks the integrity of Account, Container and Object by continuously scanning the disk. If data is found to be damaged, Auditor will isolate the file and obtain a complete copy from other nodes to replace it. , And the task of this replica is completed by the Replicator. In addition, in the rebalance operation of Ring, the Replicator is required to complete the actual data migration work. When the Object is deleted, the Replicator also completes the actual deletion operation.

Updater is responsible for handling those Account or Container update operations that fail due to insufficient load and other reasons. The Updater scans the Container or Object data on the local node, and then checks whether there are records of these data on the corresponding Account or Container node, and if not, pushes the records of these data to the Account or Container node. Only Container and Object have corresponding Updater processes, and there is no Account Updater process.

The implementation process of these three processes is similar. Here we take the Replicator process of Account as an example.

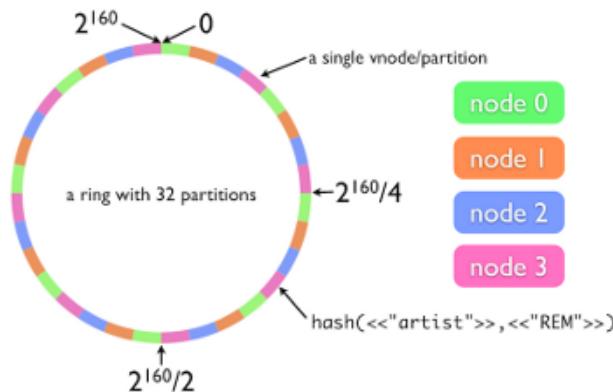
There are two types of Replicators in Swift: One is Database Replicator, for the two types of data in the form of a database, Account and Container, and the other is Object Replicator, which serves Object data.



Consistent hashing

Swift uses the principle of [consistent hashing](#). It builds what it calls a *ring*. A ring represents the space of all possible computed hash values divided in equivalent parts. Each part of this space is called a *partition*.

Example:



In order to store some objects and distribute them on 4 nodes, you would split your hash space in 4. You would have 4 partitions, and computing $hash(object) \bmod 4$ would tell you where to store your object: on node 0, 1, 2 or 3.

But since you want to be able to extend your storage cluster to more nodes without breaking the whole hash mapping and moving everything around, you need to build a lot more partitions. Let's say we're going to build 2^{10} partitions. Since we have 4 nodes, each node will have $2^{10} \div 4 = 256$ partitions. If we ever want to add a 5th node, it's easy: we just have to re-balance the partitions and move 1/5 of the partitions from each node to this 5th node. That means all our nodes will end up with $2^{10} \div 5 \approx 204$ partitions. We can also define a *weight* for each node, in order for some nodes to get more partitions than others.

With 2^{10} partitions, we can have up to 2^{10} nodes in our cluster.



Data duplication

To assure availability and partitioning, by default **Swift stores 3 copies of every objects**, but that's configurable.

In that case, we need to store each partition defined above not only on 1 node, but on 2 others. So Swift adds another concept: **zones**. A zone is an isolated space that does not depends on other zone, so in case of an outage on a zone, the other zones are still available. Concretely, a zone is likely to be a disk, a server, or a whole cabinet, depending on the size of your cluster. It's up to you to choose anyway.

Consequently, each partitions has not to be mapped to 1 host only anymore, but to N hosts. Each node will therefore store this number of partitions:

$$\begin{aligned} \text{number of partition stored on one node} &= \text{number of replicas} \\ &\times \text{total number of partitions} \div \text{number of node} \end{aligned}$$

Examples:

We split the ring in $2^{10} = 1024$ partitions. We have 3 nodes. We want 3 replicas of data.

- Each node will store a copy of the full partition space:
 $3 \times 2^{10} \div 3 = 2^{10} = 1024$ partitions.
- We split the ring in $2^{11} = 2048$ partitions. We have 5 nodes. We want 3 replicas of data.
- Each node will store $2^{11} \times 3 \div 5 \approx 1129$ partitions.
- We split the ring in $2^{11} = 2048$ partitions. We have 6 nodes. We want 3 replicas of data.
- Each node will store $2^{11} \times 3 \div 6 = 1024$ partitions.



Multiple Ring Operation

- Since in Swift 3 categories of things to store exist, **accounts**, **containers** and **objects**, Swift makes use of 3 different and independent rings to store its 3 kind of things (*accounts*, *containers* and *objects*).
- Internally, the two first categories are stored as [SQLite](#) databases, whereas the last one is stored using regular files.
- Note that these 3 rings can be stored and managed on 3 completely different set of servers.

OpenStack RESTful Interface

There are several different ways to access the OpenStack RESTful interface. Four of the most popular are:

- The OpenStack Dashboard.
- Using REST clients to send appropriate HTTP commands
- Using Python with the Python libraries provided with OpenStack
- Using the command-line clients
- Using HOT scripts

Example: Accessing Keystone for Authentication

CURL Command

```
# curl -i -H "Content-Type: application/json" \
-d @my_credentials
http://localhost:5000/v3/auth/tokens
```

my_credentials file:

```
{
  "auth": {
    "identity": {
      "methods": [
        "password"
      ],
      "password": {
        "user": {
          "name": "admin",
          "domain": {
            "id": "default"
          },
          "password": "mypassword"
        }
      }
    }
  }
}
```

137

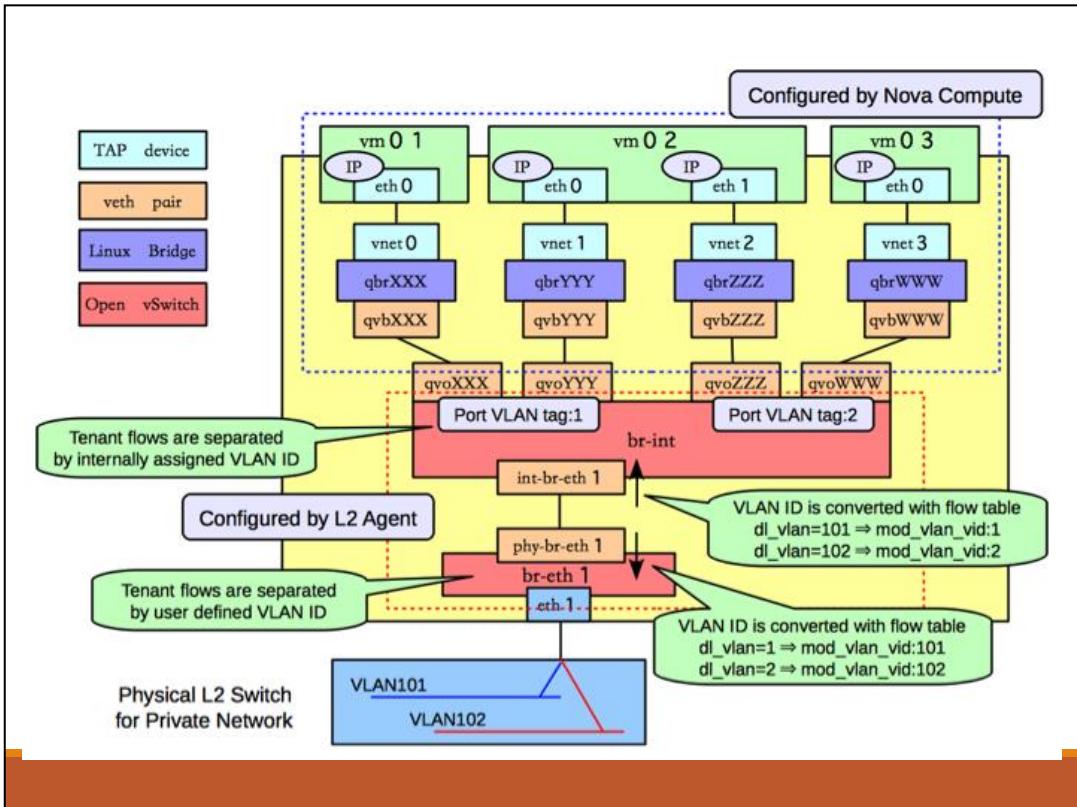
Other examples -

https://docs.openstack.org/keystone/rocky/api_curl_examples.html

Example: Authentication by Python

- Use of the v3 interface. When you're using the v3 interface, you should use sessions.
- When using sessions, a session object stores your credentials. The session object is passed to a client so the client can do appropriate OpenStack-related actions.
- After a session is established, the session object is responsible for handling authentication. When a request that needs authentication is sent to the session, then the session is responsible for requesting a token from the authentication plugin (or using an existing token). An OpenStack authentication plugin is an implementation of a method of authentication. Identity plugins are those associated with Keystone.

```
from keystoneauth1.identity import v3
from keystoneauth1 import session
from keystoneclient.v3 import client
auth = v3.Password(
    user_domain_name='default',
    username='admin',
    password='mypassword',
    project_domain_name='default',
    project_name='admin',
    auth_url='http://localhost:5000/v3'
)
mysess=session.Session(auth=auth)
keystone=client.Client(session=mysess)
return keystone
```



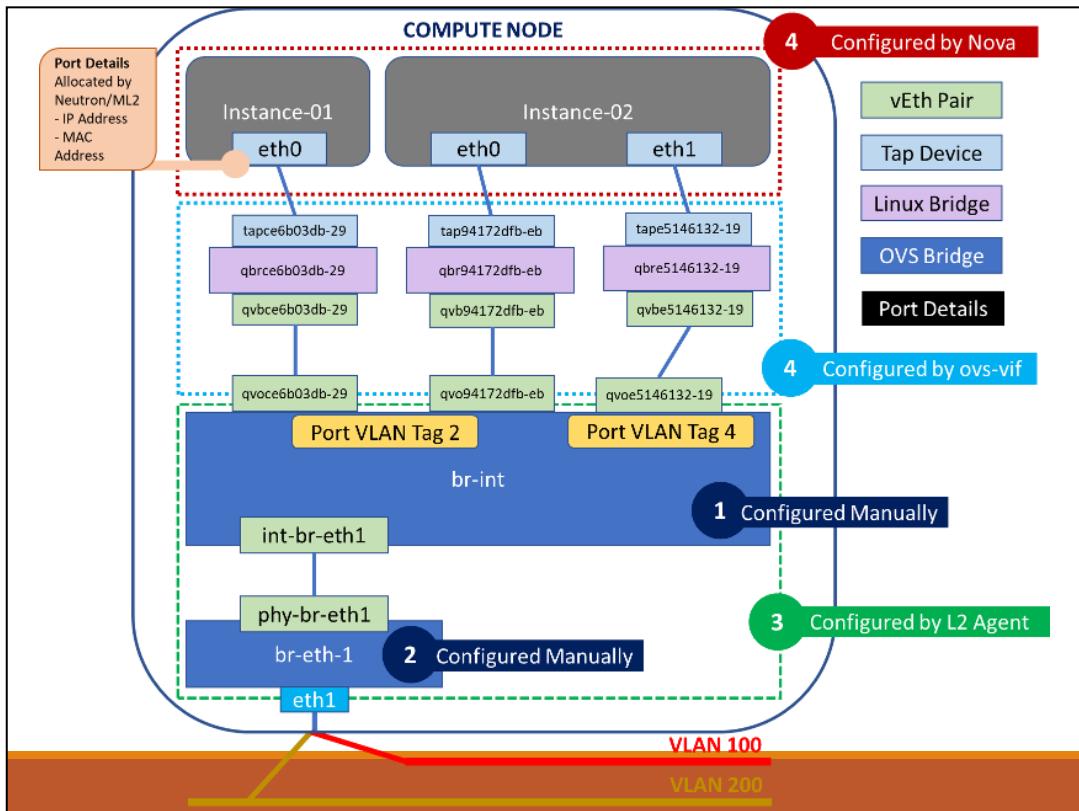
<https://docs.openstack.org/neutron/latest/contributor/internals/>

https://docs.openstack.org/neutron/latest/contributor/internals/openvswitch_agent.html

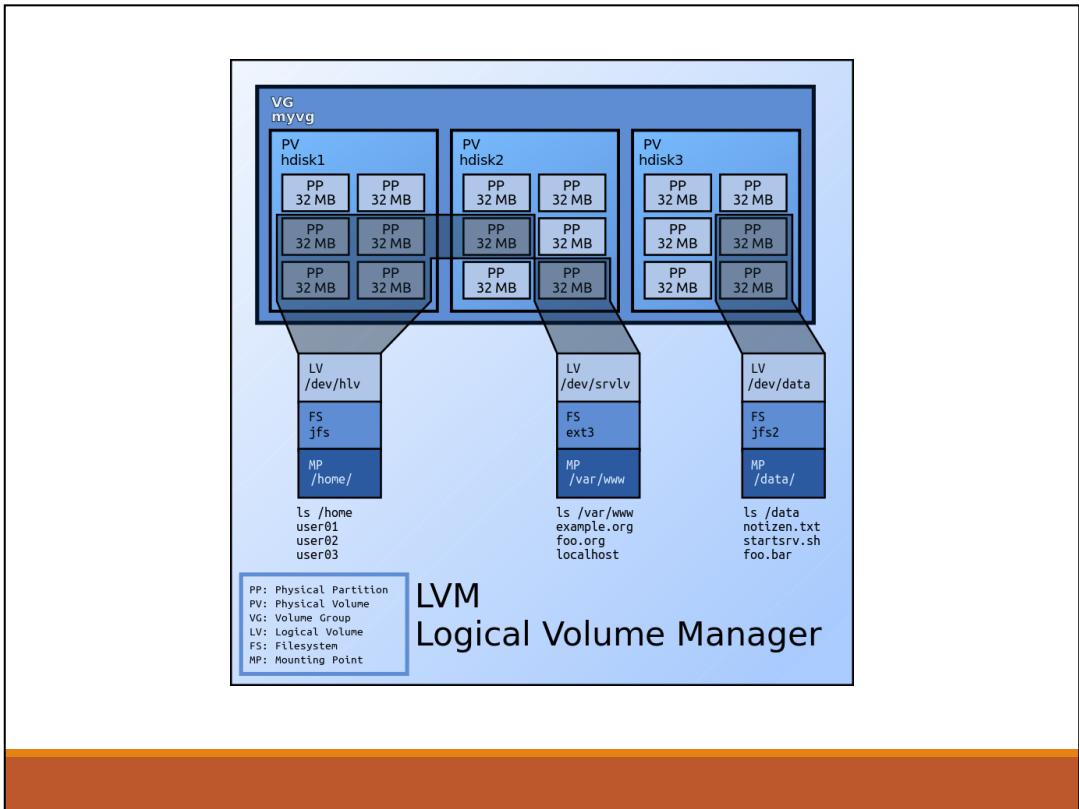
TUN/TAP provides packet reception and transmission for user space programs. It can be seen as a simple Point-to-Point or Ethernet device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program.

In order to use the driver a program has to open `/dev/net/tun` and issue a corresponding `ioctl()` to register a network device with the kernel. A network device will appear as `tunXX` or `tapXX`, depending on the options chosen. When the program closes the file descriptor, the network device and all corresponding routes will disappear.

Depending on the type of device chosen the userspace program has to read/write IP packets (with tun) or ethernet frames (with tap). Which one is being used depends on the flags given with the `ioctl()`.



<https://www.techblog.moebius.space/posts/2018-02-17-openstack-neutron-understanding-l2-networking-and-port-binding/>





Part 3 – OpenStack Cloud Management and Orchestration

Gianluca Reali

Università degli Studi di Perugia

Ref. www.opestack.org

1



Telemetry service: Ceilometer

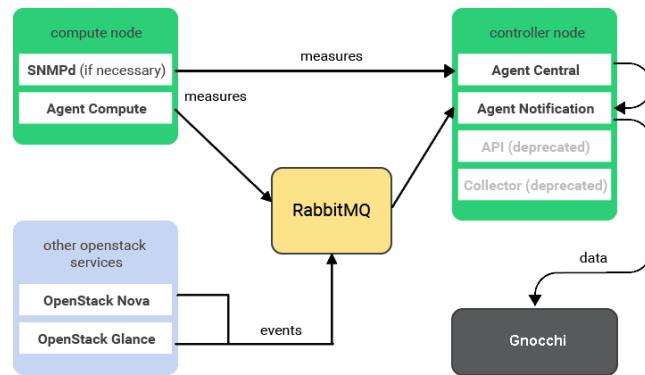
The *Ceilometer* project is a data collection service that provides the ability to normalise and transform data across all current OpenStack core components.

Ceilometer is a component of the *Telemetry* project. Its data can be used to provide customer billing, resource tracking, and alarming capabilities across all OpenStack core components.

<https://docs.openstack.org/ceilometer/latest/>



Ceilometer Architecture



These services communicate by using the OpenStack **messaging bus**. Ceilometer data is designed to be published to various endpoints for storage and analysis.

Gnocchi is an open-source time series database, useful for storing and indexing of time series data and resources at a large scale. This is useful in modern cloud platforms which are not only huge but also are dynamic and potentially multi-tenant.

The peculiarity of Gnocchi is that, instead of storing the raw point data as is usually done with time series, it aggregates them before storing them. By doing so, retrieving the data is extremely fast, since no aggregation needs to be done at the time of the queries but the data is ready.

Ceilometer Architecture

The Telemetry service consists of the following components:

A compute agent (ceilometer-agent-compute): Runs on each compute node and polls for resource utilization statistics.

A central agent (ceilometer-agent-central): Runs on a central management server to poll for resource utilization statistics for resources not tied to instances or compute nodes.

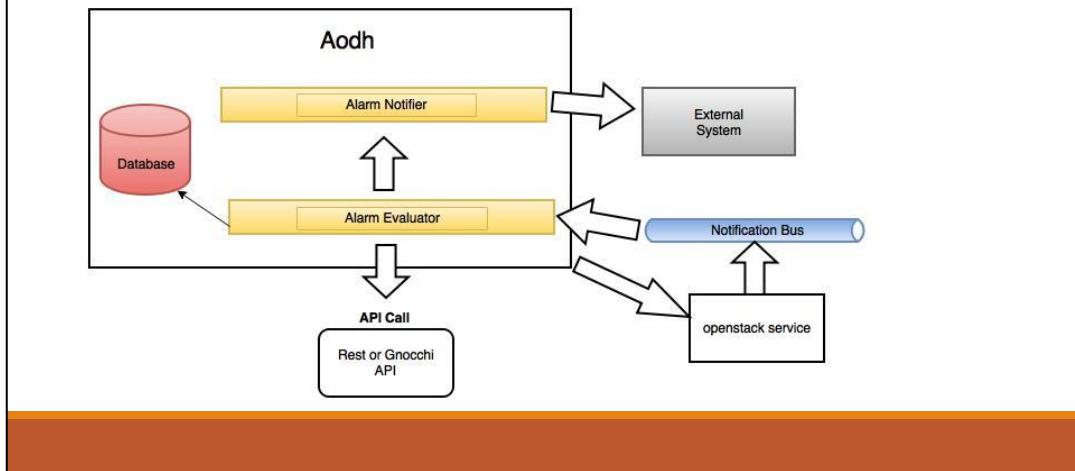
A notification agent (ceilometer-agent-notification): Runs on a central management server(s) and consumes messages from the message queue(s) to build event and metering data. Data is then published to defined targets. By default, data is pushed to [Gnocchi](#).

<https://docs.openstack.org/ceilometer/latest/>



Alarming Service (Aodh)

The Alarming service (aodh) project provides a service that enables the ability to trigger actions based on defined rules against metric or event data collected by Ceilometer or Gnocchi.





Alarming Service (Aodh)

The Telemetry Alarming service consists of the following components:

API server (aodh-api)

Runs on one or more central management servers to provide access to the alarm information stored in the data store.

Alarm evaluator (aodh-evaluator)

Runs on one or more central management servers to determine alarm state transition, that is when alarms fire due to the associated statistic trend crossing a threshold over a sliding time window.

Notification listener (aodh-listener)

Runs on a central management server and determines when to fire alarms. The alarms are generated based on defined rules against events, which are captured by the Telemetry Data Collection service's notification agents.

Alarm notifier (aodh-notifier)

Runs on one or more central management servers to allow alarms to be notified.



Alarms

Alarms provide user-oriented **Monitoring-as-a-Service** for resources running on OpenStack. This type of monitoring ensures you can automatically scale in or out a group of instances through the Orchestration service, but you can also use alarms for general-purpose awareness of your cloud resources' health.

These alarms follow a tri-state model:

- **ok:** The rule governing the alarm has been evaluated as False.
- **alarm:** The rule governing the alarm has been evaluated as True.
- **insufficient data:** There are not enough datapoints available in the evaluation periods to meaningfully determine the alarm state.

<https://docs.openstack.org/aodh/latest/admin/telemetry-alarms.html>



Alarm Definition

Threshold rule alarms

For conventional threshold-oriented alarms, state transitions are governed by:

- A **static threshold value** with a comparison operator such as greater than or less than.
- A **statistic selection** to aggregate the data.
- A **sliding time window** to indicate how far back into the recent past you want to look.



Alarm Creation

An example of creating a Gnocchi threshold-oriented alarm, based on an upper bound on the CPU utilization for a particular instance:

```
$ aodh alarm create \  
  --name cpu_hi \  
  --type gnocchi_resources_threshold \  
  --description 'instance running hot' \  
  --metric cpu_util \  
  --threshold 70.0 \  
  --comparison-operator gt \  
  --aggregation-method mean \  
  --granularity 600 \  
  --evaluation-periods 3 \  
  --alarm-action 'log://' \  
  --resource-id INSTANCE_ID \  
  --resource-type instance
```

This creates an alarm that will fire when the average CPU utilization for an individual instance exceeds 70% for three consecutive 10 minute periods. The notification in this case is simply a log message, though it could alternatively be a webhook URL



Orchestration service overview

- The **Orchestration service** provides a template-based orchestration for describing a cloud application by running OpenStack API calls to generate running cloud applications.
- The software integrates other core components of OpenStack into a one-file template system. The templates allow you to **create most OpenStack resource types such as instances, floating IPs, volumes, security groups, and users**.
- It also provides advanced functionality such as instance high availability, instance auto-scaling, and nested stacks.



Orchestration service overview

Stack - A stack stands for all the resources necessary to deploy an application. It can be as simple as a single instance and its resources, or as complex as multiple instances with all the resource dependencies that comprise a multi-tier application.



Heat Orchestration Template (HOT)

The most basic template may contain only a single resource definition using only predefined properties (along with the mandatory Heat template version tag).

```
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: cirros
      flavor: m1.nano
```

https://docs.openstack.org/heat/latest/template_guide/index.html
https://docs.openstack.org/heat/latest/template_guide/openstack.html

12

Each HOT template has to include the `heat_template_version` key with a valid version of HOT, e.g. 2015-10-15 (see [Heat template version](#) for a list of all versions). While the `description` is optional, it is good practice to include some useful text that describes what users can do with the template.

In case you want to provide a longer description that does not fit on a single line, you can provide multi-line text in YAML, for example:

```
description: >
```

This is how you can provide a longer description
of your template that goes over several lines.

The `resources` section is required and must contain at least one resource definition. In the example above, a compute instance is defined with fixed values for the ‘key_name’, ‘image’ and ‘flavor’ parameters.

Note that all those elements, i.e. a key-pair with the given name, the image and the flavor have to exist in the OpenStack environment where the template is used. Typically a template is made more easily reusable, though, by defining a set of *input parameters* instead of hard-coding such values.

See OpenStack Resource Types file Heat pag. 239



Template input parameters

Input parameters defined in the `parameters` section of a HOT template allow users to customize a template during deployment.

```
heat_template_version: 2015-04-30
description: Simple template to deploy a single compute instance
parameters:
  key_name:
    type: string
    label: Key Name
    description: Name of key-pair to be used for compute instance
  image_id:
    type: string
    label: Image ID
    description: Image to be used for compute instance
  instance_type:
    type: string
    label: Instance Type
    description: Type of instance (flavor) to be used

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: { get_param: key_name }
      image: { get_param: image_id }
      flavor: { get_param: instance_type }
```

this allows for providing custom key-pair names or image IDs to be used for a deployment. From a template author's perspective, this helps to make a template more easily reusable by avoiding hardcoded assumptions.

Three input parameters have been defined that have to be provided by the user upon deployment. The fixed values for the respective resource properties have been replaced by references to the corresponding input parameters by means of the `get_param` function (see also [Intrinsic functions](#)).

The `get_param` function references an input parameter of a template. It resolves to the value provided for this input parameter at runtime.

OpenStack Resource Types:

https://docs.openstack.org/heat/latest/template_guide/openstack.html



Template input parameters

You can also **define default values** for input parameters which will be used in case the user does not provide the respective parameter during deployment.

```
parameters:  
  instance_type:  
    type: string  
    label: Instance Type  
    description: Type of instance (flavor) to be used  
    default: m1.small
```

Another option that can be specified for a parameter is **to hide** its value when users request information about a stack deployed from a template.

```
parameters:  
  database_password:  
    type: string  
    label: Database Password  
    description: Password to be used for database  
    hidden: true
```



Providing template outputs

In addition to template customization through input parameters, you will typically want to **provide outputs to users**, which can be done in the *outputs* section of a template

```
outputs:  
  instance_ip:  
    description: The IP address of the deployed instance  
    value: { get_attr: [my_instance, first_address] }
```

You can **restrict the values** of an input parameter to make sure that the user defines valid data for this parameter.

```
parameters:  
  flavor:  
    type: string  
    label: Instance Type  
    description: Type of instance (flavor) to be used  
  constraints:  
    - allowed_values: [ m1.medium, m1.large, m1.xlarge ]  
      description: Value must be one of m1.medium, m1.large or m1.xlarge.
```

For example, the IP address by which the instance defined in the example above can be accessed should be provided to users. Otherwise, users would have to look it up themselves. The definition for providing the IP address of the compute.

Output values are typically resolved using intrinsic function such as the *get_attr* function in the example above (see also [Intrinsic functions](#)). instance as an output is shown.

The *get_attr* function references an attribute of a resource. The attribute value is resolved at runtime using the resource instance created from the respective resource definition.



Template structure

```
heat_template_version: 2016-10-14

description:
  # a description of the template

parameter_groups:
  # a declaration of input parameter groups and order

parameters:
  # declaration of input parameters

resources:
  # declaration of template resources

outputs:
  # declaration of output parameters

conditions:
  # declaration of conditions
```

The value of `heat_template_version` tells Heat not only the format of the template but also features that will be validated and supported.

The `parameter_groups` section allows for specifying how the input parameters should be grouped and the order to provide the parameters in. These groups are typically used to describe expected behavior for downstream user interfaces.

These groups are specified in a list with each group containing a list of associated parameters. The lists are used to denote the expected order of the parameters. Each parameter should be associated with a specific group only once using the parameter name to bind it to a defined parameter in the `parameters` section.

The `parameters` section allows for specifying input parameters that have to be provided when instantiating the template. Such parameters are typically used to customize each deployment (e.g. by setting custom user names or passwords) or for binding to environment-specifics like certain images.

Each parameter is specified in a separated nested block with the name of the parameters defined in the first line and additional attributes such as type or default value defined as nested elements.

The `resources` section defines actual resources that make up a stack deployed from the HOT template (for instance compute instances, networks, storage volumes).

Each resource is defined as a separate block in the `resources` section.

The `outputs` section defines output parameters that should be available to the user after a stack has been created. This would be, for example, parameters such as IP addresses of deployed instances, or URLs of web applications deployed as part of a stack.

Each output parameter is defined as a separate block within the `outputs` section

The `conditions` section defines one or more conditions which are evaluated based on input parameter values provided when a user creates or updates a stack. The condition can be associated with resources, resource properties and outputs. For example, based on the result of a condition, user can conditionally create resources, user can conditionally set different values of properties, and user can conditionally give outputs of a stack.

Example: Autoscaling Group

env.yaml file

```
parameters:  
  server_image: cirros  
  server_flavor: m1.nano  
  dns_nameserver: 8.8.8.8  
  ssh_key_name: mykey  
  external_network_id: e6e3539a-bf61-4497-af3e-14d59bcc8b72  
  cooldown_value: 180  
  granularity_value: 300  
  threshold_high: 60  
  threshold_low: 15
```



Example: Autoscaling Group

```
heat_template_version: 2013-05-23
description: >
    This template defines a single server

parameters:

  server_image:
    type: string
    description: glance image used to boot the server

  server_flavor:
    type: string
    description: flavor to use when booting the server

  dns_nameserver:
    type: string
    description: address of a dns nameserver reachable in your environment

  ssh_key_name:
    type: string
    description: name of ssh key to be provisioned on our server

  fixed_network_id:
    type: string
  fixed_subnet_id:
    type: string

  security_groups:
    type: comma_delimited_list
  metadata:
    type: json
```

basic-server-template.yaml file

```
resources:
  simple_server:
    type: "OS::Nova::Server"
    properties:
      image:
        get_param: server_image
      flavor:
        get_param: server_flavor
      key_name:
        get_param: ssh_key_name
      metadata:
        get_param: metadata
      networks:
        - port:
            get_resource: simple_server_ether
```



```
simple_server_ether0:
  type: "OS::Neutron::Port"
  properties:
    network_id:
      get_param: fixed_network_id
    fixed_ips:
      - subnet_id:
          get_param: fixed_subnet_id
    security_groups:
      get_param: security_groups
```

OpenStack resource types:

https://docs.openstack.org/heat/latest/template_guide/openstack.html

<https://docs.openstack.org/python-heatclient/latest/cli/stack.html>

```
stack create [-f {json,shell,table,value,yaml}] [-c COLUMN] [--noindent] [--prefix PREFIX] [--max-width <integer>] [--fit-width] [--print-empty] [-e <environment>] [-s <files-container>] [--timeout <timeout>] [--pre-create <resource>] [--enable-rollback] [--parameter <key=value>] [--parameter-file <key=file>] [--wait] [--poll SECONDS] [--tags <tag1,tag2...>] [--dry-run] -t <template> <stack-name>
```

Example: Autoscaling Group

basic-server-template.yaml file

```
heat_template_version: 2013-05-23
description: >
    This template defines a single server

parameters:

server_image:
    type: string
    description: glance image used to boot the server

server_flavor:
    type: string
    description: flavor to use when booting the server

dns_nameserver:
    type: string
    description: address of a dns nameserver reachable in your
environment
```

Example: Autoscaling Group

basic-server-template.yaml file

```
ssh_key_name:  
  type: string  
  description: name of ssh key to be provisioned on our server  
  
fixed_network_id:  
  type: string  
fixed_subnet_id:  
  type: string  
  
security_groups:  
  type: comma_delimited_list  
metadata:  
  type: json
```

Example: Autoscaling Group

```
resources:  
  simple_server:  
    type: "OS::Nova::Server"  
    properties:  
      image:  
        get_param: server_image  
      flavor:  
        get_param: server_flavor  
      key_name:  
        get_param: ssh_key_name  
      metadata:  
        get_param: metadata  
    networks:  
      - port:  
          get_resource:simple_server_eth0
```

basic-server-template.yaml file

```
simple_server_eth0:  
  type: "OS::Neutron::Port"  
  properties:  
    network_id:  
      get_param: fixed_network_id  
    fixed_ips:  
      - subnet_id:  
          get_param: fixed_subnet_id  
    security_groups:  
      get_param: security_groups
```



Example: Autoscaling Group

```
heat_template_version: 2013-05-23
description: >
    This template deploys an autoscaling
    group of nova servers.

parameters:

  server_image:
    type: string
    description: glance image used to boot
    the server

  server_flavor:
    type: string
    description: flavor to use when
    booting the server

  dns_nameserver:
    type: string
    description: address of a dns
    nameserver reachable in your
    environment

  ssh_key_name:
    type: string
    description: name of ssh key to be
    provisioned on our server

  external_network_id:
    type: string
    description: uuid of a network to use

  cooldown_value:
    type: number
    description: cooldown to trigger the
    scaleup and scaledown policy
```

autoscaling-template.yaml file



Example: Autoscaling Group

`autoscaling-template.yaml` file

```
granularity_value:
  type: number
  description: metric's granularity to
check

threshold_high:
  type: number
  description: threshold value for the
cpu_alarm_high

threshold_low:
  type: number
  description: threshold value for the
cpu_alarm_low

resources:
#####
# network resources. allocate a network
and router for our server. It would also be
possible to take advantage of existing
network resources
fixed_network:
  type: "OS::Neutron::Net"

# This is the subnet on which we will
deploy our server.
fixed_subnet:
  type: "OS::Neutron::Subnet"
  properties:
    cidr: 10.0.20.0/24
    network_id:
      get_resource: fixed_network
    dns_nameservers:
      - get_param: dns_nameserver
```



Example: Autoscaling Group

```
# create a router attached to the
external network provided as a
parameter to this stack.

extrouter:
  type: "OS::Neutron::Router"
  properties:
    external_gateway_info:
      network:
        get_param: external_network_id

    # attached fixed_subnet to our
    extrouter router.

extrouter_inside:
  type: "OS::Neutron::RouterInterface"
  properties:
    router_id:
      get_resource: extrouter
    subnet_id:
      get_resource:
        fixed_subnet
```

autoscaling-template.yaml file

```
#####
#
# security group.
#
# this permits ssh and icmp traffic
secgroup_common:
  type: "OS::Neutron::SecurityGroup"
  properties:
    rules:
      - protocol: icmp
      - port_range_min: 22
        port_range_max: 22
      protocol: tcp
```

SSH port 22

The port is used for Secure Shell (SSH) communication and allows remote administration access to the VM.



Example: Autoscaling Group

autoscaling-template.yaml file

```
#####
# autoscaling group: ....
instances_group:
type: "OS::Heat::AutoScalingGroup"
properties:
  min_size: 1
  max_size: 3
resource:
  # Here is our nested stack.
  type: basic-server-template.yaml
  properties:
    server_image:
      get_param: server_image
    server_flavor:
      get_param: server_flavor
    ssh_key_name:
      get_param: ssh_key_name
    dns_nameserver:
      get_param: dns_nameserver
fixed_network_id:
  get_resource: fixed_network
fixed_subnet_id:
  get_resource: fixed_subnet
security_groups:
  - get_resource: secgroup_common
# This metadata is used for restricting
# an Aodh query to servers launched by this
# stack.
metadata: {"metering.server_group":
{get_param: "OS::stack_id"}}
```



Example: Autoscaling Group

```
#####
# autoscale logic. The following
resources define a pair of Aodh alarms
based on Gnocchi and associates them
with a specific policy...
instance_scaleup_policy:
  type: "OS::Heat::ScalingPolicy"
  properties:
    adjustment_type:
      change_in_capacity
        auto_scaling_group_id:
          get_resource: instances_group
      cooldown:
        get_param: cooldown_value
        scaling_adjustment: 1
```

autoscaling-template.yaml file

```
instance_scaledown_policy:
  type: "OS::Heat::ScalingPolicy"
  properties:
    adjustment_type:
      change_in_capacity
        auto_scaling_group_id:
          get_resource: instances_group
      cooldown:
        get_param: cooldown_value
        scaling_adjustment: -1
```

Details about the resource "OS::Heat::ScalingPolicy" can be found at

https://docs.openstack.org/heat/pike/template_guide/openstack.html#OS::Heat::ScalingPolicy

In particular it has the attributes:

alarm_url: A signed url to handle the alarm.

Show: Detailed information about resource.

signal_url: A url to handle the alarm using native API.



Example: Autoscaling Group

```
cpu_alarm_high:  
  type:  
  OS::Aodh::GnocchiAggregationByResourcesAlarm  
  properties:  
    description: Scale up if CPU > 60%  
    metric: cpu_util  
    aggregation_method: mean  
    granularity:  
      get_param: granularity_value  
    evaluation_periods: 1  
    threshold:  
      get_param: threshold_high  
    resource_type: instance  
    comparison_operator: gt
```

```
autoscaling-template.yaml file  
  
alarm_actions:  
  - str_replace:  
    template: trust+url  
    params:  
      url: {get_attr:  
        [instance_scaleup_policy, signal_url]}  
    query:  
      str_replace:  
        template: '{"": {"server_group":  
          "stack_id": ""}}'  
        params:  
          stack_id: {get_param:  
            "OS::stack_id"}  
  cpu_alarm_low:  
    type:  
    OS::Aodh::GnocchiAggregationByResourcesAlarm  
    properties:  
      description: Scale down if CPU < 15%  
      for 5 minutes
```

Specifications of the resource OS::Aodh::GnocchiAggregationByResourcesAlarm can be found in
https://docs.openstack.org/heat/pike/template_guide/openstack.html#OS::Heat::ScalingPolicy-attr-signal_url

OpenStack Identity manages authentication and authorization. A trust is an OpenStack Identity extension that enables delegation and, optionally, impersonation through keystone. A trust extension defines a relationship between:

Trustor The user delegating a limited set of their own rights to another user. **Trustee** The user trust is being delegated to, for a limited time.

The trust can eventually allow the trustee to impersonate the trustor. For security reasons, some safeties are added. For example, if a trustor loses a given role, any trusts the user issued with that role, and the related tokens, are automatically revoked.

When adding an alarm action with the scheme `trust+http`, Aodh allows the user to provide a trust ID to acquire a token with which to make a webhook request (Webhooks are **automated messages sent from apps when something happens**. They have a message—or payload—and are sent to a unique URL). (If no trust ID is provided then Aodh creates a trust internally, in which case the issue is not present.) However, Aodh makes no attempt to verify that the user creating the alarm is the trustor or has the same rights as the trustor - it also does not attempt to check that the trust is for the same project as the alarm.

The nature of the `trust+http` alarm notifier is that it allows the user to obtain a token given the ID of a trust for which Aodh is the trustee, since the URL is arbitrary and not limited to services in the Keystone catalog.

The str_replace function replaces strings.

signal_url: A url to handle the alarm using native API.

The get_attr function allows referencing an attribute of a resource. At runtime, it will be resolved to the value of an attribute of a resource instance created from the respective resource definition of the template.

Aodh calls whatever URL you ask it to. The URL that the Heat autoscaling resources provide (via the signal_url attribute) is for the [resource signal API](#).

Send a signal to a resource

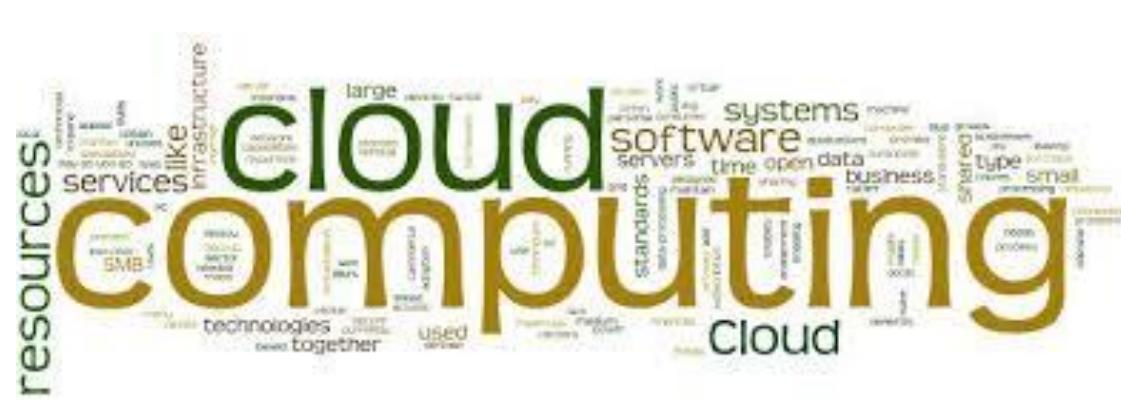
POST

/v1/{tenant_id}/stacks/{stack_name}/{stack_id}/resources/{resource_name}/signal

Example: Autoscaling Group

autoscaling-template.yaml file

```
metric: cpu_util
aggregation_method: mean
granularity:
  get_param: granularity_value
evaluation_periods: 1
threshold:
  get_param: threshold_low
resource_type: instance
comparison_operator: Lt
alarm_actions:
  - str_replace:
      template: trust+url
      params:
        url: {get_attr:
          [instance_scaledown_policy, signal_url]}
      query:
        str_replace:
          template: '{": {"server_group":
            "stack_id"}}'
          params:
            stack_id: {get_param:
              "OS::stack_id"}
```



Part 4 – Serverless Computing and OpenFaas

GIANLUCA REALI

Evolution of software development

Deploy hardware

Install the operating system

Deploy the application package

Run the code

Traditional software development

Deploy hardware

Install the operating system

Deploy the application package

Run the code

Software development in early cloud computing

Deploy hardware

Install the operating system

Deploy the application package

Run the code

Software development with virtualization

Deploy hardware

Install the operating system

Deploy the application package

Run the code

Software development with containerization



Software development with serverless

Deploy hardware

Install the operating system

Deploy the application package

Run the code

With serverless applications, all low-level concerns are outsourced and managed, and the focus is on the last step : **Run the code.**

Serverless is a confusing term. It means leaving the responsibility of servers to third-party organizations. In other words, it means not getting rid of servers but server operations. When you run serverless, someone else handles the procurement, shipping, and installation of your server operations. This decreases your costs since it lets you focus on the application logic.

Serverless Compute Manifesto (2006 AWS developer conference)



-
- Functions as the building blocks
 - No servers, VMs, or containers
 - No storage
 - Implicitly fault-tolerant functions
 - Scalability with the request
 - No cost for idle time
 - Bring Your Own Code (BYOC)

Sample Serverless Use Cases



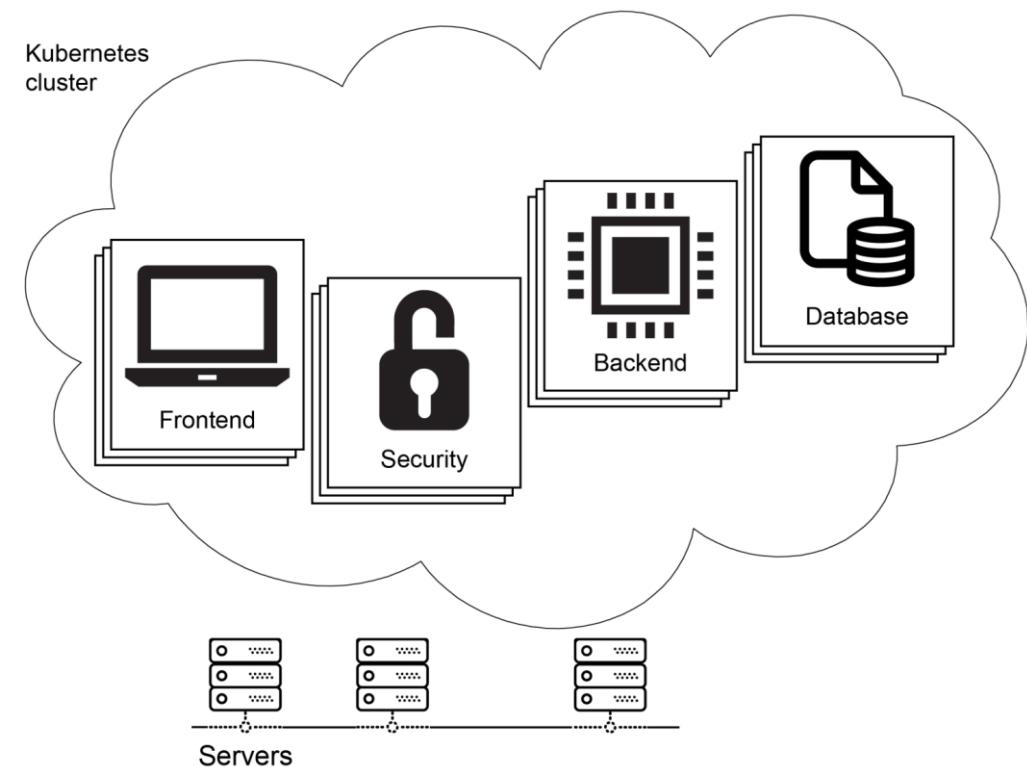
- Data processing
- Webhooks
- Check-out and payment
- Real-time chat applications

In synthesis, applications tolerant to latency, short lived, avoiding vendor lock-in and ecosystem dependencies



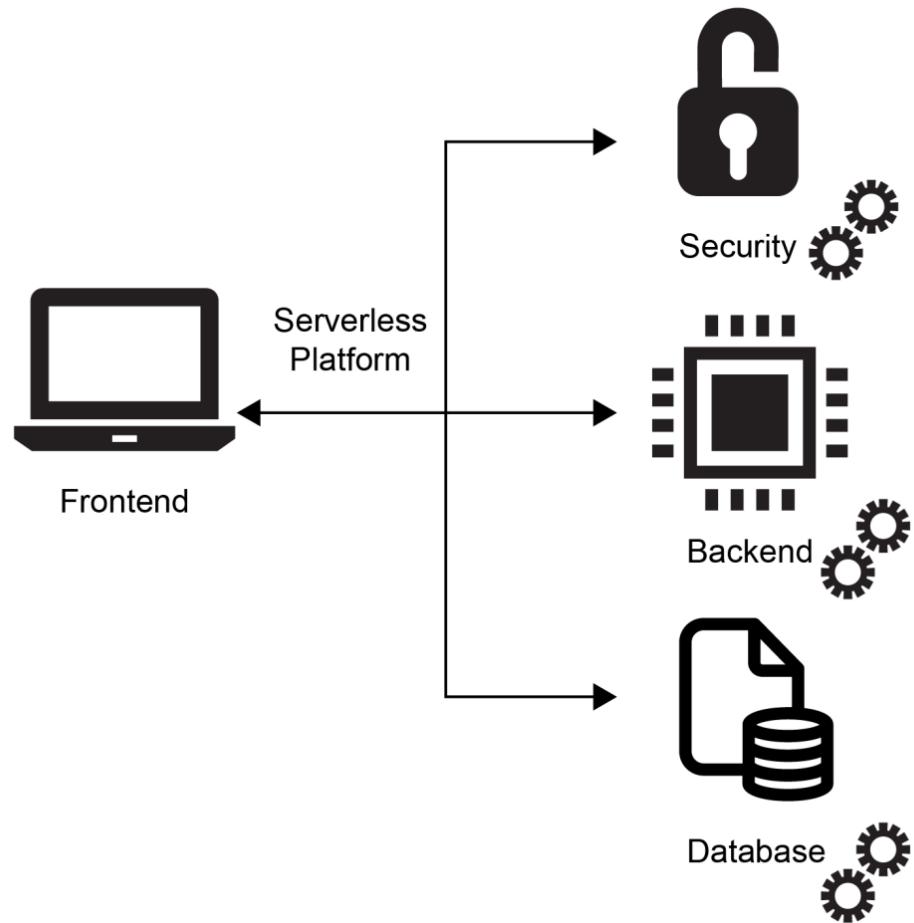
Example (legacy µservices approach)

- For the same **e-commerce system**, you have frontend, backend, database, and security components, but they would be isolated units. These components would be packaged as containers and would be managed by a container orchestrator such as Kubernetes. This enables the installing and scaling of components independently since they are distributed over multiple servers.
- Microservices are deployed to the servers, which are still managed by the operations teams.



Example (serverless)

- The best option for the backend logic is to convert it into functions and deploy them into a serverless platform such as AWS Lambda or Google Cloud Functions. Finally, the frontend could be served by storage services such as AWS Simple Storage Service (S3) or Google Cloud Storage.
- With a serverless design, it is only required to define these services for you to have scalable, robust, and managed applications running in harmony.





Function as a Service (FaaS)

-
- FaaS is the most popular and widely adopted implementation of serverless architecture.
 - All major cloud providers have FaaS products, such as AWS Lambda, Google Cloud Functions, and Azure Functions.
 - As its name implies, the unit of deployment and management in FaaS is the function. Functions in this context are no different from any other function in any other programming language. They are expected to take some arguments and return values to implement business needs.
 - FaaS platforms handle the management of servers and make it possible to run event-driven, scalable functions.

Properties of a FaaS



- **Stateless:** Functions are designed to be stateless and ephemeral operations where no file is saved to disk and no caches are managed. At every invocation of a function, it starts quickly with a new environment, and it is removed when it is done.
- **Event-triggered:** Functions are designed to be triggered directly and based on events such as cron time expressions, HTTP requests, message queues, and database operations. For instance, it is possible to call the startConversation function via an HTTP request when a new chat is started. Likewise, it is possible to launch the syncUsers function when a new user is added to a database.
- **Scalable:** Functions are designed to run as much as needed in parallel so that every incoming request is answered and every event is covered.
- **Managed:** Functions are governed by their platform so that the servers and underlying infrastructure is not a concern for FaaS users.



Kubernetes and Serverless

A strong connection between Kubernetes and serverless architectures exists:

- Serverless and Kubernetes arrived on the cloud computing scene at **about the same time**, in 2014.
- Kubernetes gained dramatic adoption in the industry and became the de facto container management system. It enables running **both stateless applications**, such as web frontends and data analysis tools, **and stateful applications**, such as databases, inside containers. running microservices and containerized applications is a crucial factor for successful, scalable, and reliable cloud-native applications.
- **No vendor lock-in:** If you use a Kubernetes-backed serverless platform, you will be able to quickly move between cloud providers or even on-premises systems.
- **Reuse of services:** Kubernetes offers an opportunity to deploy serverless functions side by side with existing services. It makes it easier to operate, install, connect, and manage both serverless and containerized applications.

Overview of OpenFaas

-
- OpenFaas is an **open source (kind of...)** framework written in Go and used to build and deploy serverless functions on top of containers
 - OpenFaaS was originally designed to work with Docker Swarm, which is the clustering and scheduling tool for Docker containers. Later, the OpenFaaS framework was rearchitected to **support also Kubernetes**.
 - OpenFaaS comes with a **built-in UI named OpenFaaS Portal**, which can be used to create and invoke the functions from the web browser. This portal also offers a **CLI named faas-cli** that allows us to manage functions through the **command line**.



Overview of OpenFaas

- **Main Features**

- Portable functions platform - run functions on any cloud or on-premises without fear of lock-in
- Write functions in any language and package them in Docker/OCI-format containers
- Easy to use - built-in UI, powerful CLI and one-click installation
- Scaling capabilities, able to scale up a function when there is increased demand, and it will scale down when demand decreases, or even scale down to zero when the function is idle
- Ecosystem - community marketplace for functions and language templates

OpenFaaS main components



Functions as a Service

API Gateway

Function Watchdog



Prometheus



Swarm



Kubernets

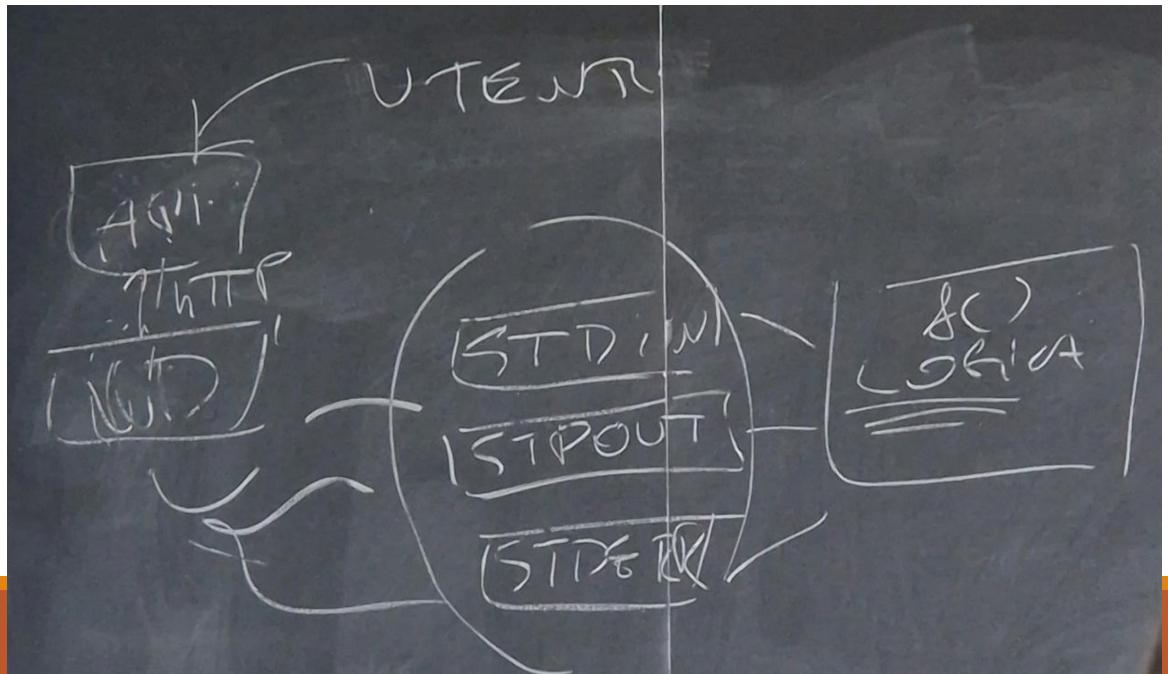


docker

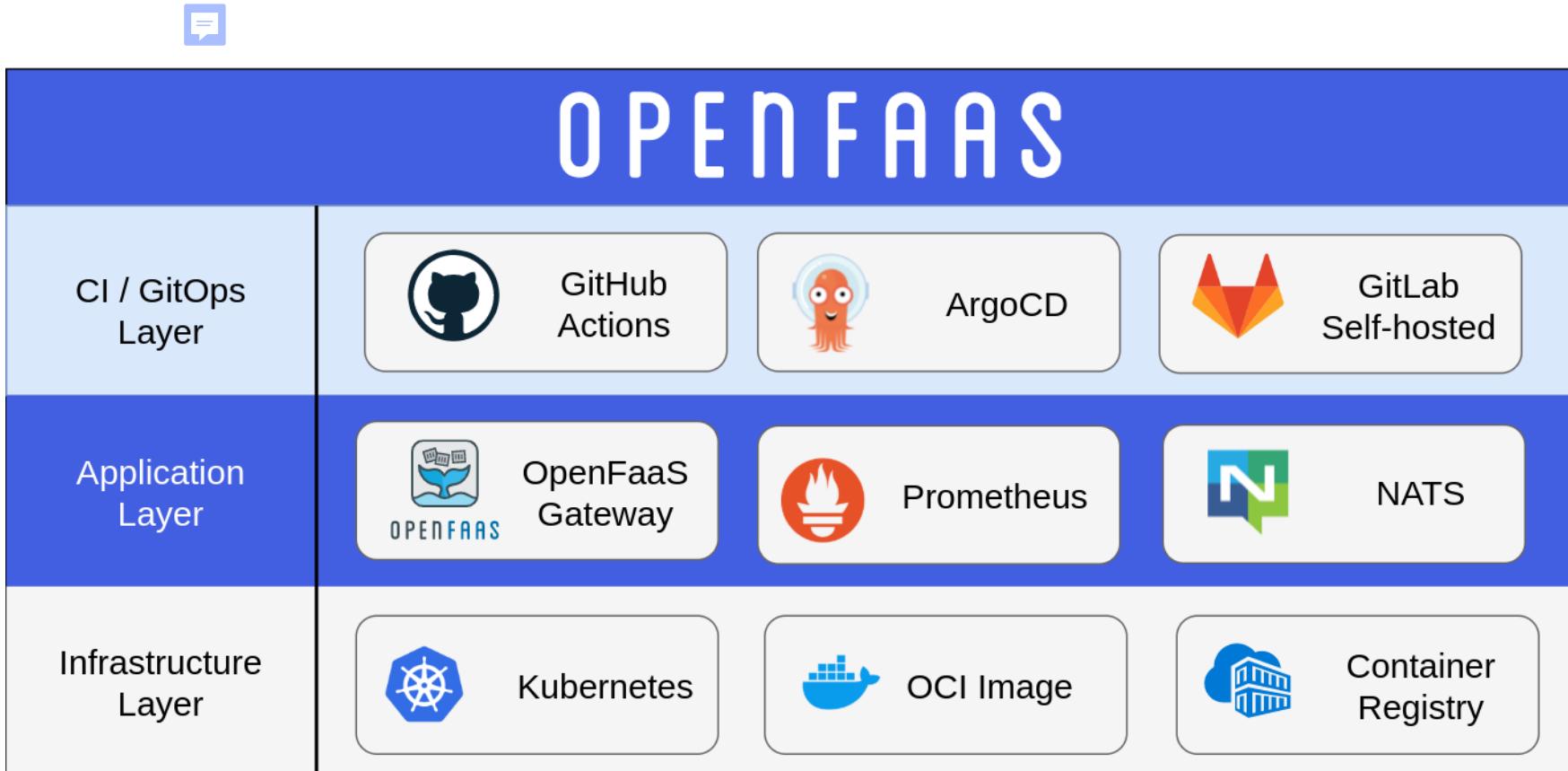
Versions

ci sono diverse versioni di OpenFaaS, noi usiamo la seconda

- OpenFaaS Standard/For Enterprises for commercial use & production, deployed to Kubernetes
- Community Edition - free for personal use only, 60-day limit for commercial use, deployed to Kubernetes
- faasd - Free to use on any cloud or on-premises, works on a single VM (no clustering or Kubernetes)



Conceptual layers of the OpenFaaS stack



OpenFaas functions



- OpenFaas functions can be written in **any language** supported by Linux or Windows, and they can then be converted to a serverless function using Docker containers. This is a major advantage of the OpenFaas framework compared to other serverless frameworks that support only predefined languages and runtimes
- This requires to follow some specific steps, as to **create the function code**, **add any dependencies**, and **create a Dockerfile to build the Docker image**. It requires a certain amount of **understanding** of the OpenFaas platform in order to be able to perform all the needed tasks.
- As a **solution**, OpenFaas has a **template store** that includes **prebuilt templates** for a set of supported languages. This means that you can **download** these templates from the template store, **update** the function code, and then the **CLI** does the rest to **build** the Docker image.

Create functions



Once you've installed the **faas-cli** you can start creating and deploying functions via the **faas-cli up** command or using the individual commands:

- *faas-cli build* - build an image into the local Docker library
- *faas-cli push* - push that image to a remote container registry
- *faas-cli deploy* - deploy your function into a cluster

The *faas-cli up* command automates all of the above in a single command.

Templates



The OpenFaaS CLI has a **template engine** built-in which can create new functions in a given programming language. The way this works is by reading a list of templates from the `./template` location in your current working folder.

Before creating a new function you can **pull** in the official OpenFaaS language templates from GitHub via the [templates repository](#).

```
$ faas-cli template pull
```

This way it possible to generate functions in the most popular languages and explains how you can manage their dependencies too.



Classic watchdog vs. of-watchdog templates

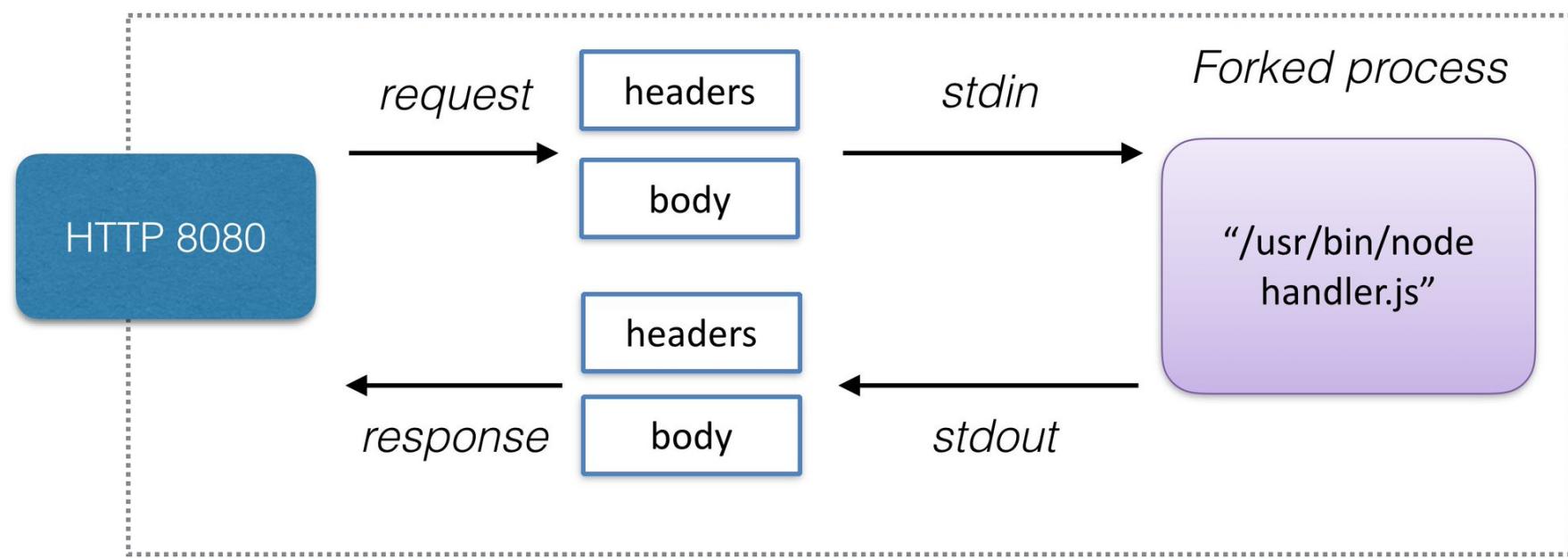
- The **Classic Templates** are held in the openfaas/templates repository and are based upon the **Classic Watchdog** which uses **STDIO** to communicate with your function.
- The **of-watchdog** uses **HTTP** to communicate with **functions** and most of its templates are available in the openfaas organisation in their own separate repositories on GitHub and in the store.



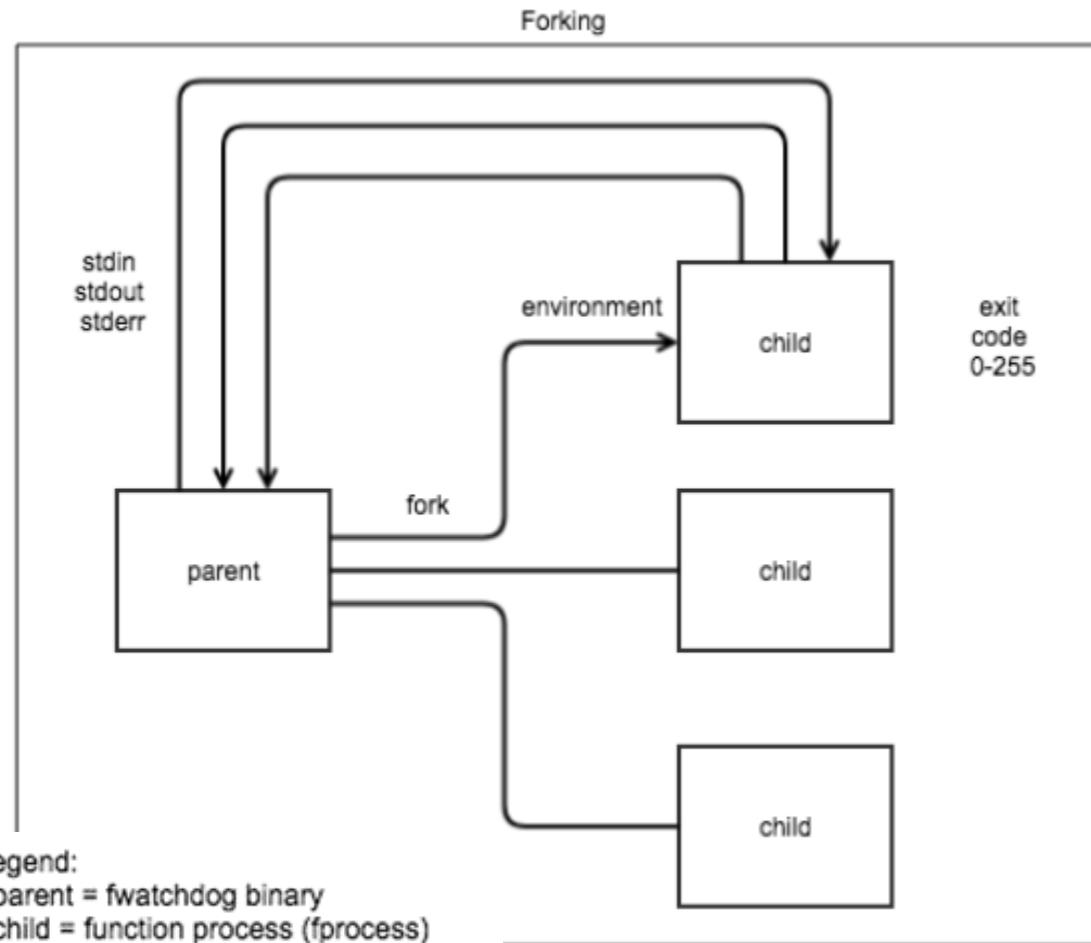
Classic Whatchdog

Every function needs to embed this binary and use it as its **ENTRYPOINT** or **CMD**, in effect it is the init process for your container. Once your process is forked the watchdog passes in the HTTP request via `stdin` and reads a HTTP response via `stdout`. This means your process does not need to know anything about the web or HTTP.

a tiny web-server or shim that forks your desired process for every incoming HTTP request



Classic Whatchdog mode



Behave as defined by the FaaS definition

1. Receive an event from the controller
2. Create a child process to execute the function code
3. Run the function code in the child
4. Pass the event as argument to the function
5. Waits for the end of execution of the function
6. Terminates the child



Whatchdog HTTP methods

- The HTTP methods supported for the watchdog are:
 - With a body: **POST, PUT, DELETE, UPDATE**
 - Without a body: **GET**
- The API Gateway currently supports the POST route for functions.
- **Content-Type of request/response**
 - By default the watchdog will match the response of your function to the "Content-Type" of the client.
 - If your client sends a JSON post with a Content-Type of application/json this will be matched automatically in the response.
 - If your client sends a JSON post with a Content-Type of text/plain this will be matched automatically in the response too



of-watchdog

- It introduced an **improved control over HTTP responses**, "hot functions", persistent connection pools or to cache a machine-learning model in memory.
- The of-watchdog implements an **HTTP server** listening on port 8080, and acts as a reverse proxy for running functions and microservices.
- It does not aim to replace the Classic Watchdog, but offers another option for those who need the additional functions introduced.

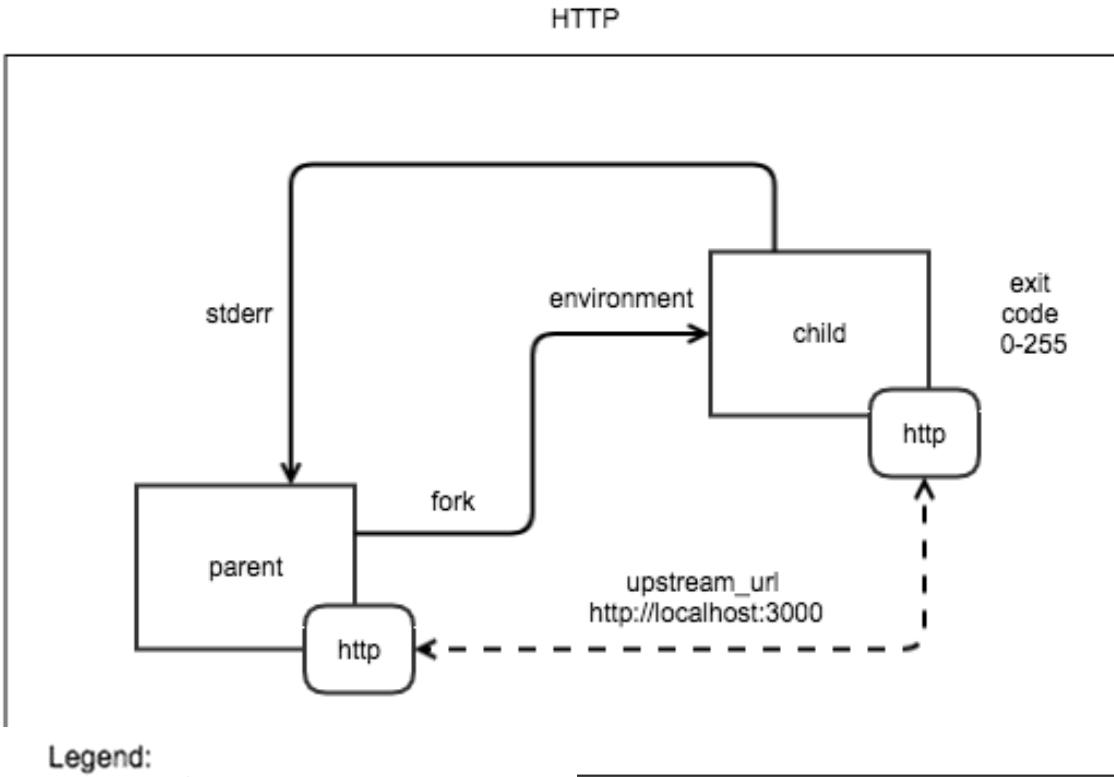


of-watchdog

Goals

- Keep function process **warm** for lower latency / caching / persistent connections through using HTTP
- Enable **streaming of large responses from functions**, beyond the RAM or disk capacity of the container
- Different modes available for the of-watchdog which **changes how it interacts with your microservice or function code.**

of-watchdog HTTP mode



HTTP mode - the default and most efficient option

HTTP mode is recommended for all templates where the target language has a HTTP server implementation available.

Legend:
- parent = watchdog binary
- child = function process (fprocess)

HTTP reverse proxy: A process is forked when the watchdog starts, then any request incoming to the watchdog is forwarded to a HTTP port within the container.

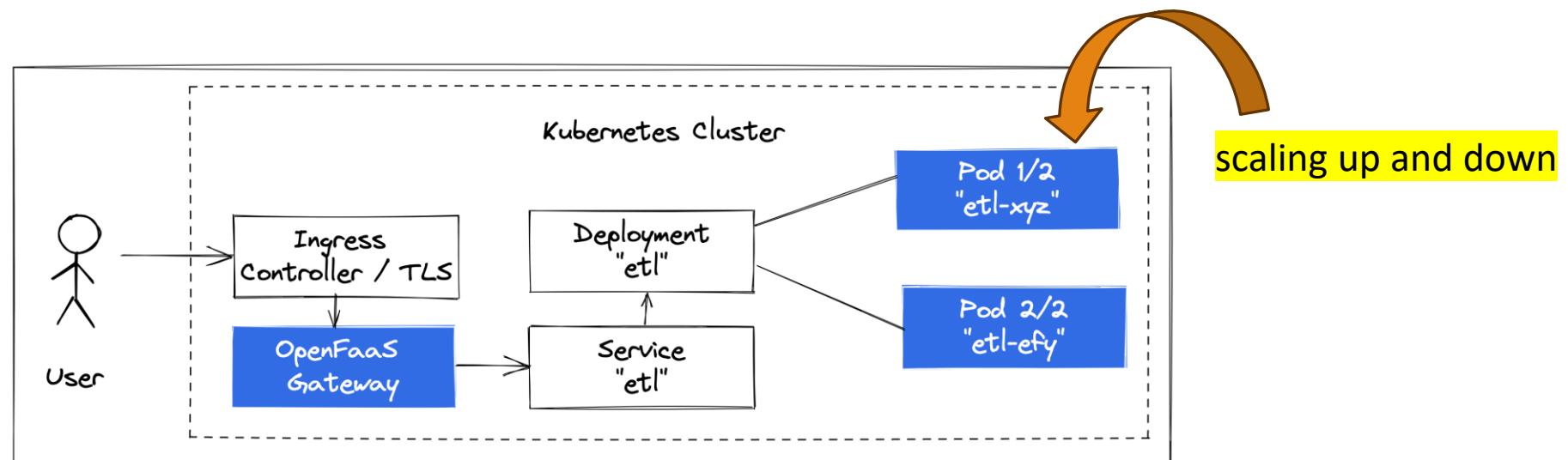


Invocation of functions

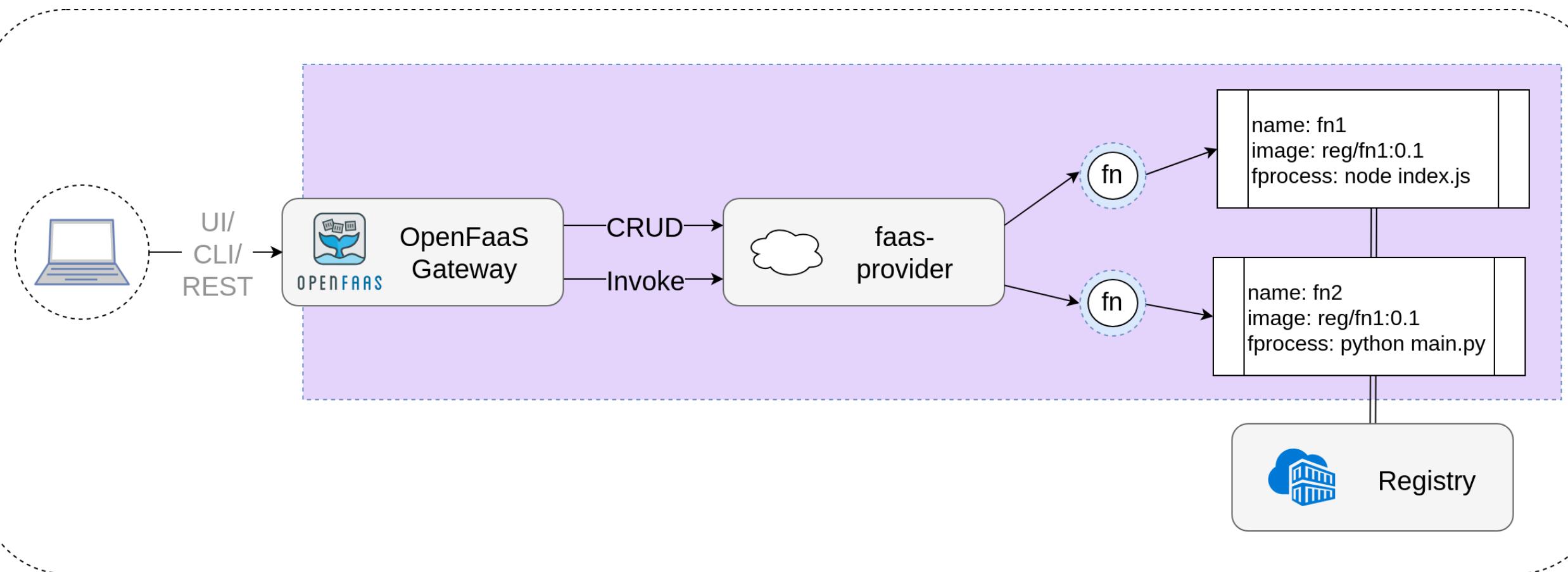
Synchronous function invocations:

In an OpenFaaS installation, functions can be invoked through **HTTP requests** to the **OpenFaaS gateway**, specifying the path as part of the URL.

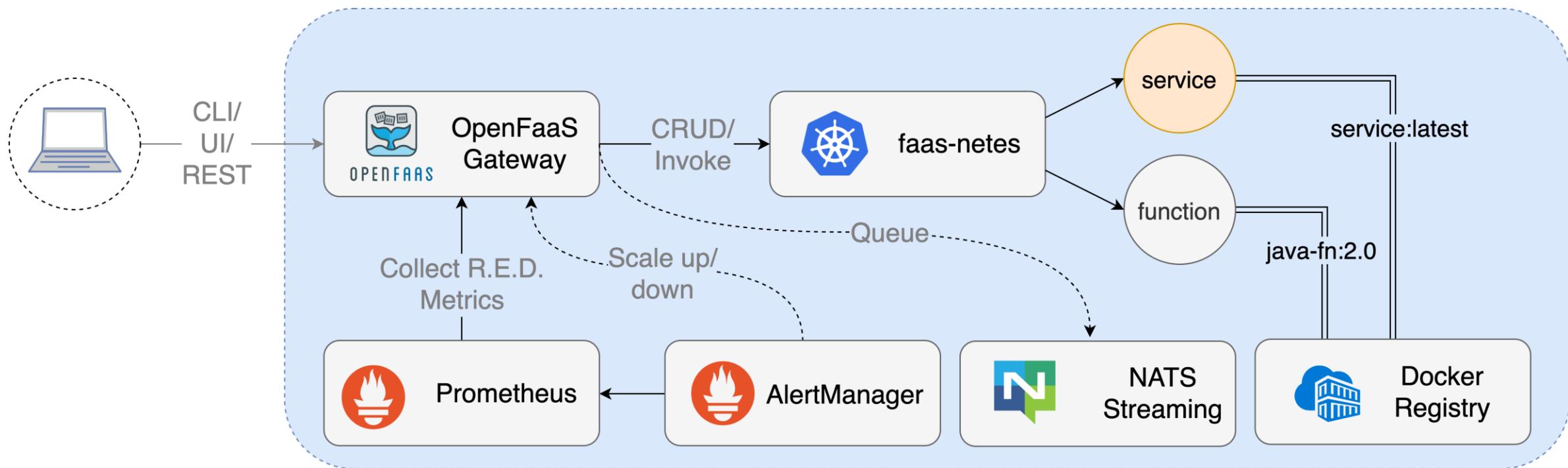
Each function is deployed as a Kubernetes Deployment and Service



Conceptual workflow



Conceptual workflow



faas-netes is an [OpenFaaS provider](#) which enables Kubernetes for [OpenFaaS](#). It's part of a larger stack that brings a cloud-agnostic serverless experience to Kubernetes.



Invocation of functions

Synchronous function invocations:

- During development you may invoke the OpenFaaS gateway using a HTTP request to `http://127.0.0.1:8080/function/NAME`, where NAME is the name of the function.
- When you move to production, you may have another layer between your users and the gateway such as a reverse proxy or Kubernetes Ingress Controller.
- The connection between the caller and the function remains connected until the invocation has completed, or times out.

Invocation of functions

nah

Asynchronous function invocations:

With an asynchronous invocation, the HTTP request is enqueued to NATS, followed by an "accepted" header and call-id being returned to the caller. Next, at some time in the future, a separate queue-worker component dequeues the message and invokes the function synchronously.

There is never any direct connection between the caller and the function, so the caller gets an immediate response, and can subscribe for a response via a webhook when the result of the invocation is available.



Events

- OpenFaaS functions can be triggered easily by any kind of event. The most common use-case is **HTTP** which acts as a lingua franca between internet-connected systems.
- A number of **event triggers** are supported in OpenFaaS. With each of them, a long-running daemon subscribes to a topic or queue, then when it receives messages looks up the relevant **functions** and invokes them synchronously or asynchronously.
- Popular event sources include: Cron, Apache Kafka and AWS SQS.

Built-in sample triggers¶

HTTP / webhooks¶

This is the default, and standard method for interacting with your Functions.

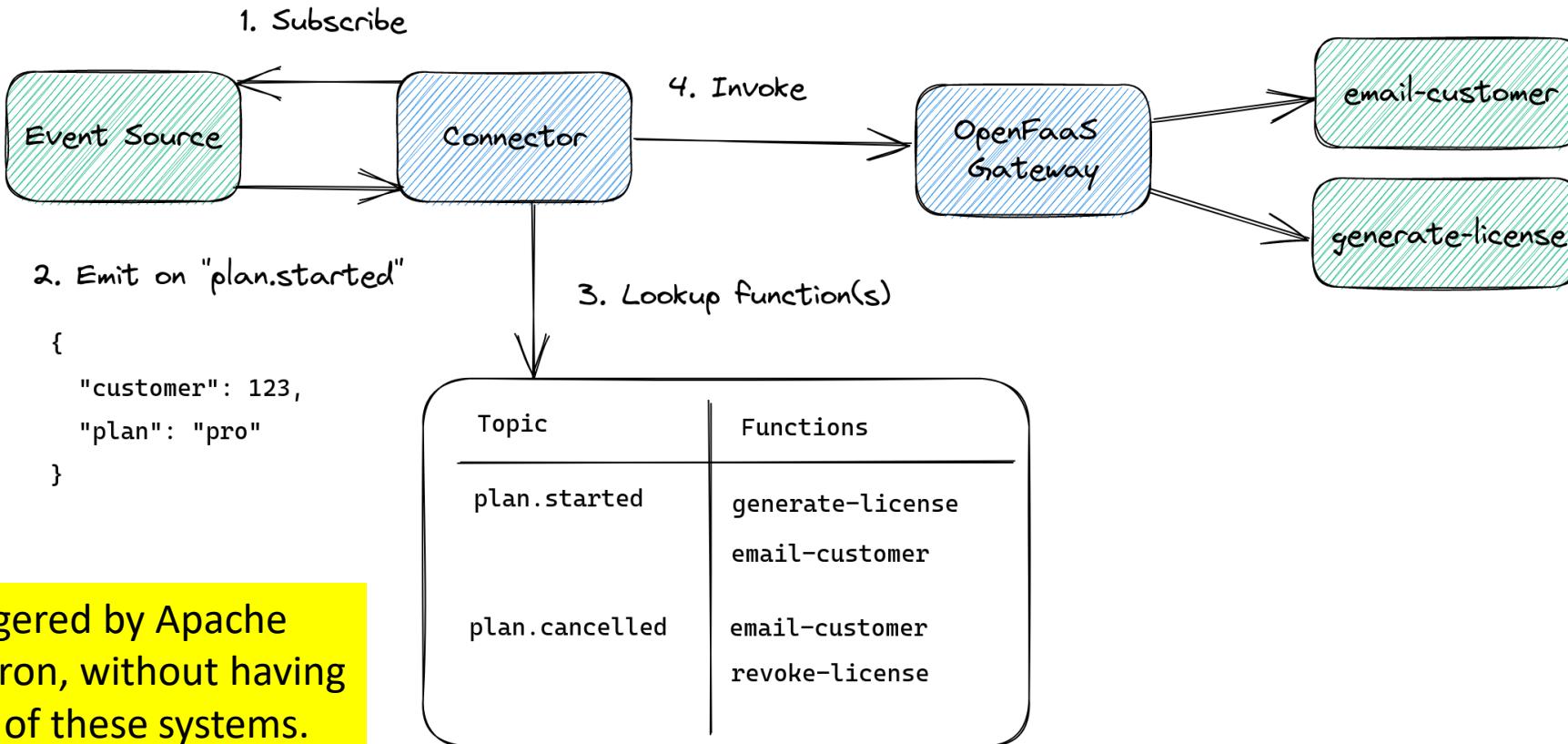
CLI¶

Trigger a function using the faas-cli by using the function name

Events



Event-connector pattern





Autoscaling

- Autoscaling is a feature available in OpenFaaS that scales up or scales down function replicas based on demand. This feature was built using both Prometheus and the Alert Manager components available with the OpenFaaS framework.
- Alert Manager will fire alerts when the function invocation frequency exceeds the defined threshold.
- While deploying the functions, the following labels are used to control the number of minimum replicas, maximum replicas, and the increase/decrease factor of the functions:
 - **com.openfaas.scale.min** – This defines the initial number of replicas, which is 1 by default.
 - **com.openfaas.scale.max** – This defines the maximum number of replicas.
 - **com.openfaas.scale.factor** – This defines the percentage of pod replica increase (or decrease) when the Alert Manager fires the alerts. By default, this is set to 20% and should have a value between 0 and 100.

Scaling modes



There are three auto-scaling modes: **Capacity, RPS, CPU**

- When configuring auto-scaling for a function, you need to set a target number which is the **average load per replica** of your function.
- **Each mode** can be used to record a current load for a function across all replicas in the OpenFaaS cluster.
- Then, a **query** is run periodically to calculate the current load.
- The current load is used to calculate the **new number of replicas**.

desired = ready pods * (mean load per pod / target load per pod)

<- **CEIL()**



Scaling modes

The target-proportion flag value, in the range [0,1], can be used to adjust scaling:

desired = ready pods * (mean load per pod / (target load per pod * **target-proportion**))

Scaling modes



- **Capacity** - Based upon inflight requests (or connections). Ideal for long-running functions or functions which can only handle a limited number of requests at once. A hard limit can be enforced through the `max_inflight` environment variable on the function.
- **RPS** - Based upon requests per second completed by the function. A good fit for functions which execute quickly and have **high throughput**. You can tune this value on a per function basis.
- **CPU** - Based upon CPU usage of the function. Ideal for CPU-bound workloads, or where Capacity and RPS are not giving the optimal scaling profile. The value configured here is in **milli-CPU**.
- **Scaling to zero** - Scaling to zero is an opt-in feature on a per function basis. It can be used in combination with any of the three scaling modes listed above.



Autoscaling Example

For example:

- sleep is running in the **capacity mode** and has a **target load of 5 in-flight requests**.
- The load on the sleep function is measured as **15 inflight requests**.
- There is only one replica of the sleep function because its **minimum range is set to 1**.
- We are assuming **com.openfaas.scale.target-proportion is set to 1.0 (100%)**.

$$\text{mean per pod} = 15 / 1$$

$$3 = \text{ceil} (1 * (15 / 5 * 1))$$

Therefore, 3 replicas will be set.



Autoscaling Example

With **3 replicas** and **25 ongoing requests**, the load will be spread more evenly, and evaluate as follows:

$$\text{mean per pod} = 25 / 3 = 8.33$$

$$5 = \text{ceil}(3 * (8.33 / 5 * 1))$$

When the load is no longer present, it will evaluate as follows:

$$\text{mean per pod} = 0 / 3 = 0 \quad 0 = \text{ceil} (3 * (0 / 5 * 1))$$

But the function will not be set to zero yet, it will be brought up to the minimum range which is 1.

Autoscaling Example



If you are limiting how much concurrency goes to a function, let's say for 100 requests maximum, then you may want to set the target to 100 with a proportion of 0.7, in this instance, when there are 70 ongoing requests, the autoscaler will add more replicas:

total load = 90

mean per pod = $90 / 1 = 90$

$2 = \text{ceil}(1 * (90 / (100 * 0.7)))$



Scaling up from zero replicas

The latency between accepting a request for an unavailable function and serving the request is sometimes called a "**Cold Start**".

- The cold start in OpenFaaS is strictly optional and it is recommended that for time-sensitive operations you avoid one by having a minimum scale of 1 or more replicas. This can be achieved by not scaling critical functions down to zero replicas, or by invoking them through the asynchronous route which decouples the request time from the caller.
- The "Cold Start" consists of the following: creating a request to schedule a container on a node, finding a suitable node, pulling the Docker image and running the initial checks once the container is up and running. This "running" or "ready" state also has to be synchronised between all nodes in the cluster. The total value can be reduced by pre-pulling images on each node and by setting the Kubernetes Liveness and Readiness Probes to run at a faster cadence.

Sources:

- Onur Yilmaz Sathsara Sarathchandra: «Serverless Architectures with Kubernetes». Packt Publishing, 2019.
- <https://docs.openfaas.com/>
- <https://docs.openfaas.com/architecture/autoscaling/>
- <https://github.com/openfaas/classic-watchdog/blob/master/README.md>
- <https://github.com/openfaas/of-watchdog/blob/master/README.md>
- <https://iximiuz.com/en/posts/openfaas-case-study/>

Processi di Markov e code markoviane

Gianluca Reali



Motivazioni

Supponiamo di dover affrontare le seguenti problematiche:

- Progettare una **cloud ibrida** facendo in modo che la **probabilità di offloading sia inferiore a un valore predifinito**
- Dato un sistema di calcolo distribuito, valutare **l'effetto della presenza di un broker sulla latenza media e massima di accesso a un servizio**
- Determinare **quanti server includere in un tenant** per far sì che le richieste di servizio possano essere **servite immediatamente con una probabilità predefinita**.
- Valutare quante indirizzi IP configurare in una **subnet** di OpenStack affinché la **probabilità di non trovarne disponibili** da parte di una istanza **sia inferiore a un valore predefinito**.
- Calcolare il numero **minimo e massimo di istanze** da configurare in un **hpa** di kubernetes affinché la **la latenza media di accesso al servizio sia inferiore a un valore predefinito**
- Calcolare il **numero minimo di connessioni** da include a **un pool predefinito** per la **connessione dal front-end al back-end di un servizio web** affinché la **probabilità di non trovarne disponibili sia inferiore a un valore predefinito**.
- In un **servizio faas in ambito edge**, determinare la **frequenza massima accettabile di accesso a una funzione** affiché la **latenza della risposta** sui **minore o uguale al valore massimo tollerabile**.

Obiettivi della lezione

Comprendere i concetti di base relativi alle **prestazioni** di un sistema dinamico, sia concentrato sia distribuito.

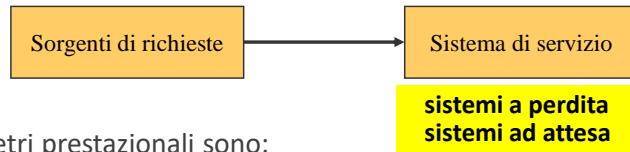
- efficienza
- utilizzazione
- ritardo
- perdita
- Tempi di inattività

Essere in grado di **valutare quantitativamente** tali parametri in casi semplici ma significativi --> il goal



Modello del sistema

Per valutare quantitativamente le prestazioni di un sistema dinamico è necessario rappresentare in modo astratto le sue funzionalità:



Tipici parametri prestazionali sono:

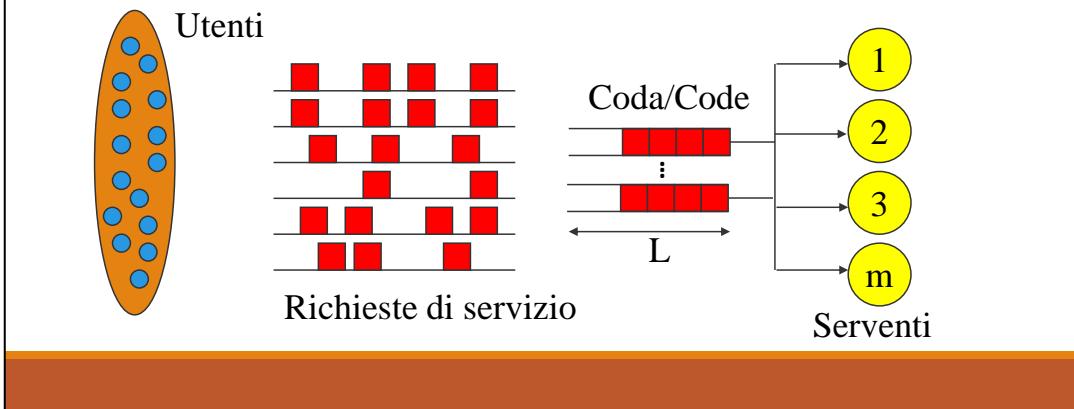
- sistemi a perdita:
 - frequenza con cui le richieste sono rifiutate
 - intensità del carico smaltito
 - durata dei periodi di congestione
- sistemi ad attesa:
 - tempo (...?) di attesa
 - tempo (...?) di permanenza
 - numero (...?) di richieste in attesa



Sistema di servizio

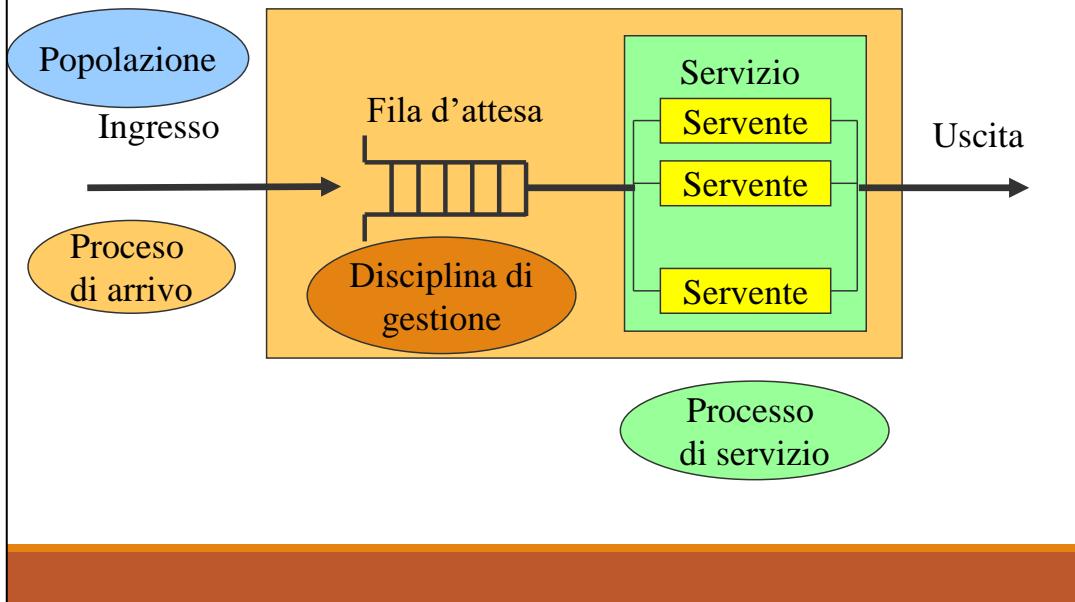
Risoluzione delle contese di utilizzazione:

- sistema a perdita pura ($L=0$)
- sistema orientato alla perdita (L piccolo)
- sistema orientato al ritardo (L grande)
- sistema a ritardo senza perdita ($L \rightarrow \infty$ oppure $L \geq \text{utenti}$)





Caratterizzazione dei Sistemi a coda





Elementi descrittivi del sistema a coda

Cardinalità della popolazione:

- finita, infinita

Processo di Arrivo :

- frequenza media, varianza ...

Accodamento:

- dimensione della coda:
 - finita, infinita
 - numero di code

Disciplina di gestione, o Selezione:

- discipline di coda
 - primo arrivato primo servito (FIFO)
 - shortest job first (SJF)
 - ...
- classi di priorità

Numero dei serventi

Fissiamo le grandezze principali:

Caratterizzazione del servizio

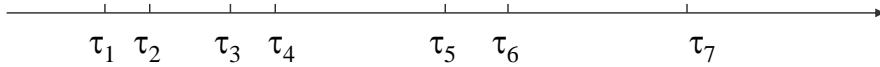
- Siano
 - n = cardinalità della popolazione
 - L = dimensione della coda
 - m = numero di serventi
 - $C=m+L$ capacità del sistema
- Classificazione:
 - se $n \leq C$ e $L > 0$: sistema ad attesa (senza perdita)
 - se $n > C$ e $L > 0$: sistema a perdita (con attesa)
 - se $n > C$ e $L = 0$: sistema a perdita pura (senza attesa)
- Tipico aarametro prestazionale:
 - $\text{Tempo di sistema (s)} = \text{tempo di attesa (w)} + \text{tempo di servizio (x)}$



Caratterizzazione della domanda

La domanda è caratterizzata dalle richieste di servizio presentate dagli utenti del sistema

- consideriamo una sequenza di istanti di richiesta di servizio (τ_i)



- tali istanti sono distribuiti secondo una specifica descrizione statistica sull'asse dei tempi e costituiscono un insieme numerabile

Tempo di interarrivo:

- i -esimo tempo di interarrivo t_i è l'intervallo che intercorre tra l'istante di presentazione della richiesta $(i-1)$ -esima (τ_{i-1}) e quello della richiesta i -esima (τ_i)

$$t_i = \tau_i - \tau_{i-1} \quad i = 1, 2, \dots$$



Tempo di servizio

Sia L_i la ‘quantità di lavoro’, espresso in modo quantitativo, che i serventi del sistema devono erogare per soddisfare la richiesta i -esima

Definiamo l' i -esimo tempo di servizio x_i , come l’intervallo di tempo che un servente impiega per soddisfare la richiesta i -esima

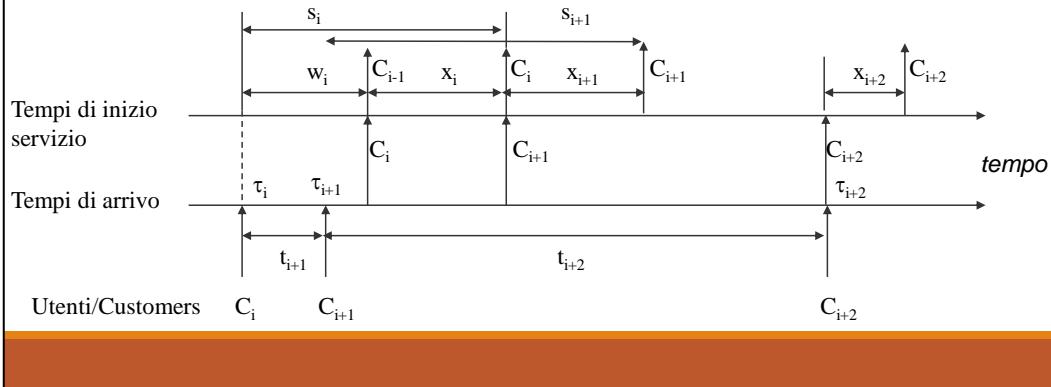
Se supponiamo che la capacità di erogare lavoro Γ (quantità di lavoro erogato nell’unità di tempo) di ogni servente del sistema sia identica, per servire la richiesta i -esima sarà necessario un tempo uguale a

$$x_i = L_i / \Gamma \quad i = 1, 2, \dots$$



Diagramma temporale del sistema

- C_i : *i-esima* richiesta di servizio (o *cliente*) ad entrare nel sistema
- Grandezze temporali:
 - τ_i : tempo di arrivo *i-esima* richiesta di servizio
 - t_i : tempo di interarrivo tra la richiesta (*i-1*-esima) e la *i-esima*
 - w_i : tempo di attesa in coda della *i-esima* richiesta di servizio
 - x_i : tempo di servizio dell'*i-esima* richiesta
 - s_i : tempo di sistema dell'*i-esima* richiesta



Processi di ingresso e di servizio

La sequenza dei tempi di interarrivo $\{t_i\}$ e quella dei tempi di servizio $\{x_i\}$ costituiscono delle realizzazioni di due processi stocastici:

- il processo di ingresso
- il processo di servizio

Ogni valore t_i e x_i è una realizzazione di una variabile aleatoria

Normalmente si suppone i due processi siano stazionari, almeno WSS, e statisticamente indipendenti

N.B.



Grandezze limite

Comportamento al limite delle variabili aleatorie:

- tempi di interarrivo

$$\tilde{t} = \lim_{n \rightarrow \infty} t_n$$

$$P[t_n \leq t] = A_n(t) \xrightarrow{n \rightarrow \infty} P[\tilde{t} \leq t] = A(t)$$

$$E[t_n] = \bar{t}_n \xrightarrow{n \rightarrow \infty} E[\tilde{t}] = \bar{t} = \frac{1}{\lambda}$$

Analogamente si possono definire le stesse grandezze per

- i tempi di attesa

$$\tilde{w} = \lim_{n \rightarrow \infty} w_n, \quad E[\tilde{w}] = \bar{w} = W$$

- i tempi di servizio

$$\tilde{x} = \lim_{n \rightarrow \infty} x_n, \quad E[\tilde{x}] = \bar{x} = \frac{1}{\mu}$$

- i tempi di sistema

$$\tilde{s} = \lim_{n \rightarrow \infty} s_n, \quad E[\tilde{s}] = \bar{s} = T$$



Caratterizzazione di un sistema a coda

Una coda è definita da:

- processo degli arrivi (D.d.p.) $A(t)$
- tempi di servizio (D.d.p.) $B(t)$
- numero di serventi m
- dimensione del sistema L
- cardinalità della popolazione n
- disciplina di servizio

Notazione sintetica di Kendall ($A/B/m/L/n$)

- A e B posono assumere i valori:
 - M esponenziale negativa o “Markoviana”
 - D deterministica o costante
 - E_i erlangiana con i stadi
 - H_i iper-esponenziale con i stadi
 - G generale

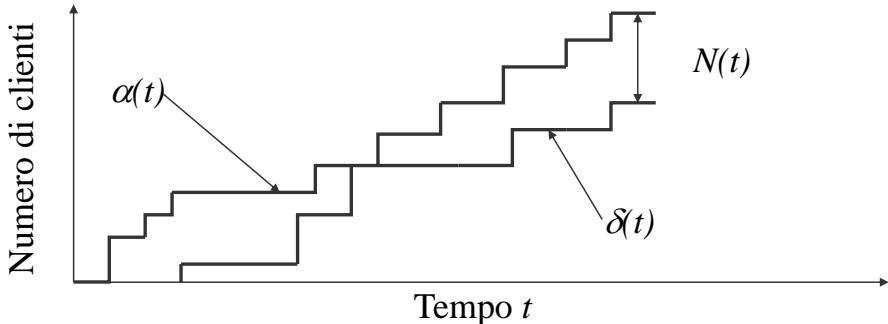
N.B.



Legge di Little

Ipotesi: sistema senza perdite

- $\alpha(t)$: numero di arrivi in $[0,t]$
- $\delta(t)$: numero di partenze in $[0,t]$
- $N(t) = \alpha(t) - \delta(t)$
- $\gamma(t)$: l'area tra le due curve rappresenta il tempo che tutti i clienti hanno trascorso nel sistema nell'intervallo $[0,t]$



Legge di Little

Definiamo:

- frequenza media di interarrivo in $[0,t]$: $\lambda_t = \alpha(t)/t$ (a)
- tempo medio di sistema per ogni utente in $[0,t]$: $T_t = \gamma(t)/\alpha(t)$ (b)
- numero medio di clienti nel sistema in $[0,t]$: $N_t = \gamma(t)/t$

Quindi si ottiene:

- $N_t = \gamma(t)/t$; dalla (a) $N_t = \gamma(t) \lambda_t / \alpha(t)$; dalla (b) $N_t = T_t * \lambda_t$
- assumendo che esistano per il sistema i valori limite: $\lambda = \lim_{t \rightarrow \infty} \lambda_t$, $T = \lim_{t \rightarrow \infty} T_t$
- si ottiene:

Super
N.B.

$$\bar{N} = \lambda T = \lambda W + \lambda \bar{x} = \bar{N}_q + \bar{N}_s$$

Risultato indipendente da A(t), B(t) e m

IPOTESI: ASSENZA DI PERDITE!

Fattore di utilizzazione

Definizione: rappporto tra la frequenza con la quale di 'lavoro' entra nel sistema e quella con la quale i serventi riescono a smaltirlo

Assumendo che la frequenza di erogazione del servizio sia indipendente da qualsiasi altro parametro del sistema, possiamo scrivere:

- caso 1 servente: $\rho = \lambda / \mu = \lambda \bar{x}$
- caso m serventi: $\rho = \lambda / (m\mu) = \lambda \bar{x} / m$

λ/μ , ossia il ritmo con cui il lavoro entra nel sistema, normalizzato alla capacità del singolo servente, è l'intensità di traffico (espressa in Erlang), vale $m\rho$

Fattore di utilizzazione

Se $0 \leq \rho < 1$:

- può essere interpretato come:
 - frazione di serventi occupati*
- rappresenta la condizione di stabilità del sistema. Infatti, nel caso G/G/m, risulta quanto segue:
 - dato un intervallo τ sufficientemente grande, con probabilità tendente a 1 il numero di arrivi sarà pari a $\lambda\tau$
 - quindi un servente sarà occupato per un tempo pari a $\tau(1-p_0)$, definendo p_0 come la probabilità di trovare il server libero in un generico istante di tempo
 - il numero di utenti serviti in tale intervallo sarà $m(\tau - \tau p_0)\mu$
 - eguagliando il numero di arrivi con il numero di utenti serviti (condizione di stabilità), si ottiene:

$$\lambda\tau \cong m(\tau - \tau p_0)\mu \xrightarrow{\tau \rightarrow \infty} \rho = 1 - p_0$$



Concetto di Stato

- Intuitivamente, le prestazioni osservate da un utente generico (ossia che non gode di diritti di prelazione dei serventi) che entra in un sistema a coda dipendono da quanti utenti trova già presenti nel sistema.
- Le prestazioni, quindi, dipendono dalla storia più o meno recente del sistema.
- Ciò significa che quanto accaduto in passato ha lasciato traccia nel sistema, traccia che influenzera il futuro del sistema stesso, e che chiameremo **STATO** del sistema.

Concetto di stato

La probabilità che il sistema si trovi in un generico stato j al tempo t è uguale a:

$$\Pi_j(t) = P[X(t)=j]$$

Sia la variabile t sia lo stato $X(t)$ possono assumere valori in un insieme continuo o in un insieme discreto (finito o numerabile)

Noi considereremo processi **continui** nel tempo e **discreti** nelle realizzazioni

Chiaramente $\sum_{j=0,1,\dots} \Pi_j(t) = 1$

Catene di Markov

Un processo aleatorio è detto Catena di Markov se lo spazio degli stati è discreto e gode della proprietà di Markov

Proprietà di Markov:

- dato un insieme di variabili aleatorie $\{X_n\}$, questo forma una catena di Markov se la probabilità di trovarsi in un tempo futuro in un determinato stato può essere espressa in funzione solo dello stato assunto al tempo corrente e non occorre specificare quali stati sono stati assunti in precedenza:
- lo stato attuale riassume tutta la storia del sistema
- la conoscenza più recente dello stato del sistema rende inutile la conoscenza degli stati assunti in precedenza
- la conoscenza del passato non ci consente di predire quanto tempo il processo debba rimanere nello stato in cui si trova
 - la distribuzione del tempo che il processo rimane in uno stato è “senza memoria”, e nell’ipotesi d tempo continuo quest porta alla distribuzione esponenziale del tempo di permanenza nello stato.

N.B.

Catene di Markov

Formalmente:

- il processo aleatorio $X(t)$ forma una catena di Markov tempo-continua se per tutti gli interi n e per una sequenza di istanti temporali $t_1 < t_2 < \dots < t_n < t_{n+1}$ risulta

$$P[X(t_{n+1})=j \mid X(t_n)=i_n, X(t_{n-1})=i_{n-1}, \dots, X(t_1)=i_1] = P[X(t_{n+1})=j \mid X(t_n)=i_n]$$

Classificazione:

- Catene di Markov tempo continuo:
 - distribuzione del tempo di stato esponenziale $p_T(t)=\lambda e^{-\lambda t}$, $t \geq 0$
- Catene di Markov tempo discreto
 - distribuzione del tempo di stato geometrica $p_N(n)=(1-p)^{n-1}p$, $n \geq 0$

Catene di Markov

Probabilità di transizione da uno stato $i \rightarrow j$:

$$p_{ij}(s,t) = P[X(t) = j \mid X(s) = i] \text{ per } t \geq s$$

per passare dallo stato i all'istante s allo stato j all'istante $t > s$, il processo dovrà passare per uno stato intermedio k ad un certo istante intermedio u :

$$\begin{aligned} p_{ij}(s,t) &= \sum_k P[X(t) = j, X(u) = k \mid X(s) = i] = \\ &= \sum_k P[X(u) = k \mid X(s) = i] P[X(t) = j \mid X(s) = i, X(u) = k] = \\ &\equiv \sum_k P[X(u) = k \mid X(s) = i] P[X(t) = j \mid X(u) = k] = \\ &= \sum_k p_{ik}(s,u) p_{kj}(u,t) \end{aligned}$$

Equazioni di Chapman-Kolmogorov

$$\sum_B P(A, B \mid C) = P(A \mid C) \quad P(A \mid B, C) P(B \mid C) = \frac{P(A, B, C)}{P(B, C)} \frac{P(B, C)}{P(C)} = P(A, B \mid C)$$

Catene di Markov



$$p_{ij}(s,t) = \sum_k p_{ik}(s,u)p_{kj}(u,t) = \\ = \begin{bmatrix} p_{11}(s,u) & p_{12}(s,u) & \dots & p_{1k}(s,u) & \dots \\ p_{21}(s,u) & p_{22}(s,u) & \dots & p_{2k}(s,u) & \dots \\ \vdots & \vdots & & \vdots & \dots \\ p_{i1}(s,u) & p_{i2}(s,u) & \dots & p_{ik}(s,u) & \dots \\ \vdots & \vdots & & \vdots & \dots \end{bmatrix} \begin{bmatrix} p_{11}(u,t) & p_{12}(u,t) & \dots & p_{1j}(u,t) & \dots \\ p_{21}(u,t) & p_{22}(u,t) & \dots & p_{2j}(u,t) & \dots \\ \vdots & \vdots & & \vdots & \dots \\ p_{i1}(u,t) & p_{i2}(u,t) & \dots & p_{kj}(u,t) & \dots \\ \vdots & \vdots & & \vdots & \dots \end{bmatrix}$$

$$H(s,t) = H(s,u)H(u,t)$$

Equazioni di C-K in forma matriciale

$$H(s,t) = \begin{bmatrix} p_{11}(s,t) & p_{12}(s,t) & \dots & p_{1k}(s,t) & \dots \\ p_{21}(s,t) & p_{22}(s,t) & \dots & p_{2k}(s,t) & \dots \\ \vdots & \vdots & & \vdots & \dots \\ p_{i1}(s,t) & p_{i2}(s,t) & \dots & p_{ik}(s,t) & \dots \\ \vdots & \vdots & & \vdots & \dots \end{bmatrix}$$

Catene di Markov

In forma matriciale:

$$H(s,t) = [p_{ij}(s,t)] = H(s,u)H(u,t), \text{ con } H(t,t) = I$$

definendo $P(t) = H(t, t + \Delta t) = [p_{ij}(t, t + \Delta t)]$ si ottiene:

$$\begin{aligned} H(s,t) - H(s,t - \Delta t) &= H(s,t - \Delta t)H(t - \Delta t, t) - H(s,t - \Delta t) = \\ &= H(s,t - \Delta t)P(t - \Delta t) - H(s,t - \Delta t) = H(s,t - \Delta t)(P(t - \Delta t) - I) \end{aligned}$$

dividendo per Δt e facendo il limite tendente a zero:

$$\lim_{\Delta t \rightarrow 0} \frac{H(s,t) - H(s,t - \Delta t)}{\Delta t} =$$

$$\frac{\partial H(s,t)}{\partial t} = \lim_{\Delta t \rightarrow 0} \left(H(s,t - \Delta t) \underbrace{\frac{(P(t - \Delta t) - I)}{\Delta t}}_{\Delta t} \right) = H(s,t)Q(t)$$

$$Q(t) = \lim_{\Delta t \rightarrow 0} \frac{P(t - \Delta t) - I}{\Delta t} = [q_{ij}(t)], \text{ con } q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$$

N.B.



Catene di Markov

$$\frac{\partial H(s,t)}{\partial t} = H(s,t)Q(t)$$

Equazioni di C-K in forma differenziale

$$Q(t) = [q_{ij}(t)], \quad q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$$

Q è nota come **generatore infinitesimale** or **rate matrix** del processo di Markov.

È evidente che

$$\sum_j q_{ij}(t) = 0 \quad \Rightarrow q_{ii}(t) = - \sum_{j \neq i} q_{ij}(t) < 0$$

Catene di Markov

Pertanto:



$$Q(t) = \begin{bmatrix} -\sum_{j \neq 1} q_{1j}(t) & q_{12}(t) & \dots & q_{1N}(t) \\ q_{21}(t) & -\sum_{j \neq 2} q_{2j}(t) & \dots & q_{2N}(s, t) \\ \vdots & \vdots & & \vdots \\ q_{N1}(t) & q_{N2}(t) & \dots & -\sum_{j \neq N} q_{Nj}(t) \end{bmatrix}$$

Perché la matrice $Q(t)$ è detta “rate” matrix?

Perché il valore degli elementi q_{ij} sono in relazione con la frequenza degli eventi che determinano l’evoluzione del processo di Markov process.



Catene di Markov

$$q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$$

~~$p_{ij}(t - \Delta t, t) - p_{ij}(t, t)$~~

Quindi:

$$p_{ij}(t, t + \Delta t) \approx |q_{ij}(t)| \Delta t$$

$$1 - p_{ij}(0, t + \Delta t) = p_0(0, t + \Delta t) = p_0(0, t)(1 - p_{ij}(t, t + \Delta t)) \approx p_0(0, t)(1 - |q_{ij}(t)| \Delta t)$$

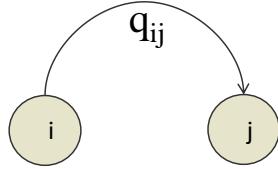
$$\lim_{\Delta t \rightarrow 0} \frac{p_0(0, t + \Delta t) - p_0(0, t)}{\Delta t} = \frac{dp_0(0, t)}{dt} \approx -|q_{ij}(t)| p_0(0, t)$$

$$p_0(0, t) = e^{-\int_0^t |q_{ij}(t')| dt} \quad p_{ij}(0, t) = 1 - e^{-\int_0^t |q_{ij}(t')| dt} \quad t \geq 0$$

$$\text{if } q_{ij}(t) = q_{ij} \quad p_{ij}(0, t) = 1 - e^{-|q_{ij}|t}, \quad \bar{T}_{ij} = \frac{1}{|q_{ij}|}$$

se costanti, la distribuzione di prob di transizione di stato è un'exp

q_{ij} risulta essere uguale alla frequenza media di transizione di stato



Catene di Markov



Equazioni di Chapman-Kolmogorov:

- forma matriciale

- in avanti $\frac{\partial H(s,t)}{\partial t} = H(s,t)Q(t) \quad s \leq t$

- all'indietro $\frac{\partial H(s,t)}{\partial s} = -Q(s)H(s,t) \quad s \leq t$

- termine a termine

- in avanti

$$\frac{\partial p_{ij}(s,t)}{\partial t} = q_{jj}(t)p_{ij}(s,t) + \sum_{k \neq j} q_{kj}(t)p_{ik}(s,t) \quad \text{con } p_{ij}(s,s) = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$$

- all'indietro

$$\frac{\partial p_{ij}(s,t)}{\partial s} = -q_{ii}(t)p_{ij}(s,t) - \sum_{k \neq j} q_{ik}(s)p_{kj}(s,t) \quad \text{con } p_{ij}(t,t) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$



Catene di Markov

Soluzione alle due equazioni:

$$H(s, t) = e^{\int_s^t Q(u) du}$$

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots$$

La probabilità di fare una transizione da $i \rightarrow j$ nell'intervallo $(t, t + \Delta T)$ è data da:

$$p_{ij}(t, t + \Delta t) = q_{ij}(t) \Delta t + o(\Delta t) \quad o(\Delta t) \Leftrightarrow \lim_{\Delta t \rightarrow 0} \frac{o(\Delta t)}{\Delta t} = 0$$

La probabilità di uscire dallo stato i -esimo nell'intervallo $(t, t + \Delta T)$ è data da:

$$1 - p_{ii}(t, t + \Delta t) = \Delta t \sum_{j \neq i} q_{ij}(t) + o(\Delta t) = -q_{ii}(t) \Delta t + o(\Delta t)$$

credo abbia ripreso questa di slide 28

$$p_{ij}(t, t + \Delta t) \cong |q_{ij}(t)| \Delta t$$

Catene di Markov



Probabilità dello stato j : $\pi_j(t) = P[X(t)=j]$

È immediato scrivere che per $t \geq s$

$$\pi_j(t) = \sum_i \pi_i(s) p_{ij}(s, t) = [\pi_1(s) \quad \pi_2(s) \quad \dots] \begin{bmatrix} p_{1j}(s, t) \\ p_{2j}(s, t) \\ \vdots \\ p_{ij}(s, t) \end{bmatrix}$$

$$\pi^T(t) = \pi^T(s) H(s, t)$$

$$\frac{d\pi^T(t)}{dt} = \pi^T(s) \frac{\partial H(s, t)}{\partial t} = \pi^T(s) H(s, t) Q(t) = \pi^T(t) Q(t)$$

Forward Chapman-Kolmogorov equations

$$\frac{d\pi(s)}{ds} = Q(s)\pi(s) \quad \text{Backward Chapman-Kolmogorov equations}$$



Catene di Markov Omogenee

Una catena di Markov tempo-continua è detta **omogenea** se le probabilità di transizione fra due stati qualsiasi, in un dato intervallo di tempo, non dipende dall'istante di inizio di tale intervallo ma solo dalla sua durata:

Per catene di Markov omogenee

$$p_{ij}(s,t) = p_{ij}(\tau) \Rightarrow H(s,t) = H(\tau) = [p_{ij}(\tau)], \quad \tau = t - s$$
$$q_{ij}(t) = q_{ij} \quad i,j = 1,2,\dots \Rightarrow Q(t) = Q = [q_{ij}]$$

Le **equazioni** di Chapman-Kolmogorov diventano

$$\frac{d\pi^T(t)}{dt} = \pi^T(t)Q$$

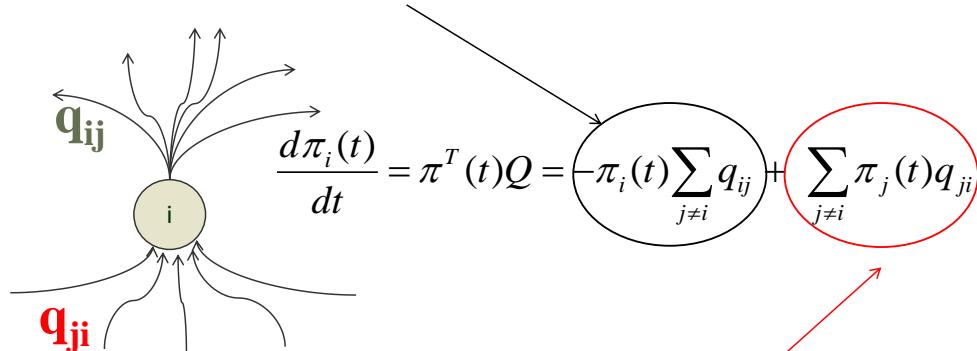
E' intuitivo il fatto che se la frequenza di transizione fra due stati in un certo intervallo di tempo non dipende dall'istante considerato, allora la frequenza di transizione di stato si mantiene costante nel tempo.



Catene di Markov Omogenee

Le equazioni di Chapman-Kolmogorov si possono scrivere facilmente mediante ispezione diretta:

Flusso di probabilità verso altri stati



Flusso di probabilità dagli altri stati

33

Ricorda come era fatta Q (togli la dipendenza dal tempo bc siamo in caso di OMOGENEITA')

$$Q(t) = \begin{bmatrix} -\sum_{j \neq 1} q_{1j}(t) & q_{12}(t) & \dots & q_{1N}(t) \\ q_{21}(t) & -\sum_{j \neq 2} q_{2j}(t) & \dots & q_{2N}(t) \\ \vdots & \vdots & \ddots & \vdots \\ q_{N1}(t) & q_{N2}(t) & \dots & -\sum_{j \neq N} q_{Nj}(t) \end{bmatrix}$$

Catene di Markov Omogenee

Catena di Markov irriducibile:

se \circ ogni stato è raggiungibile da qualsiasi altro: $p_{ij}(t) > 0$

$$\lim_{t \rightarrow \infty} p_{ij}(t) = p_j$$

Catena di Markov ergodica:

- \circ le probabilità di stato convergono ad un valore limite, indipendente dalla distribuzione iniziale degli stati

$$\lim_{t \rightarrow \infty} \pi_j(t) = \pi_j = p_j$$

L'irriducibilità garantisce che tutti gli stati debbano avere probabilità positiva di essere visitati in qualche istante, l'ergodicità garantisce che il comportamento statistico della catena non dipende dallo stato di partenza. In questo modo il comportamento statistico di tutte le realizzazioni del processo di Markov è lo stesso, pertanto le statistiche temporali coincidono con quelle d'insieme.



Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

$$\frac{d\pi(t)}{dt} = \pi(t)A \quad \text{A matrice generica}$$

Caso monidimensionale, $A = a$, scalare: $\pi(t) = \pi_0 e^{at}$

$$\frac{de^{at}}{dt} = e^{at}a = ae^{at}$$

$$e^{at} = 1 + at + \frac{1}{2!}(at)^2 + \cdots \frac{1}{k!}(at)^k + \cdots = \sum_{k=0}^{\infty} \frac{(at)^k}{k!}$$

$$k! = k \times (k-1) \times \cdots \times 2 \times 1$$

$$\begin{aligned} \frac{de^{at}}{dt} &= \frac{d(1 + at + \frac{1}{2}(at)^2 + \cdots \frac{1}{k!}(at)^k + \cdots)}{dt} \\ &= \frac{0 + a + \frac{2}{2}(at)a + \cdots \frac{k}{k!}(at)^{k-1}a + \cdots}{dt} = ae^{at} \end{aligned}$$



Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

Caso multidimensionale: $\pi(t) = \pi_0 e^{At}$

L'esponenziale di matrice e^{At} è definito come:

$$e^{At} = I + At + \frac{1}{2!} A^2 t^2 + \cdots + \frac{1}{k!} A^k t^k + \cdots = \sum_{k=0}^{\infty} \frac{A^k t^k}{k!}$$

$$\frac{de^{At}}{dt} = \frac{de^{At}}{d(At)} \frac{d(At)}{dt} = Ae^{At} = e^{At} A$$



Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

$$\begin{aligned}\frac{d}{dt} e^{At} &= \frac{d}{dt} (I + At + \frac{1}{2!} A^2 t^2 + \cdots + \frac{1}{k!} A^k t^k + \cdots) \\&\equiv 0 + A + \frac{1}{2} 2tA^2 + \cdots + \frac{1}{k!} kt^{k-1} A^k + \cdots \\&\equiv A + A^2 t + \cdots + \frac{1}{(k-1)!} A^k t^{k-1} + \cdots \\&\equiv A(I + At + \cdots + \frac{1}{(k-1)!} A^{k-1} t^{k-1} + \cdots) \\&\equiv A \sum_{k=0}^{\infty} \frac{A^k t^k}{k!} = Ae^{At} \\&\equiv (I + At + \cdots + \frac{1}{(k-1)!} A^{k-1} t^{k-1} + \cdots)A \\&\equiv (\sum_{k=0}^{\infty} \frac{A^k t^k}{k!})A = e^{At} A\end{aligned}$$



Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

$$\text{Curved arrow pointing down} \quad \frac{d\pi(t)}{dt} = \pi(t)A \quad \Rightarrow \quad \pi(t) = \pi_0 e^{At}$$

$$s\pi(s) - \pi_0 = \pi(s)A$$

$$\pi(s) = \pi_0 (sI - A)^{-1}$$

$$\pi(t) = \pi_0 L^{-1}[(sI - A)^{-1}]$$

$$e^{At} = L^{-1}[(sI - A)^{-1}]$$

$$(sI - A)^{-1} = L[e^{At}] = L[I + At + \dots + \frac{1}{k!} A^k t^k + \dots]$$

$$= \frac{I}{s} + \frac{A}{s^2} + \frac{A^2}{s^3} + \dots + \frac{A^k}{s^{k+1}} + \dots = \sum_{k=0}^{\infty} \frac{A^k}{s^{k+1}}$$

Proprietà dell'esponenziale di matrice

"rivediamole"

$$e^{At} \Big|_{t=0} = I \quad [e^{At}]^{-1} = e^{-At}$$

$$e^{A(t_1+t_2)} = e^{At_1} e^{At_2} = e^{At_2} e^{At_1}$$

$$e^{A(t-t)} = e^{At} e^{-At} = e^{-At} e^{At} = I$$

$$e^{At} e^{Bt} = e^{(A+B)t} \quad \text{solo se A e B commutano}$$

Esempio



Si consideri la matrice sottostante.

$$\dot{x} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} x \quad \mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

Si ha che

$$\mathbf{A}^2 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 2^2 \end{pmatrix}, \quad \mathbf{A}^3 = \begin{pmatrix} 1 & 0 \\ 0 & 2^2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 2^3 \end{pmatrix}, \dots$$

In generale,

$$\mathbf{A}^n = \begin{pmatrix} 1 & 0 \\ 0 & 2^n \end{pmatrix}$$

Quindi

$$e^{\mathbf{At}} = \sum_{n=0}^{\infty} \frac{\mathbf{A}^n t^n}{n!} = \sum_{n=0}^{\infty} \begin{pmatrix} 1/n! & 0 \\ 0 & 2^n/n! \end{pmatrix} t^n = \begin{pmatrix} e^t & 0 \\ 0 & e^{2t} \end{pmatrix}$$

Esempio

$$\dot{x} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} x \quad [sI - A] = \begin{bmatrix} s-1 & 0 \\ 0 & s-2 \end{bmatrix}$$

$$[sI - A]^{-1} = \begin{bmatrix} \frac{s-2}{(s-1)(s-2)} & 0 \\ 0 & \frac{s-1}{(s-1)(s-2)} \end{bmatrix}$$

$$\Phi(t) = L^{-1}\{[sI - A]^{-1}\} = \begin{bmatrix} e^t & 0 \\ 0 & e^{2t} \end{bmatrix}$$



Catene di Markov Omogenee in Equilibrio

Le probabilità limite (ossia in condizione di **equilibrio statistico**) sono le soluzioni del sistema lineare:

N.B.

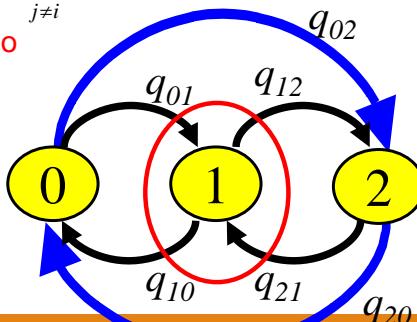
$$\left\{ \begin{array}{l} q_{jj}\pi_j + \sum_{k \neq j} q_{kj}\pi_k = 0 \\ \sum_j \pi_j = 1 \end{array} \right.$$

oppure

$$\left\{ \begin{array}{l} \pi Q = 0 \\ \sum_j \pi_j = 1 \end{array} \right.$$

$$q_{ii}(t) = -\sum_{j \neq i} q_{ij}(t) < 0$$

Esempio



$$\begin{cases} q_{00}\pi_0 + q_{10}\pi_1 + q_{20}\pi_2 = 0 \\ q_{11}\pi_1 + q_{01}\pi_0 + q_{21}\pi_2 = 0 \\ q_{22}\pi_2 + q_{02}\pi_0 + q_{12}\pi_1 = 0 \\ \pi_0 + \pi_1 + \pi_2 = 1 \end{cases}$$

$$\begin{cases} q_{00} = -q_{01} - q_{02} \\ q_{11} = -q_{10} - q_{12} \\ q_{22} = -q_{20} - q_{21} \end{cases}$$



Processi di nascita e morte

Catene di Markov in cui sono permesse, da un generico stato j , soltanto transizioni verso gli stati $j+1$ (nascita) e $j-1$ (morte), definendo

$$P_k(t) = p_k(t) = P[X(t)=k]$$

Consideriamo il caso di una catena di Markov omogenea; definiamo:

$$\begin{cases} \lambda_k = q_{k,k+1} \\ \mu_k = q_{k,k-1} \\ q_{k,k} = -(\lambda_k + \mu_k) \text{ da } \sum_j q_{kj} = 0 \\ q_{k,j} = 0 \text{ per } |k-j| > 1 \end{cases}$$
$$Q = \begin{bmatrix} -\lambda_0 & \lambda_0 & 0 & 0 & \dots \\ \mu_1 - (\lambda_1 + \mu_1) & \lambda_1 & 0 & \dots \\ 0 & \mu_2 - (\lambda_2 + \mu_2) & \lambda_2 & \dots \\ \vdots & & & \end{bmatrix}$$



Processi di nascita e morte

La distribuzione di probabilità degli stati si ottiene risolvendo le seguenti equazioni:

$$\frac{d\pi^T(t)}{dt} = \pi^T(t)Q$$

$$\begin{cases} \sum_{k=0}^{\infty} P_k(t) = 1 \\ \frac{dP_k(t)}{dt} = -(\lambda_k + \mu_k)P_k(t) + \lambda_{k-1}P_{k-1}(t) + \mu_{k+1}P_{k+1}(t) \\ \frac{dP_0(t)}{dt} = -\lambda_0P_0(t) + \mu_1P_1(t) \\ \text{Condizioni iniziali } P_k(0), k = 0, 1, \dots \end{cases}$$





Processi di nascita e morte

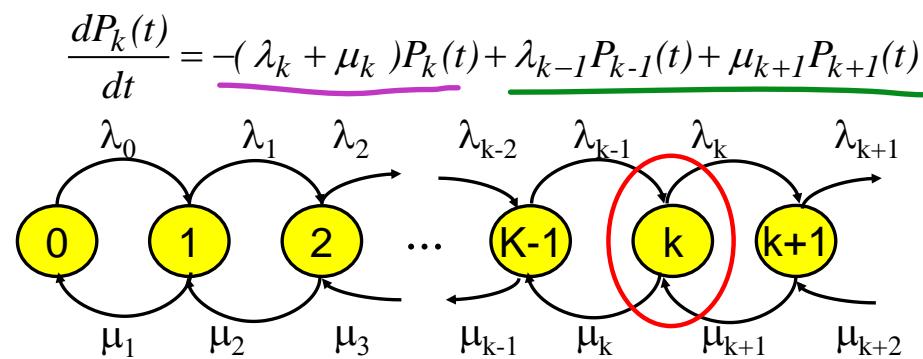
... che si possono vedere anche per **ispezione visiva**, facendo il **bilancio** dei flussi di probabilità:

- Flusso entrante nello stato k : $\lambda_{k-1}P_{k-1}(t) + \mu_{k+1}P_{k+1}(t)$
- Flusso uscente dallo stato k : $-(\lambda_k + \mu_k)P_k(t)$

La differenza tra queste due quantità rappresenta il tasso di variazione di probabilità dello stato k :

N.B. {

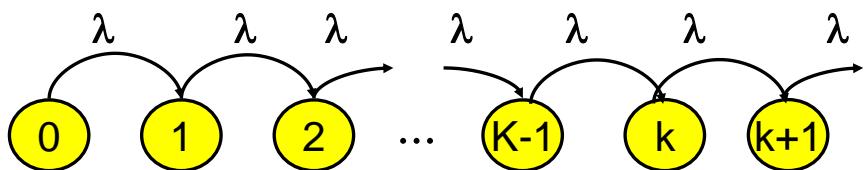
quindi questa è la formula generale della derivata rispetto al tempo dello stato k -esimo





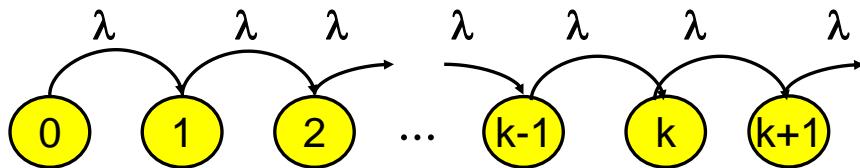
Processi di Poisson

Caratteristiche: catena di Markov di pura nascita a tasso costante (λ)





Processi di Poisson



$$\begin{cases} \frac{dP_0(t)}{dt} = -\lambda P_0(t) \\ \frac{dP_k(t)}{dt} = -\lambda P_k(t) + \lambda P_{k-1}(t), \quad k \geq 1 \end{cases}$$

$P_0(0)=1$ Condizioni iniziali

$$P_0(t) = e^{-\lambda t} \rightarrow \frac{dP_1(t)}{dt} = -\lambda P_1(t) + \lambda e^{-\lambda t} \quad t \geq 0$$

$$P_1(t) = \lambda t e^{-\lambda t} \rightarrow \frac{dP_2(t)}{dt} = -\lambda P_2(t) + \lambda^2 t e^{-\lambda t} \dots$$

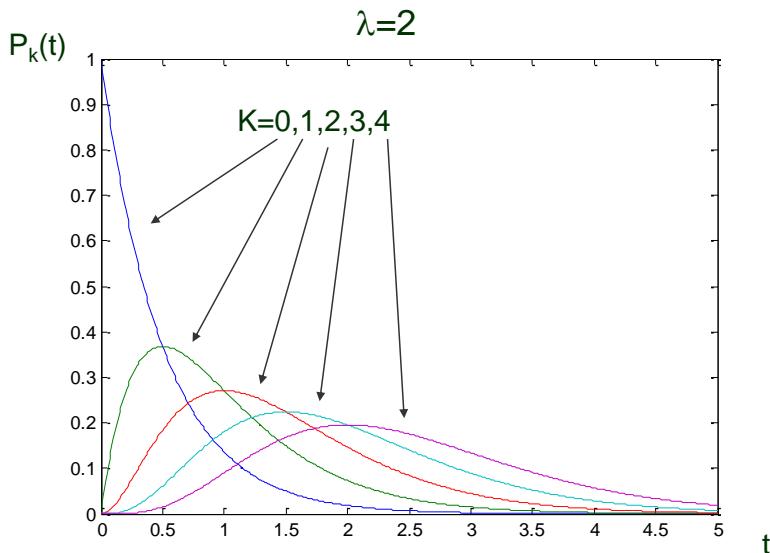
$$P_2(t) = \frac{(\lambda t)^2}{2} e^{-\lambda t} \rightarrow \dots$$

$$P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad t \geq 0 \quad \text{Probabilità di } k \text{ arrivi in (0-t)}$$

Avendo ipotizzato che $P_0(t)=1$, ossia si considerino zero arrivi al tempo zero, $P_k(t)$ rappresenta la probabilità che ci siano un numero k di arrivi nell'intervallo di tempo che va da 0 a t .



Processi di Poisson





Processi di Poisson

$$E[k] = \sum_{k=0}^{\infty} k \frac{(\lambda t)^k}{k!} e^{-\lambda t} = e^{-\lambda t} \sum_{k=0}^{\infty} k \frac{(\lambda t)^k}{k!} = e^{-\lambda t} \sum_{k=1}^{\infty} \frac{(\lambda t)^k}{(k-1)!} =$$

$$e^{-\lambda t} \lambda t \sum_{k=1}^{\infty} \frac{(\lambda t)^{k-1}}{(k-1)!} = e^{-\lambda t} \lambda t \sum_{j=0}^{\infty} \frac{(\lambda t)^j}{j!} = \lambda t$$

$$\begin{aligned}\sigma_k^2 &= E[(k - E[k])^2] = E[k^2 - 2kE[k] + E[k]^2] = * \\ &= E[k^2] - \cancel{k} - 2kE[k] + E[k]^2 = E[k(k-1)] + E[k] - (E[k])^2 \\ E[k(k-1)] &= \sum_{k=0}^{\infty} k(k-1) \frac{(\lambda t)^k}{k!} e^{-\lambda t} = e^{-\lambda t} (\lambda t)^2 \sum_{k=2}^{\infty} \frac{(\lambda t)^{k-2}}{(k-2)!} = (\lambda t)^2 \\ \sigma_k^2 &= (\lambda t)^2 + \lambda t - (\lambda t)^2 = \lambda t\end{aligned}$$

Valor medio = varianza



$$\begin{aligned}&= E[k^2] - E[2KE[k]] + E[E[k]^2] = \\&= E[k^2] - 2E[k]E[k] + E[k]^2 = \quad \text{SIN LC} \quad E[k] = \text{const} \\&= E[k^2] - 2\underbrace{E[k]}_{\text{const}}^2 + \underbrace{E[k]}_{\text{const}}^2 = \\&= E[k^2] - E[k]^2 = E[k - K + K] - E[k]^2 = \\&= E[K(K-1)] + E[k] - \underbrace{E[k]}_{\text{const}}^2\end{aligned}$$



Processi di Poisson

statistica dei tempi di interarrivo

$$P(x(s, s+t) = k) = P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad k \geq 0, \quad t \geq 0, \quad \forall s$$

$$A(t) = p(\tilde{t} \leq t) = 1 - p(\tilde{t} > t) = 1 - P_0(t)$$

$$A(t) = 1 - e^{-\lambda t} \quad t \geq 0$$

$$a(t) = \lambda e^{-\lambda t} \quad t \geq 0$$

$$E(t) = \frac{1}{\lambda}; \quad E(t^2) = \frac{2}{\lambda^2}; \quad \sigma_t^2 = \frac{1}{\lambda^2}$$

$$A(s) = \int_0^{\infty} \lambda e^{-\lambda t} e^{-st} dt = \frac{\lambda}{s + \lambda}$$

Proprietà “memoryless” della distribuzione esponenziale

se il tempo di permanenza in uno stato è una v.a. esponenziale, il tempo trascorso nello stato non è utilizzabile per predire quanto si resterà ancora nello stato stesso.

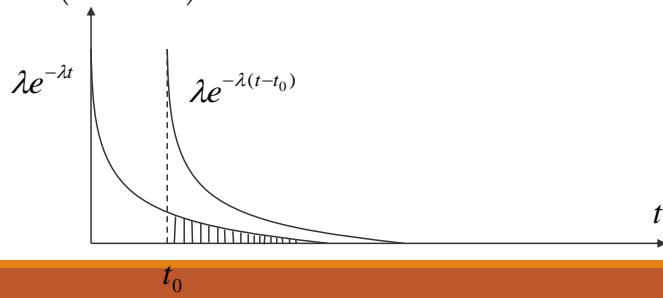


Processi di Poisson

Proprietà “**memoryless**” della distribuzione esponenziale

Supponiamo di fissare un istante $t=0$ di riferimento in corrispondenza di un arrivo. Se al tempo t_0 non vi è stato nessun altro arrivo, ci chiediamo quale sia la probabilità che il prossimo arrivo si verifichi dopo t a partire da t_0 .

$$\begin{aligned} P[\tilde{t} \leq t + t_0 | \tilde{t} > t_0] &= \frac{P[t_0 < \tilde{t} \leq t + t_0]}{P[\tilde{t} > t_0]} = \frac{P[\tilde{t} \leq t + t_0] - P[\tilde{t} \leq t_0]}{P[\tilde{t} > t_0]} = \\ &= \frac{A(t + t_0) - A(t_0)}{1 - A(t_0)} = \frac{1 - e^{-\lambda(t+t_0)} - (1 - e^{-\lambda(t_0)})}{1 - (1 - e^{-\lambda(t_0)})} = \frac{-e^{-\lambda(t+t_0)} + e^{-\lambda(t_0)}}{e^{-\lambda(t_0)}} = \\ &= 1 - e^{-\lambda(t)} = A(t) \end{aligned}$$



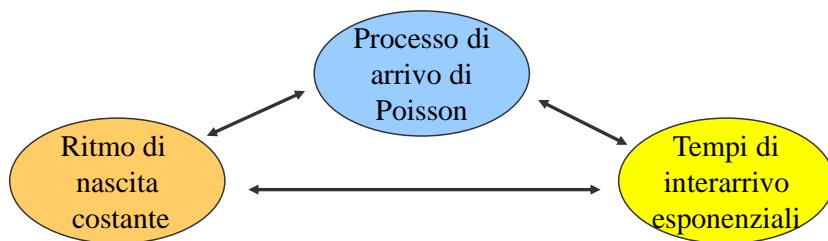


Processi di Poisson

Caratteristiche: catena di Markov di pura nascita a ritmo costante (λ)

- $\lambda_k = \lambda$ $k=0,1,2,\dots$
- $\mu_k = 0$
- $E[K] = \lambda t$, $\sigma_K^2 = \lambda t$

$$P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad k \geq 0, t \geq 0$$





Processi di Poisson



Ci siano k arrivi in [0-t] (evento condizionante B_k). Suddividiamo l'intervallo 0-t in intervalli generici di tipo α_i , senza arrivo, e in intervalli di tipo β_i , con un singolo arrivo (evento A_k)

$$P_1(\beta_i) = \lambda \beta_i e^{-\lambda \beta_i}$$

$$P_0(\alpha_i) = e^{-\lambda \alpha_i}$$

$$\begin{aligned} P(A_k | B_k) &= \frac{e^{-\lambda \alpha_1} \lambda \beta_1 e^{-\lambda \beta_1} e^{-\lambda \alpha_2} \lambda \beta_2 e^{-\lambda \beta_2} \dots e^{-\lambda \alpha_k} \lambda \beta_k e^{-\lambda \beta_k} e^{-\lambda \alpha_{k+1}}}{\frac{(\lambda t)^k}{k!} e^{-\lambda t}} = \\ &= \frac{\lambda^k (\beta_1 \beta_2 \dots \beta_k) e^{-\lambda t}}{\frac{(\lambda t)^k}{k!} e^{-\lambda t}} = \frac{(\beta_1 \beta_2 \dots \beta_k) k!}{t^k} \end{aligned}$$



Processi di Poisson

A questo punti si immagini di distribuire aleatoriamente k punti nell'intervallo 0-t con distribuzione uniforme. E' semplice ricavare che

$$P(A_k | B_k) = \left(\frac{\beta_1}{t} \frac{\beta_2}{t} \frac{\beta_3}{t} \dots \frac{\beta_k}{t} \right) k!$$

Siccome le due probabilità coincidono, ne consegue che:

dati k arrivi in 0-t, se questi sono generati da un processo di Poisson, allora questi sono distribuiti uniformemente nell'intervallo 0-t.

Super NB.



Processi di Poisson

Si supponga di accumulare k arrivi di un processo di Poisson. A tal fine sarà necessario un tempo pari alla somma di k v.a. esponenziali, la cui funzione caratteristica sarà pari a:

$$\Pi(s) = \left(\frac{\lambda}{s + \lambda} \right)^k$$

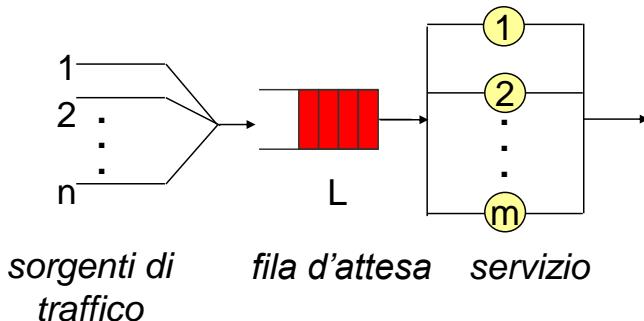
la cui anti-trasformata è pari a

$$f_X(x) \frac{\lambda(\lambda x)^{k-1}}{(k-1)!} e^{-\lambda x} \quad x \geq 0$$

DISTRIBUZIONE DI ERLANG (k)



Sistemi di servizio



- Il sistema è descritto attraverso variabili aleatorie:
- k = numero di utenti nel sistema
- l = numero di utenti nella sola fila d'attesa
- h = numero di serventi contemporaneamente occupati
- x = tempo di servizio
- s = tempo di permanenza nel sistema (tempo di coda o di ritardo)
- w = tempo di permanenza nella fila d'attesa



Sistemi di servizio

La variabile aleatoria k è caratterizzata attraverso la sua probabilità limite

$\pi_k = p_k$ =probabilità che in un generico istante di osservazione **in regime permanente** siano presenti k utenti (richieste di servizio) all'interno del sistema



Parametri prestazionali

- Probabilità di sistema bloccato (m serventi)

$$S_p = \Pr\{k = L + m\} = p_{L+m}$$

- Probabilità di rifiuto
 - Data una richiesta di servizio offerto (r.s.o.)

$$\Pi_p = \Pr\{\text{sistema bloccato/r.s.o.}\} = S_p \frac{\Pr\{\text{r.s.o./sistema bloccato}\}}{\Pr\{\text{r.s.o.}\}} = S_p$$

$$P(A|B) = \frac{P(A, B)}{P(B)} = \frac{P(A)P(B|A)}{P(B)}$$



Parametri prestazionali

- Probabilità di servizio bloccato (m serventi)

$$S_r = \Pr\{k \geq m\}$$

- Probabilità di ritardo

- Data una richiesta di servizio accolta (r.s.a.)

$$\Pi_r = \Pr\{\text{servizio bloccato/r.s.a.}\} = S_r \frac{\Pr\{\text{r.s.a./servizio bloccato}\}}{\Pr\{\text{r.s.a.}\}} = S_r$$

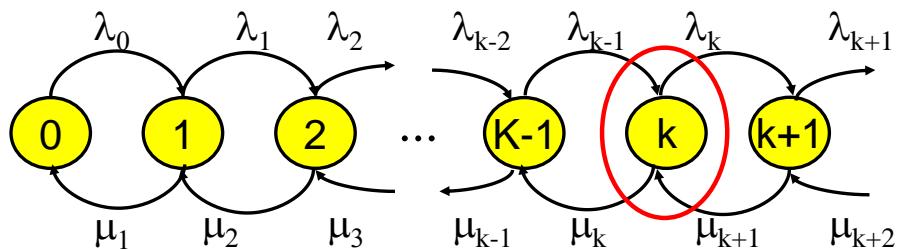
$$P(A|B) = \frac{P(A, B)}{P(B)} = \frac{P(A)P(B|A)}{P(B)}$$



Processi di nascita e morte in equilibrio

statistico

$$\begin{cases} \sum_{k=0}^{\infty} P_k = 1 \\ 0 = -(\lambda_k + \mu_k)P_k + \lambda_{k-1}P_{k-1} + \mu_{k+1}P_{k+1} \\ 0 = -\lambda_0P_0 + \mu_1P_1 \end{cases}$$





Processi di nascita e morte in equilibrio statistico

$$\mu_{k+1}P_{k+1} - \lambda_k P_k = \mu_k P_k - \lambda_{k-1} P_{k-1}$$

$$\text{sia } \alpha_k = \mu_k P_k - \lambda_{k-1} P_{k-1}$$

risulta

$$\alpha_k = \text{costante} = \mu_1 P_1 - \lambda_0 P_0 = 0, \quad \text{quindi}$$

$$P_k = \frac{\lambda_{k-1}}{\mu_k} P_{k-1} \quad \Rightarrow \quad P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}$$

siccome

$$P_0 + P_0 \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = 1 \quad \Rightarrow \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$



Sistema a coda M/M/1/ ∞/∞

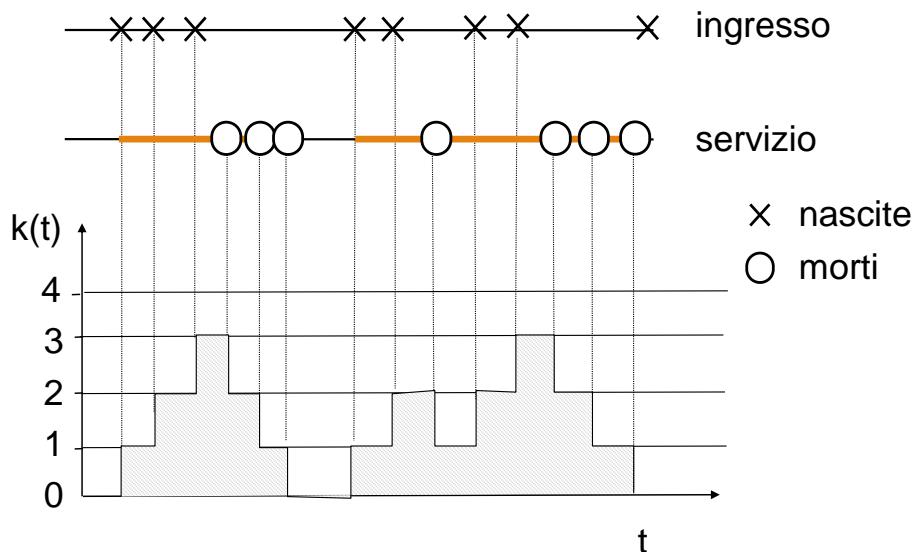
Ipotesi:

- tempi di interarrivo i.i.d. con distribuzione esponenziale negativa di parametro λ (ingresso di Poisson);
- tempi di servizio i.i.d. con distribuzione esponenziale negativa di parametro μ ;
- processi di arrivo e di servizio statisticamente indipendenti.
- singolo servente;
- spazio infinito per la fila di attesa.

Il processo di coda $K(t)$ è descrivibile mediante un processo di Markov di nascita e morte con spazio di stato $\{0,1,\dots\}$

Il processo di coda $K(t)$ è ergodico se $\lambda/\mu < 1$

Evoluzione temporale

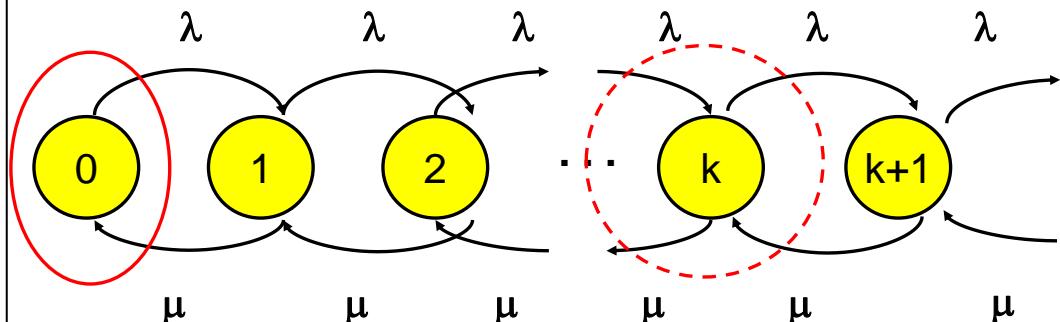




Frequenze di transizione di stato

$$\lambda_k = \lambda \quad \text{per } k \geq 0 \quad \text{frequenza di nascita}$$

$$\mu_k = \mu \quad \text{per } k \geq 1 \quad \text{frequenza di morte}$$





Probabilità limite di stato

Sostituendo λ e μ nella soluzione generale:

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \rho^k} = 1 - \rho$$

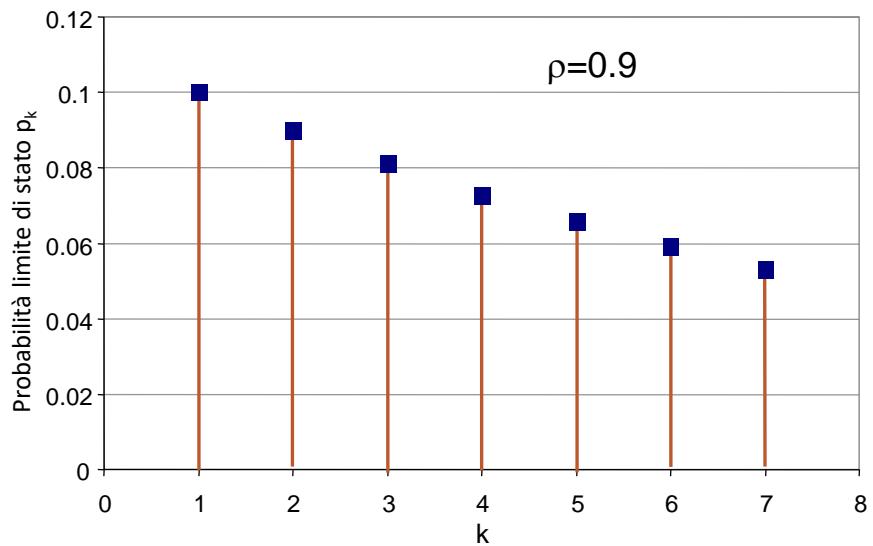
$$\text{dove } \rho = \lambda / \mu, \rho < 1$$

$$P_k = (1 - \rho) \rho^k$$

$k=0,1,2, \dots$ (distribuzione geometrica)



Probabilità limite di stato



La distribuzione è di tipo geometrico con parametro ρ



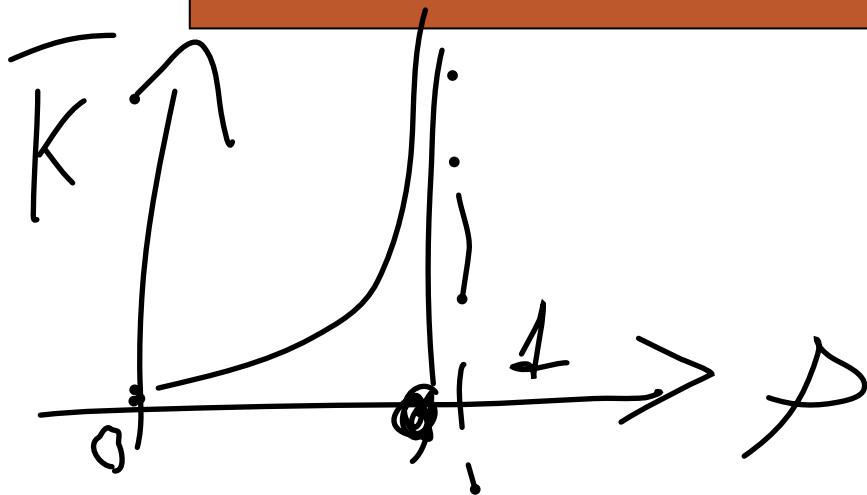
Probabilità limite di stato

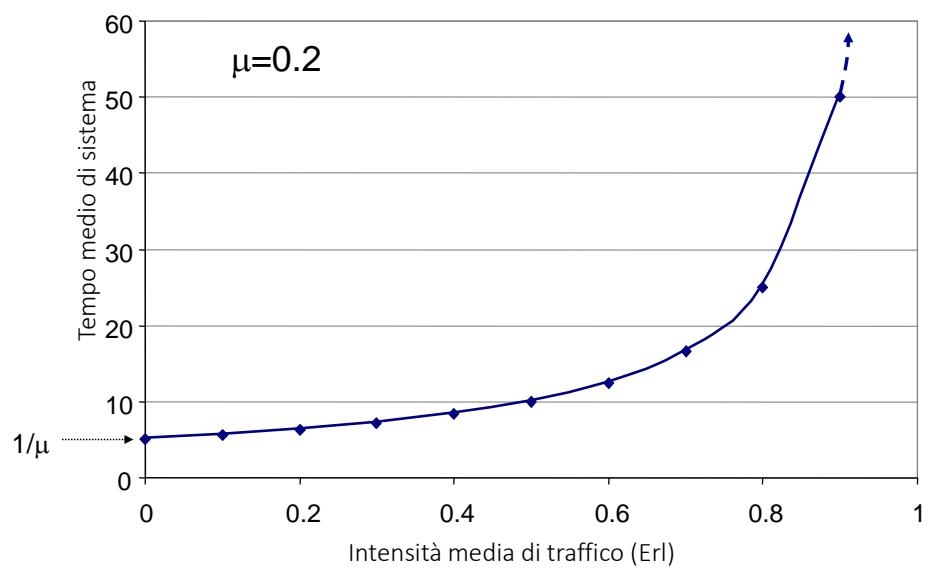
Il numero medio di utenti nel sistema è

$$\begin{aligned} E[K] = \bar{k} &= \sum_{k=0}^{\infty} k \cdot p_k = \sum_{k=0}^{\infty} k(1-\rho)\rho^k = (1-\rho) \sum_{k=0}^{\infty} k\rho^k = \\ &= (1-\rho)\rho \sum_{k=0}^{\infty} k\rho^{k-1} = (1-\rho)\rho \sum_{k=1}^{\infty} k\rho^{k-1} = (1-\rho)\rho \left(\frac{\partial}{\partial \rho} \sum_{k=0}^{\infty} \rho^k \right) = \\ &= (1-\rho)\rho \left(\frac{\partial}{\partial \rho} \frac{1}{1-\rho} \right) = \frac{\rho}{1-\rho} \end{aligned}$$

Il tempo di permanenza medio nel sistema è (Legge di Little)

$$T = \frac{\bar{k}}{\lambda} = \frac{\rho}{\lambda(1-\rho)} = \frac{1/\mu}{(1-\rho)} = \frac{1}{\mu - \lambda}$$





Al crescere dell'intensità di traffico il tempo di coda tende all'infinito



Parametri prestazionali

In condizioni di equilibrio statistico l'intensità media di richieste smaltite A_s coincide con l'intensità di richieste di servizio offerte A_o

$$A_s = A_o = \rho = 1 - p_0$$

La probabilità di servizio bloccato S_r coincide con la probabilità di ritardo nel ricevere servizio Π_r

$$S_r = \Pi_r = (1 - \rho) \sum_{k=1}^{\infty} \rho^k = \rho$$

ρ = prob. che il servente sia occupato = la percentuale temporale di occupazione del servente = la prob. che una richiesta in arrivo sia costretta ad attendere in coda



Distribuzioni in equilibrio statistico

l = lunghezza della fila d'attesa=numero di utenti nella fila d'attesa

$$Pr\{l = j\} = \begin{cases} (1-\rho) + \rho(1-\rho) = 1 - \rho^2 & j = 0 \\ (1-\rho) \cdot \rho^{j+1} & j \geq 1 \end{cases}$$
$$\bar{l} = \frac{\rho^2}{1-\rho}$$

h =numero di serventi impegnati

$$Pr\{h = j\} = \begin{cases} 1 - \rho & j = 0 \\ \rho & j = 1 \end{cases}$$
$$\bar{h} = \rho$$

il numero medio di utenti all'interno del sistema è quindi

$$\bar{k} = \bar{l} + \bar{h} = \frac{\rho}{1-\rho}$$



Tempi di attesa in coda

Si supponga che un utente trovi, al suo arrivo, il sistema nello stato k , ossia vi sono altri k utenti presenti nel sistema (uno in servizio e $k-1$ nella fila di attesa). Nel caso di disciplina **FIFO**, l'utente, prima di essere servito dovrà attendere un tempo pari alla somma di k v.a. esponenziali. Questo avverrà con probabilità $\rho^k(1-\rho)$. In media si avrà che:

$$\begin{aligned} W(s) &= \sum_{k=0}^{\infty} \left(\frac{\mu}{s+\mu} \right)^k \rho^k (1-\rho) = (1-\rho) \frac{1}{1 - \frac{\mu}{s+\mu} \rho} = \\ &= (1-\rho) \frac{s+\mu}{s+\mu - \mu\rho} = \frac{(1-\rho)(s+\mu + \lambda - \lambda)}{s+\mu(1-\rho)} = (1-\rho) + \frac{\lambda(1-\rho)}{s+\mu(1-\rho)} \end{aligned}$$

$$w(t) = (1-\rho)\delta(t) + \lambda(1-\rho)e^{-\mu(1-\rho)t}$$



Tempi di attesa in coda

$$F_w(t) = \Pr(w \leq t) = 1 - \rho \cdot e^{-(1-\rho)\mu \cdot t}$$

$$W = \frac{\rho}{\mu} \cdot \frac{1}{1-\rho}$$

Detto inoltre w_r l' **r-percentile** del **tempo di attesa** (cioè quel valore che non è superato per una percentuale di tempo uguale a $r\%$)

$$\Pr(w \leq w_r) = \frac{r}{100}$$

$$w_r = \frac{W}{\rho} \ln\left(\frac{100\rho}{100-r}\right)$$

Nel tuo appunto, il **percentile w_r** indica il tempo di attesa tale che la probabilità che il tempo di attesa sia **maggiore** di w_r sia pari a una determinata percentuale r . In altre parole, w_r è il tempo di attesa per cui la probabilità che il tempo di attesa sia **minore o uguale** a w_r sia $r\%$. Quindi:

- Se vogliamo sapere, per esempio, quanto tempo è necessario affinché l'80% delle persone attenda **meno di quel tempo** (e quindi il 20% attenda **più di quel tempo**), stiamo cercando l'**80^o percentile**.

Matematicamente, questo si traduce nel trovare il valore w_r tale che:

$$F_w(w_r) = r$$



Tempi di permanenza nel sistema

Nel caso di disciplina **FIFO**, l'utente, prima di essere servito dovrà attendere un tempo pari alla somma di $k+1$ v.a. esponenziali. Questo avverrà con probabilità $\rho^k(1-\rho)$. In media si avrà che:

$$S(s) = \sum_{k=0}^{\infty} \left(\frac{\mu}{s+\mu} \right)^{k+1} \rho^k (1-\rho) = \frac{\mu}{s+\mu} (1-\rho) \frac{1}{1 - \frac{\mu}{s+\mu} \rho} =$$

$$= \frac{\mu(1-\rho)}{s+\mu(1-\rho)} \quad \Rightarrow \quad s(t) = \mu(1-\rho) e^{-\mu(1-\rho)t}$$

$$F_s(t) = Pr(s \leq t) = 1 - e^{-(1-\rho)\mu t}$$

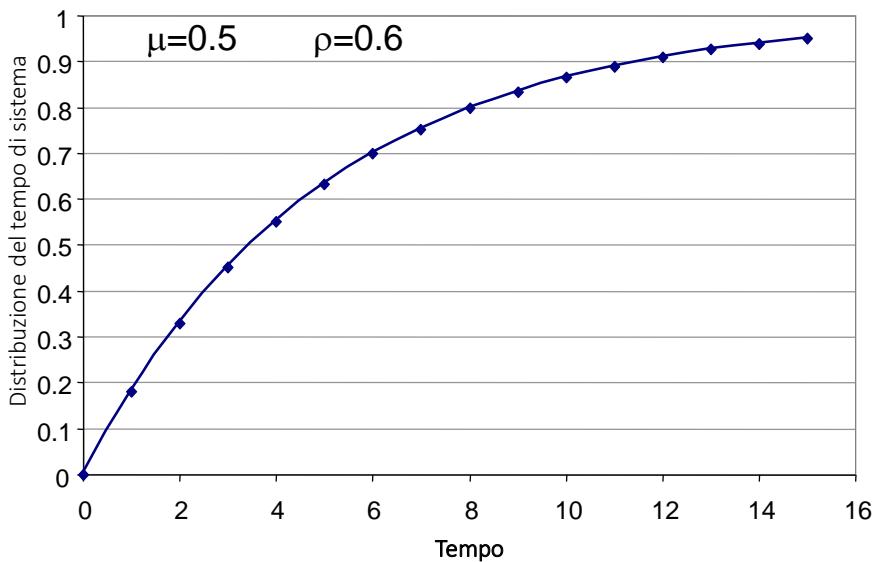
$$T = \frac{I}{\mu \cdot (1-\rho)} = \frac{I}{\mu - \lambda} = \bar{w} + \frac{I}{\mu}$$

detto inoltre s_r il percentile $r\%$ del tempo di coda

$$Pr(s \leq s_r) = \frac{r}{100}$$

$$s_r = T \frac{-\ln(1-r/100)}{1-\rho}$$

Tempi di permanenza nel sistema

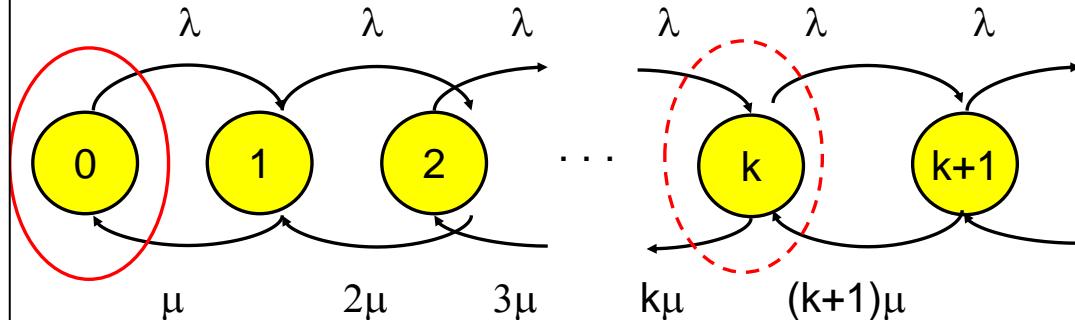




Sistema a coda M/M/ ∞

$$\lambda_k = \lambda \quad \text{per } k \geq 0 \quad \text{frequenza di nascita}$$

$$\mu_k = k\mu \quad \text{per } k \geq 0 \quad \text{frequenza di morte}$$



$$\frac{\lambda}{\mu} < \infty \quad P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$



Sistema a coda M/M/ ∞

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu}} = \frac{1}{1 + \sum_{k=1}^{\infty} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}} = e^{-\frac{\lambda}{\mu}}$$

$$P_k = e^{-\frac{\lambda}{\mu}} \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu} = \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} e^{-\frac{\lambda}{\mu}}, \quad k = 0, 1, 2, \dots$$

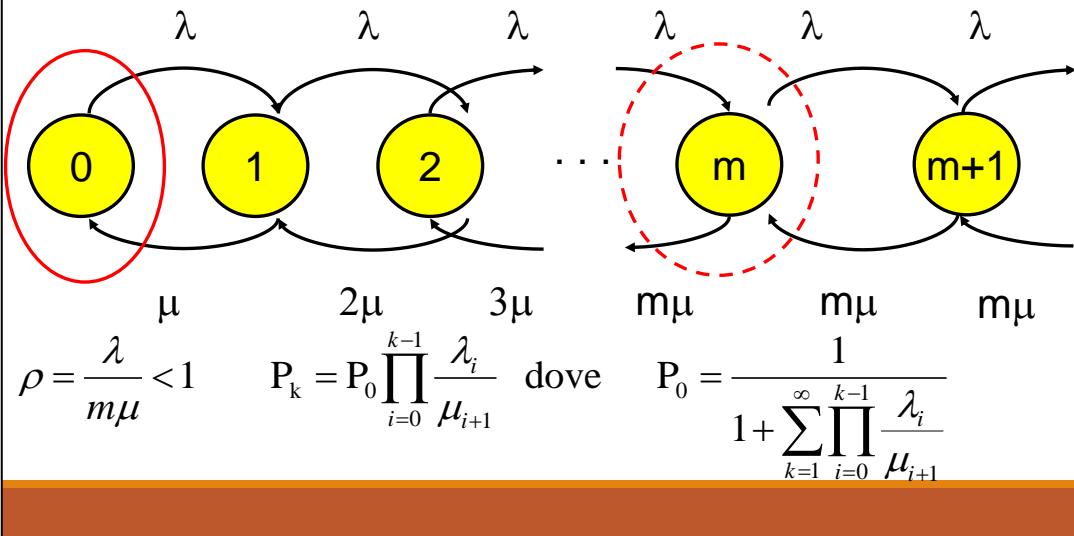
La distribuzione di probabilità degli stati è una distribuzione di Poisson valutata per $t=1/\mu$

$$\bar{N} = \frac{\lambda}{\mu} \quad T = \frac{1}{\mu}$$



Sistema a coda M/M/m

$$\lambda_k = \lambda \quad \mu_k = \min(k\mu, m\mu) = \begin{cases} k\mu, & 0 < k < m \\ m\mu, & m \leq k \end{cases}$$





Sistema a coda M/M/m

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu} = P_0 \left(\frac{\lambda}{\mu} \right)^k \frac{1}{k!} = P_0 \frac{(m\rho)^k}{k!}, \quad k \leq m$$

$$P_k = P_0 \prod_{i=0}^{m-1} \frac{\lambda}{(i+1)\mu} \prod_{i=m}^{k-1} \frac{\lambda}{m\mu} = P_0 \left(\frac{\lambda}{\mu} \right)^k \frac{1}{m! m^{k-m}} = P_0 \frac{\rho^k m^m}{m!}, \quad k \geq m.$$

$$\begin{aligned} P_0 &= \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}} = \frac{1}{1 + \sum_{k=1}^{m-1} \left(\frac{\lambda}{\mu} \right)^k \frac{1}{k!} + \sum_{k=m}^{\infty} \left(\frac{\lambda}{\mu} \right)^k \frac{1}{m! m^{k-m}}} = \\ &= \frac{1}{1 + \sum_{k=1}^{m-1} \frac{(m\rho)^k}{k!} + \sum_{k=m}^{\infty} \frac{(m\rho)^k}{m!} \frac{1}{m^{k-m}}} = \frac{1}{\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}} \end{aligned}$$



Sistema a coda M/M/m

Un utente che arriva in ingresso al sistema ha la necessità di accodarsi con probabilità pari a:

$$P[\text{coda}] = \sum_{k=m}^{\infty} P_k = \sum_{k=m}^{\infty} P_0 \frac{(m\rho)^k}{m!} \frac{1}{m^{k-m}} = P_0 \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}$$

$$P[\text{coda}] = \frac{\frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}{\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}$$

FORMULA DI ELRLAG C,
indicata come $C(m,\lambda/\mu)$

E' utilizzata per determinare la **probabilità di attesa** nell'accesso a una risorsa condivisa di m serventi disponibili. Ad esempio, può essere utilizzata nei call center per calcolare il numero di operatori necessari per gestire le chiamate entranti posto un certo livello di servizio.

Esempio

Si consideri un centralino telefonico operante ad attesa. Si assuma che:

- a un fascio di giunzioni all'uscita dell'autocommutatore sia offerto un traffico poissoniano entrante con intensità media di 25 Erl;
- tale fascio sia composto da 30 giunzioni;
- la durata di una conversazione telefonica sia distribuita con legge esponenziale negativa e con valore medio di 3 min.

Si determini la probabilità che una chiamata venga accodata

Modello M/M/m

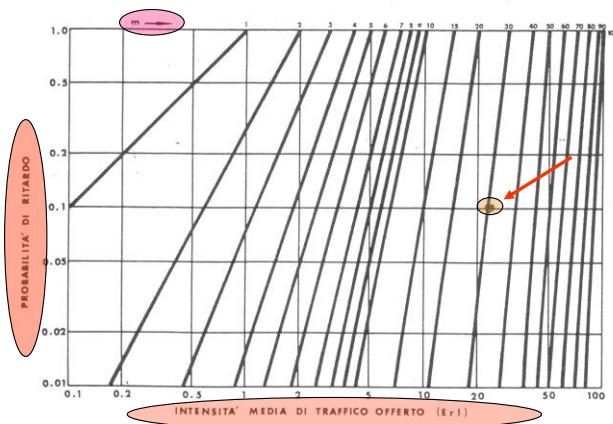
La probabilità di entrare in coda (cioè di subire un ritardo) è data da:

$$C\left(m, \frac{\lambda}{\mu}\right) = \frac{\frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}{\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}} \quad \text{C-ERLANG}$$

$$m = 30$$

Il traffico offerto A_0 è pari a 25 Erlang

$$\rho = \frac{\lambda}{m\mu} = \frac{25}{30} = 0.83 \quad \text{coefficiente di utilizzazione}$$



$$C(30,25) \approx 0.1$$



Sistema a coda M/M/m/m/ ∞

Ipotesi:

- tempi di interarrivo i.i.d. con distribuzione esponenziale negativa (λ);
- tempi di servizio i.i.d. con distribuzione esponenziale negativa (μ);
- processi di arrivo e di servizio statisticamente indipendenti.
- m serventi, statisticamente identici ed indipendenti;
- capacità nulla della fila d'attesa.

Il processo di coda è descrivibile mediante un processo di Markov di nascita e morte con spazio di stato $\{0, \dots, m\}$.

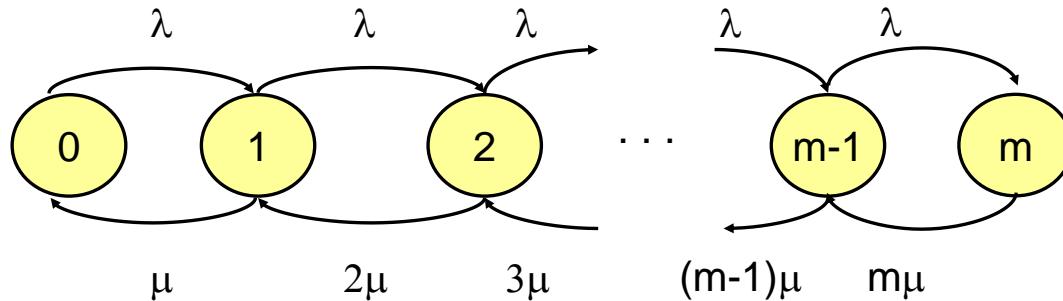
Il processo di coda è ergodico per ogni valore positivo di λ e μ (coda a perdita)



Frequenze di transizione di stato

$\lambda_k = \lambda$ per $0 \leq k \leq m-1$
frequenza di nascita

$\mu_k = k\mu$ per $1 \leq k \leq m$
frequenza di morte



$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$



Probabilità limite di stato

Per l'equilibrio dei flussi si ha (formule generali):

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu} \quad \text{dove} \quad P_0 = \frac{1}{\sum_{k=0}^m \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}}$$

posto $A_o = \lambda/\mu$: traffico offerto al sistema, risulta

$$P_k = P_0 \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} = P_0 A_o^k \frac{1}{k!} \quad 0 < k \leq m$$

$$P_0 = \frac{1}{\sum_{j=0}^m (A_o)^j \frac{1}{j!}}$$

$$P_k = \frac{A_o^k \frac{1}{k!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}}$$



Probabilità di blocco di servizio

Nel caso di processo di ingresso di Poisson, dato che la probabilità di r.s.o. è indipendente dallo stato, si ha:

$$\Pi_p = S_p \frac{\lambda_m}{\Lambda_o} = S_p$$

Nel caso di sistema a coda M/M/m/m (per k=m)

$$\Pi_p = S_p = \frac{A_o^m \frac{1}{m!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}}$$

FORMULA B
DI ERLANG

Super N.B. !



Formula B di Erlang

L'espressione della probabilità di sistema bloccato e di rifiuto per un sistema a coda M/M/m/m a perdita in senso stretto è denominata anche funzione di Erlang del 1° tipo di ordine m e di argomento A_o

Godere inoltre della proprietà di calcolo di tipo ricorsivo, infatti:

$$E_{I,m}(A_o) = \frac{A_o^m \frac{1}{m!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}} = \frac{A_o E_{I,m-1}(A_o)}{m + A_o E_{I,m-1}(A_o)}$$

- con il primo elemento pari a:

$$E_{I,1}(A_o) = \frac{A_o}{1 + A_o}$$



Formula B di Erlang

La grande importanza della formula B di Erlang risiede anche nel fatto che essa risulta valida per qualsiasi distribuzione dei tempi di servizio (resta necessaria l'ipotesi di i.i.d.).

In condizioni di equilibrio statistico la distribuzione del numero di utenti nel sistema è funzione del solo tempo medio di servizio $1/\mu$ e non della distribuzione del tempo di servizio stesso



Parametri prestazionali

Intensità media di traffico o «lavoro» smaltito A_s , che rappresenta il numero medio di serventi contemporaneamente occupati, dipende da A_o e dal numero di serventi m :

$$A_s = \sum_{k=1}^m kP_k = A_o [1 - E_{l,m}(A_o)]$$

Intensità media di traffico o «lavoro» rifiutato:

$$A_p = A_o - A_s = A_o E_{l,m}(A_o)$$

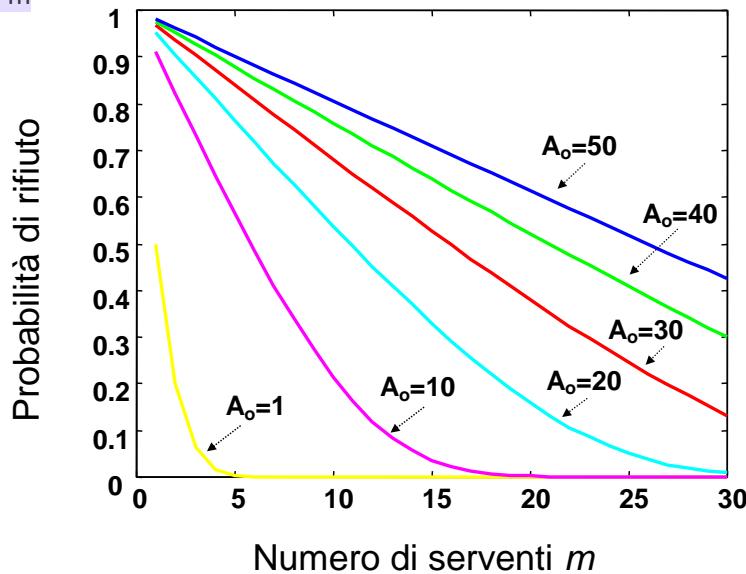
Coefficiente di utilizzazione del servente:

$$\rho = \frac{A_s}{m} = \frac{A_o}{m} [1 - E_{l,m}(A_o)]$$



Probabilità di rifiuto in funzione di m

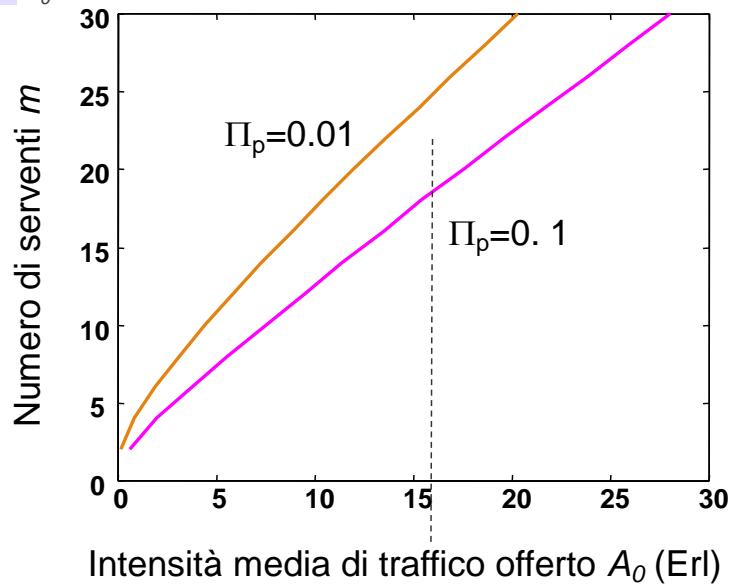
La probabilità di rifiuto, a parità di A_o , decresce al crescere del numero di serventi m





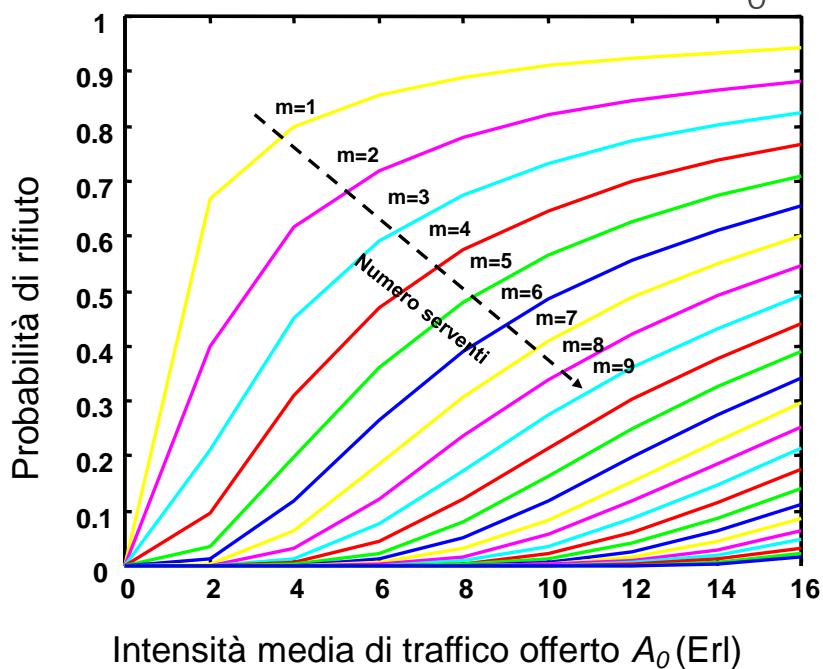
Dimensionamento di m in funzione di Π_p

La probabilità di rifiuto è, a parità di m , una funzione monotona crescente di A_0





Probabilità di rifiuto in funzione di A_0

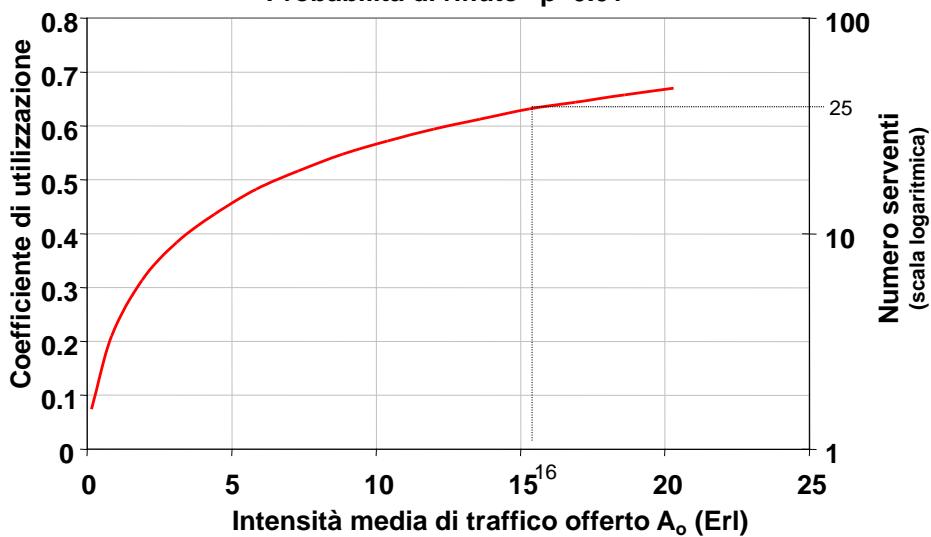




ρ in funzione di A_o

- A parità di congestione di chiamata, sistemi con elevato numero di serventi presentano, in condizioni di equilibrio statistico, un rendimento MIGLIORE rispetto a sistemi con pochi serventi.

Probabilità di rifiuto $P=0.01$





B di Erlang: dimensionamento del sistema

Dimensionamento del sistema: stimato il traffico offerto A_o e fissato il valore massimo per la probabilità di congestione di chiamata Π_{max} , determinare m:

- trovare il più piccolo valore di m tale per cui

$$E_{I,m}(A_o) \leq \Pi_{max}$$

- tale valore può essere facilmente determinato per tentativi a partire da $m=1$
- il valore effettivo della congestione di chiamata potrà risultare inferiore a Π_{max}

Esempio

- Intensità di richieste a un server DHCP $A_o=100$ Erl
- Tale traffico è offerto ad un unico server in modo tale che la probabilità di rifiuto sia minore dell'1%

$$E_{I,m}(A_0) \leq 0.01 \quad \rightarrow \quad m=117 \text{ indirizzi IP necessari}$$

- Si supponga di ripartire tali richieste uniformemente su n subnet, con $n=2, 4, 10, 25, 50, 100$
- Si può notare come all'aumentare di n aumenta il numero di indirizzi necessari e diminuisce il ρ di ogni singolo fascio

n	$A_{oi} = (A_o/n)$	m_i	$m=m_i * n$	Π_p	ρ
1	100	117	117	0.0098	0.8463
2	50	64	128	0.0084	0.7747
4	25	36	144	0.0080	0.6889
10	10	18	180	0.0071	0.5516
25	4	10	250	0.0053	0.3979
50	2	7	350	0.0034	0.2847
100	1	5	500	0.0031	0.1994



B di Erlang: valutazione delle prestazioni

Valutazione delle prestazioni: dato il numero dei serventi ed il traffico offerto, determinare la probabilità di congestione di chiamata:

- Occorre notare che solitamente è noto il traffico smaltito A_s^* e il numero di serventi m da cui si può stimare A_o attraverso la relazione seguente

$$A_o [1 - E_{I,m}(A_o)] = A_s^*$$

- Una volta calcolato A_o si calcola la probabilità di congestione di chiamata

$$\Pi_p = E_{I,m}(A_o)$$

Esempio (1/6)

- Si consideri un centralino telefonico automatico (PABX) di una grande azienda. Il centralino è collegato alla rete telefonica nazionale (RTN) tramite un certo numero di linee bidirezionali.
- Si consideri inoltre che:
 - nell'ora di punta gli utenti attestati al centralino formulano mediamente 140 chiamate dirette verso la RTN;
 - nell'ora di punta il numero di chiamate provenienti dalla RTN e dirette verso gli utenti del PABX è mediamente 180;
 - il flusso delle chiamate sia entranti sia uscenti è Poissoniano;
 - la distribuzione di probabilità delle durate delle conversazioni è di tipo esponenziale negativo con valor medio pari a 3 minuti;
 - la modularità delle linee è pari a 4, ovvero si possono inserire linee solo a gruppi di 4;
 - il PABX è del tipo a perdita pura.
- Si determini il numero di linee necessario a garantire un servizio con congestione di chiamata non superiore all'1%.
- Calcolare inoltre la frequenza massima delle chiamate consentita nell'ora di punta.

Per prima cosa calcoliamo Ao, aka traffico offerto

Esempio (2/6)

Il PABX può essere modellato con un sistema a coda del tipo $M/M/m/m$ in cui m è il numero di linee tra PABX e RTN

Si calcola il traffico globale offerto. Questo è pari alla somma del traffico uscente

$$A_u = \frac{140}{60} 3 = 7 \text{ Erl}$$

+

e del traffico entrante

$$A_e = \frac{180}{60} 3 = 9 \text{ Erl}$$

quindi

$$A_o = A_u + A_e = 16 \text{ Erl}$$

Esempio (3/6)

Per calcolare il numero di linee necessario a garantire una probabilità di congestione di chiamata minore dello 0.01 si deve determinare il minimo valore di m tale che

$$E_{I,m}(A_o) \leq 0.01$$

Si ottiene in tal caso $m=25$

A causa del vincolo sulla modularità il numero di linee da inserire sarà pari quindi a $m=28$

Dato tale numero di linee la congestione di chiamata sarà notevolmente inferiore a quella richiesta infatti

$$\Pi_{p,effettivo} = E_{I,28}(16) = 0.0019$$

E con queste 28 linee, alla fine QUANTE CHIAMATE possiamo supportare, pur RISPETTANDO il vincolo di probabilità di blocco < 0.01? --> aka riusiamo Erland, ma stavolta è Ao l'incognita, quello MAX!

Esempio (4/6)

Per determinare la frequenza massima delle chiamate consentita nell'ora di punta si calcola prima il valore di $A_{0,max}$ tale che

$$E_{1,28}(A_{0,max}) \leq 0.01$$

da cui si ricava $A_{0,max} = 18.64$

per cui

$$\lambda_{max} = A_{0,max} \frac{60}{3} \cong 373 \text{ chiamate/ ora}$$

Esempio(5/6)

Si consideri il PABX dimensionato con 28 linee bidirezionali che lo connettono alla Rete Telefonica Nazionale.

A distanza di tempo dalla sua installazione si vuole valutare la qualità di servizio offerta sapendo che a seguito di una campagna di misure si è riscontrato, nell'ora di punta, un valore di intensità media di traffico smaltito pari a circa 20.42 Erl.

Aka dobbiamo capire se persino nell'ora di punta si ha che
la prob di blocco è ancora RISPETTATA!

Esempio (6/6)

Dato il traffico smaltito misurato si può ricavare il traffico offerto al sistema risolvendo l'equazione

$$A_o (1 - E_{I,28}(A_o)) = 20.42$$

da cui si ha

$$A_o = 21 \text{ Erl} \rightarrow \text{e con questo traffico, come sono messa con la prob di blocco?}$$

Per quanto riguarda il valore di congestione di chiamata, si ha

$$E_{I,28}(21) = 0.0277 \rightarrow \text{oh shit è maggiore! Dovrò intervenire}$$

Il PABX non è più in grado di rispettare il vincolo sul grado di servizio. Le prestazioni sono variate, ad esempio, per un leggero incremento dell'utenza. Bisognerà quindi ridimensionare il numero di linee per riportare la probabilità di rifiuto sotto la soglia dello 0.01



Esempio (1/3)

Si considerino N terminali di utente che possiamo modellare come sorgenti di traffico dati. Ognuna emette traffico poissoniano con ritmo binario medio pari a λ bit/s e lunghezza dei pacchetti con distribuzione esponenziale negativa di media L .

Il traffico prodotto è inoltrato verso un load balancer (multiplatore) con capacità di smaltimento complessiva pari a C .

Si considerino due tecniche di multiplazione, con $N * \lambda < C$:

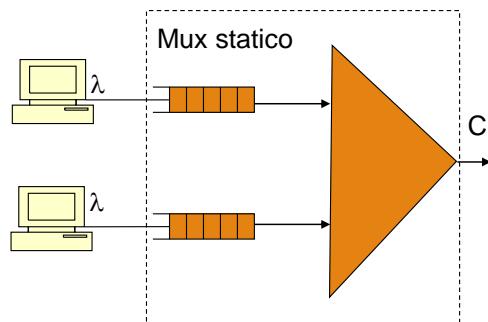
- 1) \circ Multiplazione statica: ad ogni utente è assegnato un buffer di dimensione infinita ed una capacità pari a C/N
- 2) \circ Multiplazione dinamica: tutta la capacità è dinamicamente condivisa tra tutte le sorgenti, che utilizzano un unico buffer di dimensione infinita

Si valuti e discuta la prestazioni delle due soluzioni in termini di coefficiente di utilizzazione della capacità C e del tempo medio di sistema T

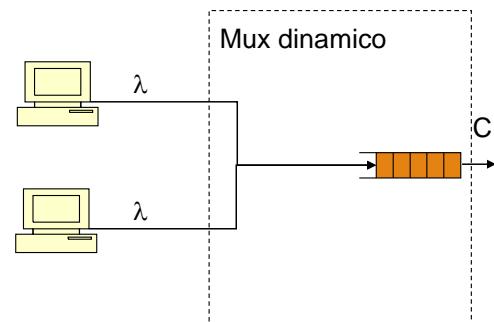


Esempio (2/3)

A



B





Esempio (3/3)

- Assunzioni: flussi statisticamente indipendenti
- Gestione delle code di tipo FIFO

A - N code M/M/1

coefficiente di utilizzazione:

$$\rho = \frac{\lambda}{C} = \frac{\lambda NL}{NL}$$

tempo medio di sistema: $T = \frac{\rho}{\lambda(1-\rho)}$

--> E' quello del sistema M/M/1, e io
se conosco rho lo conosco!
Lo avevamo visto l'altra volta!

qui affascio!

----> B - 1 coda M/M/1

coefficiente di utilizzazione:

$$\rho = \frac{N\lambda}{C} = \frac{\lambda NL}{L}$$

tempo medio di sistema: $T = \frac{\rho}{N\lambda(1-\rho)}$

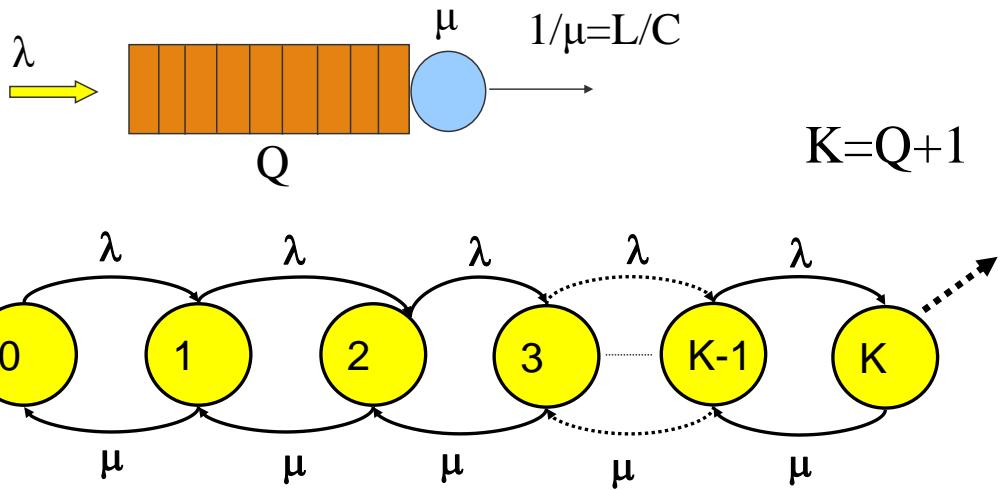
N . b

Il GUADAGNO nel MULTIPLARE:

- 1) Nei sistemi orientati ALLA PERDITA (no coda) --> se affasciamo il traffico otteniamo un COEFFICIENTE DI UTILIZZAZIONE PIU' ALTO per ogni singolo servente!
- 2) Nei sistemi orientati AL RITARDO --> se affasciamo il traffico otteniamo dei TEMPI DI PERMANENZA di CODA MOLTO RIDOTTO, ridotto di quel fattore N!



Sistema a coda M/M/1/K



$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$



Indicando con $A_0 = \lambda/\mu$ il traffico offerto, si ricavano le probabilità limite di stato

$$P_k = P_0 A_0^k \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{i=1}^K A_0^i} = \frac{1}{1 + \sum_{i=1}^K A_0^i} = \frac{1}{1 + \left(\sum_{i=1}^{\infty} A_0^i - \sum_{i=K+1}^{\infty} A_0^i \right)} = \frac{1 - A_0}{1 - A_0^{K+1}}$$
$$= \frac{1}{1 + \left(\frac{A_0}{1 - A_0} - \frac{A_0^{K+1}}{1 - A_0} \right)} = \frac{1 - A_0}{1 - A_0^{K+1}}$$

$$P_k = \begin{cases} \frac{1 - A_0}{1 - A_0^{K+1}} A_0^k & 0 \leq k \leq K \\ 0 & k > K \end{cases}$$

$$A_S = A_0 (1 - P_{rifuto})$$

Ricorda:

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$



Proprietà “PASTA”

Catena di Markov: stazionaria

- Il processo si è avviato dalla condizione statistica di stazionarietà, o
- Il processo dura per un tempo $t \rightarrow \infty$

► La probabilità che in un dato istante t il processo si trovi nello stato i è uguale alla probabilità stazionaria che

$$p_i = \lim_{t \rightarrow \infty} P\{N(t) = i\} = \lim_{t \rightarrow \infty} \frac{T_i(t)}{t}$$

T_i =tempo di permanenza del processo nello stato i

Domanda: Per un sistema a coda M/M/1, se t è il tempo di un arrivo, qual è la probabilità che $N(t)=i$?

► Risposta: Poisson Arrivals See Time Averages (PASTA).



Proprietà PASTA

Probabilità stazionarie:

$$p_n = \lim_{t \rightarrow \infty} P\{N(t) = n\}$$

Probabilità stazionarie **in corrispondenza di un arrivo**:

$$a_n = \lim_{t \rightarrow \infty} P\{N(\underline{t}) = n \mid \text{arrival at } t\}$$

Ipotesi LAA (Lack of Anticipation): I futuri tempi di interarrivo e i tempi di servizio dei clienti arrivati precedentemente sono indipendenti

► **Teorema:** In un sistema a coda che soddisfa l'ipotesi LAA:

1. Se il proceddo degli arrivi è di Poisson si ha che

$$a_n = p_n, \quad n = 0, 1, \dots$$

2. Quello di Poisson è il solo processo avente questa proprietà (condizione necessaria e sufficiente)



Proprietà PASTA

La proprietà PASTA si applica ad altri processi di arrivo?

Esempio:

Arrivi **deterministici** ogni 10 sec

Tempo di servizio **deterministico** di 9 sec

- ➡ In corrispondenza di ogni arrivo il sistema è vuoto $a_1=0$
- ➡ Il tempo medio in cui un solo utente è presente nel sistema $p_1=0.9$

Le medie osservate dall'utente **non sono necessariamente medie temporali del sistema**

L'aleatorietà non è di aiuto, a meno che sia generata da un processo di Poisson!



Proprietà PASTA: dimostrazione

Sia $A(t, t+\delta)$, l'evento in cui vi sia un arrivo in $[t, t+\delta]$

Se che un utente arriva in t , la probabilità $a_n(t)$ di trovare il sistema nello stato n è data da

$$P\{N(t^-) = n \mid \text{arrival at } t\} = \lim_{\delta \rightarrow 0} P\{N(t^-) = n \mid A(t, t+\delta)\}$$

$A(t, t+\delta)$ è indipendente dallo stato del sistema prima di t , $N(t^-)$

- $N(t^-)$ è determinato dai tempi di arrivo $< t$, e dai corrispondenti tempi di servizio
- $A(t, t+\delta)$ è indipendente dallo stato del sistema, ossia dagli arrivi $< t$ [Poisson] e dai tempi di servizio degli utenti arrivati $< t$ [LAA]

$$a_n(t) = \lim_{\delta \rightarrow 0} P\{N(t^-) = n \mid A(t, t+\delta)\} = \lim_{\delta \rightarrow 0} \frac{P\{N(t^-) = n, A(t, t+\delta)\}}{P\{A(t, t+\delta)\}}$$

$$= \lim_{\delta \rightarrow 0} \frac{P\{N(t^-) = n\} P\{A(t, t+\delta)\}}{P\{A(t, t+\delta)\}} = P\{N(t^-) = n\}$$

$$a_n = \lim_{t \rightarrow \infty} a_n(t) = \lim_{t \rightarrow \infty} P\{N(t^-) = n\} = p_n$$



Esempi

Esempio 1: Arrivi non di Poisson

Tempi di interarrivo IID distribuiti uniformemente fra 2 and 4 sec

Tempi di servizio deterministici di 1 sec

► In corrispondenza di ogni arrivo il sistema è vuoto, quindi $a_1 = 0$.

► $\lambda=1/3$, $T=1 \rightarrow N=T\lambda=1/3 \rightarrow p_0=2/3$, $p_1=1/3$

Esempio 2: mancanza dell'ipotesi LAA

Arrivi di Poisson

Tempo di servizio dell'utente i : $S_i = \alpha T_{i+1}$, $\alpha < 1$

► In corrispondenza di ogni arrivo il sistema è vuoto, $a_1 = 0$.

► Il tempo medio in cui un solo utente è presente nel sistema $p_1 = \alpha$



Distribuzione dopo la partenza

$$d_n = \lim_{t \rightarrow \infty} P\{X(t^+) = n \mid \text{departure at } t\}$$

Probabilità di stato stazionarie **dopo una partenza**:

Usando la stessa proprietà di Markov:

- I limiti di a_n e d_n esistono e coincidono
- ➔ $a_n = d_n, n = 0, 1, \dots$
- ➔ In condizioni stazionarie, il sistema appare stocasticamente identico agli utenti che arrivano e che lasciano il sistema.

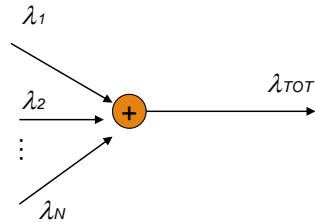
Arrivi di Poisson + LAA: un utente che arriva e un utente che lascia il sistema vedono il sistema stesso con la stessa statistica osservabile in un tempo aleatorio.



Ancora sui processi di Poisson...

L'aggregazione di N processi di Poisson indipendenti di parametro λ_i ,
 $i=1\dots N$, è un processo di Poisson di parametro

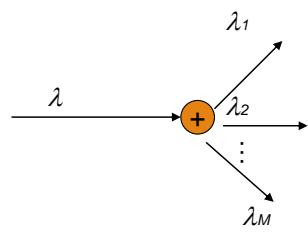
$$\lambda_{TOT} = \sum_i \lambda_i$$





Ancora sui processi di Poisson...

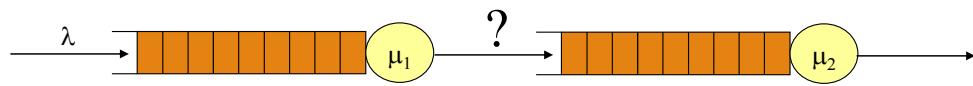
La separazione statistica di un processo di Poisson di parametro λ con probabilità p_1, p_2, \dots, p_M genera M processi di Poisson di parametro $\lambda_i = \lambda p_i$, $i=1,\dots,M$.





Teorema di Burke

Date due code in “tandem”, qual è la distribuzione dei tempi di interarrivo alla seconda coda, se la prima è una M/M/1?



Tale distribuzione sarà equivalente a quella dei tempi di interpartenza ($D(t)$) dalla coda 1



Teorema di Burke

Indichiamo con

- $D^*(s)$ la trasformata di Laplace di $D(t)$
- $B^*(s)$ la trasformata di Laplace di $B(t)$ (distribuzione dei tempi di servizio)

Quando un cliente parte da una coda (cioè ha ricevuto il suo servizio) può verificarsi uno solo dei seguenti eventi:

- un altro cliente è presente sulla coda 1 e sarà subito servito
- la coda 1 è vuota, quindi bisognerà attendere un tempo uguale alla somma di due contributi prima di un nuovo arrivo alla coda 2:
 - il tempo fino all'arrivo di un altro cliente
 - il suo tempo di servizio



Teorema di Burke

Nel primo caso risulta

- $D^*(s) \Big|_{\text{coda non vuota}} = B^*(s)$

Nel secondo caso, ho la somma di due variabili aleatorie indipendenti, perciò la variabile aleatoria somma sarà la convoluzione delle pdf, quindi, nel dominio di Laplace, il prodotto delle trasformate delle distribuzioni

- $D^*(s) \Big|_{\text{coda vuota}} = B^*(s) \lambda/(s+\lambda)$

Poiché il servente è di tipo esponenziale, si ha inoltre che

- $B^*(s) = \mu_I/(s+\mu_I)$



Teorema di Burke

Poiché la probabilità di avere il sistema non vuoto è pari $\rho = \lambda/\mu$, si ottiene che

- $D^*(s) = (1-\rho) D^*(s)|_{\text{coda vuota}} + \rho D^*(s)|_{\text{coda non vuota}}$

Sostituendo i valori precedenti si ottiene

- $D^*(s) = (1-\rho) (\lambda/(s+\lambda)) (\mu_1/(s+\mu_1)) + \rho (\mu_1/(s+\mu_1))$

che risulta uguale a

- $D^*(s) = (\lambda/(s+\lambda)) = A^*(s) \Rightarrow D(t) = A(t) = 1 - e^{-\lambda t} \text{ per } t \geq 0$

Quindi i tempi di interpartenza sono distribuiti esponenzialmente con lo stesso parametro dei tempi di interarrivo:

- **la coda 2 può essere trattata come una M/M/1 indipendente dalla coda 1 !**

Burke estende questo risultato alle code M/M/m



Teorema di Burke

Considerando le operazioni viste sui processi di Poisson, il Teorema di Burke implica che se si connettono dei server (cioè sistemi a coda) in modalità **feed-forward** allora ogni nodo della rete di code può essere esaminato SINGOLARMENTE come se fosse un sistema a coda indipendente.

Questo risultato è generalizzato dal Teorema di Jackson per le reti di code aperte



Teorema di Jackson

Si consideri una rete di code formata da K nodi (sistemi a coda) che soddisfano le seguenti tre condizioni:

- Ogni nodo contenga c_k serventi aventi tempo di servizio distribuiti esponenzialmente con parametro μ_k .
- Gli utenti provenienti dall'esterno giungono al generico nodo k secondo un processo di Poisson con parametro λ_k .
- Quando un utente è servito al nodo k è trasferito “istantaneamente” al generico nodo j con probabilità p_{kj} oppure esce dalla rete con probabilità $1 - \sum_j p_{kj}$

Teorema di Jackson

Il tasso degli arrivi al generico nodo k sarà:

$$\Lambda_k = \lambda_k + \sum_{j=1}^K p_{jk} \Lambda_j$$

Indicando con $p(n_1, \dots, n_K)$ la probabilità congiunta stazionaria di stato nei nodi, se

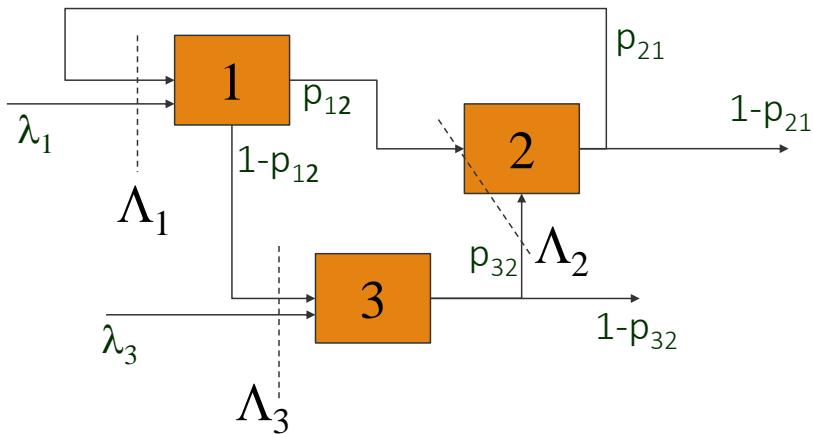
$$\Lambda_k < c_k \mu_k \quad \forall k,$$

$$\text{Allora } p(n_1, \dots, n_K) = p_1(n_1) p_2(n_2) \dots p_K(n_K)$$

dove $p_i(n_i)$, $i=1, \dots, K$, è la probabilità stazionaria che il nodo i si trovi nello stato n_i , (vi siano n_i utenti nel sistema a coda i), modellandolo come un sistema M/M/c_i con processo degli arrivi di Poisson con parametro Λ_i e tasso di servizio μ_i



Teorema di Jackson



$$\begin{cases} \Lambda_1 = \lambda_1 + p_{21}\Lambda_2 \\ \Lambda_2 = p_{12}\Lambda_1 + p_{32}\Lambda_3 \\ \Lambda_3 = \lambda_3 + (1-p_{12})\Lambda_1 \end{cases} \rightarrow \Lambda_1, \Lambda_2, \Lambda_3$$



Ipotesi di indipendenza di Kleinrock

- Tempi di interarrivo indipendenti alle varie code della rete
- Tempo di servizio di un generico pacchetto nelle varie code indipendente.
 - La lunghezza del pacchetto è random ogni volta che un pacchetto è trasmesso attraverso un collegamento.
- Tempo di servizio e tempi di interarrivo statisticamente indipendenti.

Le assunzioni sono state validate attraverso risultati sperimentali e simulazioni. In tal caso si ha che:

La distribuzione stazionaria degli stati approssima quella descritta dal teorema di Jackson

L'approssimazione è accettabile quando:

- Il processo degli arrivi dai punti in ingresso alla rete è un processo di Poisson.
- Il tempo di trasmissione del pacchetto è una variabile aleatoria approssimabile mediante un'esponenziale.
- Molto flussi di pacchetti sono multiplati in ogni collegamento.
- La rete è densamente connessa
- Vale per un'intensità di traffico da moderata a pesante.

L'indipendenza in questione riguarda i tempi di servizio e i tempi di arrivo.