Cloud Computing - Notes

Eduardo Rinaldi - rinaldi.1797800@studenti.uniroma1.it

June 2022

Contents

1 The big picture of Cloud Computing

Cloud computing is the product of a set of three main technologies:

1. **Virtualization:** capacity to abstract elements of computing (e.g hardware, storage, etc.). *Hardware virtualization* consists in abstracting hardware resources using software (e.g. vpn or virtual machines). *Software virtualization* or, also known as *application virtualization*, consists in executing an application in an isolated environment (e.g. using *docker*).

2. **Web 2.0:** set of technologies that allows to create web applications and not only static/dynamic web pages.

3. **Service orientation:** a reference model for cloud computing that base everything on a small building block called "*service*". A service can be seen as a small piece of code that solve a simple task and it can be reused in different kind of applications (e.g. a "login/register" service). So given this, a service should be reusable, programming language and location independent.

1.1 Business drivers and risks

Most important cloud computing's business drivers are three:

1. **Capacity planning:** provide the right amount of capacity when needed, acting different strategies: lag strategy (reactive), lead strategy and match strategy (proactive).

2. **Cost reduction:** cloud solutions dramatically reduce costs needed to buy and maintain a certain infrastructure. A consumer does not have to worry about maintenance costs (provider responsibility) and also some of the costs are reduced by the fact that most services are payed on-demand (you pay for what you actually use).

3. **Agility:** measures the responsiveness to the change, in other words cloud solutions allows to response to an higher/lower demand within few hours.

Even if we have several benefits, this kind of technology brings some risks:

1. Increase security vulnerabilities

2. Limited portability between cloud providers

3. Reduced governance control

1.2 NIST definition of cloud computing

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

This cloud model is based on **5 essential characteristics**, **3 service models** and **4 deployment models**:

- Characteristics:
  1. **On-demand self-service:** relies on orchestration services, programming API, dedicated shell, web interface.
  2. **Broad network access:** accessible from anywhere and from any device that can connect to internet.
  3. **Resource pooling:** resources are pooled to serve multiple consumer using a multi-tenant model
  4. **Rapid elasticity:** is related to *scalability*, is the ability of a system of adapting to workload changes in an automatic manner.
  5. **Measured services:** user should monitor resources usage so to apply automatic management tasks (e.g. scaling using thresholds, control cost by using alerts, etc.)
- Service models: they help to separate consumer and provider responsibility based on the service type
  1. **Infrastructure as a service (IaaS):** provider is responsible for the infrastructure (i.e. server, storage, virtual machine), everything else is consumer responsibility.
  2. **Platform as a service (PaaS):** provider is responsibile for the platform (i.e. infrastructure and everything else except for application and data).
  3. **Software as a service (SaaS):** provider is responsible for every resource given to the consumer, both software and hardware.
- Deployment models:
  1. **Public cloud:** cloud resources are publicly accessible.
  2. **Private cloud:** cloud resources are private, they act as they are in a virtual private network.
  3. **Hybrid cloud:** public + private cloud.
  4. **Community cloud:** cloud infrastructure can be used by consumers of a specific community with a common need.

1.3 Cloud computing roles

- **Provider:** for example Amazon AWS, Google Cloud Platform

- **Consumer:** the one who will access to cloud resources (not the application end-user)
- **Auditor:** conduct assessment of cloud services testing performance and security
- **Broker:** third part that helps the negotiation between consumer and provider (helps not in the sector consumers to choose the right services)
- **Carrier:** provide connectivity and transport of cloud services.
- **Resource administrator:** responsible for administrating cloud resources (could be the consumer, the provider or a thirdy part organization)
- **Service owner:** the legal owner of a cloud service (could be the consumer or the provider)

2 Enabling core technologies

2.1 Distributed computing

It's a computing model where computation is broken down into multiple units executed concurrently (i.e. on *multi-node* or *multi-cpu* or *multi-core*), implying different location of computing elements and heterogeneity in terms of hardware and software features. Usually these systems includes:

- Framework for distributed programming (e.g. spark)
- Inter-process communication (**IPC**) primitives for control and data
- Parallel hardware and networking

To give two formal definitions of distributed systems:

1. *"A distributed system is a collection of independent computers that appears to its users as a single coherent system"* [Tanenbaum]
2. *"A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages"* [Coulouris]

2.2 Software architectural styles

It defines the logical organization of the software components. Two main styles (not the only existing ones) are:

1. **Call and return:** can be based on a classic top-down approach where each component is either a procedure or a subprogram and the calling program pass inputs to procedures which these return an output value when the computation is done; this approach can also follow a layered structure (similar to tcp/ip stack).

2. **Independent components:** each component has its own life-cycle and they interact each other. Communication between components is defined by their architecture which can be either "process based" (on slide "communicating processes"), i.e. each service is an independent process and each process provides services that can be used by others, or "event based", i.e. each component publish a list of events which other components can subscribe and define their callback for that event.

One architectural style which is based on independent components is "**Microservices**", where basically we have 10s or 100s of small independent services each of which is handled by a small team. These kind of architecture allows us to:

- continuously deploy large applications (avoiding to deploy every time the entire application),
- easily maintain and scale services, i.e. *x-axis* (**cloning**), *y-axis* (**functional decomposition**), *z-axis* (**data partitioning**) scaling,
- test new technologies; each service is independent, this also means that they're vendor independent,
- isolate faults.

Drawbacks of microservices are given by the amount of complexity they introduce (e.g. decompose a system into services, development and testing).

2.3 System architectural styles

It defines the physical organization of processes and components over a distributed system. Two styles:

1. **Client-server:** composed by three main components which are: presentation, logic and data storage. Based on how these three components are arranged we can have *thin-client* (i.e. logic is performed on server) and *fat-client* (i.e. logic is performed on client). A distributed approach is given by multi-tier model which consists in making components conceptual layers called tiers. (classic model is a 2-tier approach, because we only have one client and one server)
2. **Peer-to-Peer:** each component (peer) can be both client and server at the same type, consistency here is maintained by consensus algorithms (like paxos or the ones used in blockchain)

2.4 IPC models

Communication between processes in distributed systems can be achieved by different IPC models:

- **Remote Procedure Call (RPC):** use a client/server approach, where the client call for the remote procedure and the server process and return the procedure result. The RPC infrastructure is responsible for marshaling/un-marshaling and for handling requests between client and server. Developers should only worry about defining and registering the remote procedure on the server and define the client code that invokes the remote procedures.
- **Distributed objects:** very similar to RPC but for object oriented paradigm (stateful approach).
- **Service Oriented Architecture (SOA):** a service encapsulate a software component where boundaries are explicit (simple interfaces), service is autonomous, services share schemas and contracts (those defines the structure of the messages that can be sent or received) and their compatibility is defined by policies. With this approach services can be both consumer and provider and they can belong to different domains or organizations. Services are then managed in two ways:
    1. **Orchestration:** there's an entity which manage the communication between services
    2. **Choreography:** there's no central entity which manage communication (this avoid single point of failure), but instead services communicates each other directly.

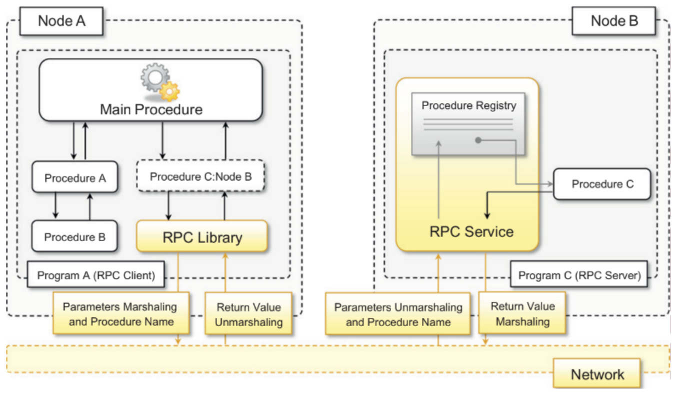


Figure 1: RPC

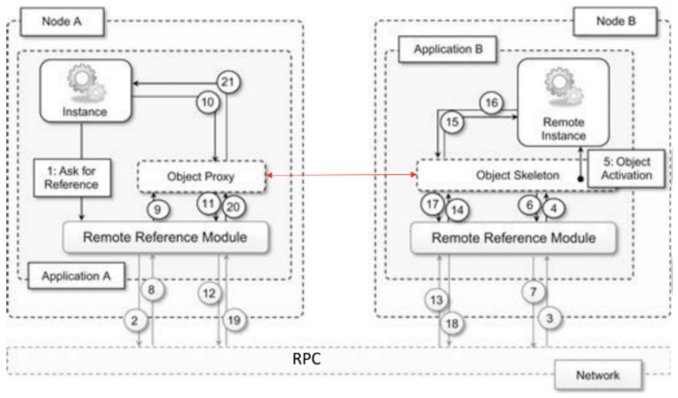


Figure 2: Distributed objects

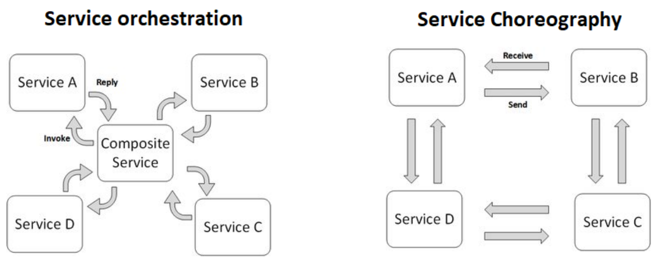


Figure 3: SOA service managing

2.5 Differences between SOA and Micro-services

Even if at a high level they seems to be the same things we can spot some differences in:

- **Communication mechanism:** SOA uses an Enterprise Service Bus as a communication middle-ware, creating a single point of failure, providing a slow communication with an high configuration and maintenance complexity. Micro-services instead use lightweight and open-source technologies like REST.
- **Data:** SOA usually have shared database, while each micro-service has its own database.
- **Services' size:** usually SOA is used for integrating monolithic applications, so that they can communicate each other, micro-services instead are very small pieces of software (a single application could easily be composed by hundreds of services).

3 Virtualization

Virtualization is a technique that by using software simulates hardware in order to create a virtual system. This technique has been widely explored since it allows us to better deal with: unused hardware and software resources, lack of spaces, administrative costs (similar to cloud computing's drivers). In addition to these, we can say that virtualization:

- **Increase security:** provides an isolation layer between guest os and host os (in terms of operations and data).
- **Execution is managed:** allows for an array of possible computing scenarios (i.e. guest programs and environments can be shared, aggregated, emulated or isolated).
- **Portability:** a virtual machine image can be easily duplicated and instantiated on another host machine. (e.g. java code can be executed on any os running a jvm).

Virtualization techniques can be split in to two macro-categories: *Hardware* (or *System*) *level virtualization* and *Software level virtualization*.

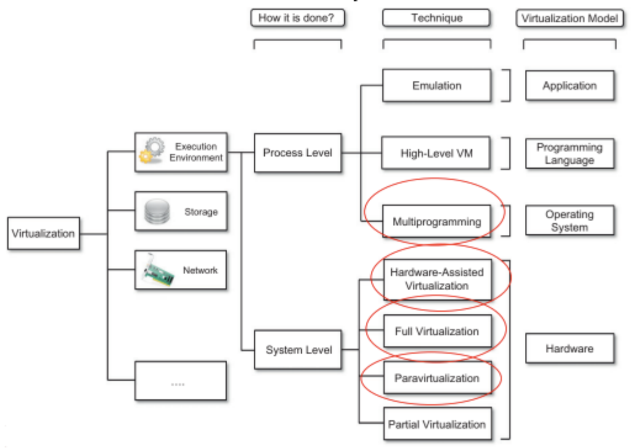


Figure 4: Virtualization techniques

3.1 Hardware / System level virtualization

This kind of a approach consists in using an **Hypervisor** (a.k.a. **Virtual Machine Manager - VMM**) so to emulate and manage CPU status on guest OS. An Hypervisor is a further abstraction layer that runs in supervisor mode (i.e. privileged mode) and it's placed between a virtual machine instance and, the operating system (*type II*) or directly the hardware (*type I*). Generally, an hypervisor is composed by:

- **Dispatcher:** entry point which dispatch instructions to either allocator or the interpreter
- **Allocator:** manage system resources available to VM. If an instruction change VM's current resource allocation, the allocator will be invoked.
- **Interpreter:** every time the virtual machine want to execute a privileged instruction a trap is triggered and the corresponding routine is executed.

An hypervisor, to be defined as so, should respect 3 properties:

1. **Equivalence:** guest should behave like if it's running on physical host.
2. **Resource control:** complete control over virtualized resources.
3. **Efficiency:** a good fraction of instructions should be executed without the intervention of VMM.

3.1.1 Full virtualization

This approach consists in scanning the instruction streams and, non-privileged instruction are directly run on hardware, while privileged instructions are trapped and VMM tries to emulate their behavior by using **binary translation**.

Binary translation is an emulation technique where instructions of source instruction set are converted into a target instruction set; this conversion can happen:

- Statically: all code in an executable file is converted without running the code
- Dynamically: code is converted at run-time, usually by looking at short sequence of instructions. This introduce a run-time overhead that is reduced by caching mechanism.

3.1.2 Hardware assisted virtualization

A **set of additional instructions is added**, in order to control the start/stop of a VM, allocate memory pages, maintain CPU state for VMs. Example of HW-Assisted virtualization are *Intel-VT* and *AMD-V*.

3.1.3 Para-virtualization

It's a technique that presents a software interface to the virtual machines, similar (not identical) to the underlying hardware-software interface. The intent is to reduce the execution time required by those instructions which would run faster into a non-virtualized environment. Para-virtualization requires substantial modifications to guest OS.

3.2 Virtual machine migration

VM should be portable, that means that they should be moved from an host to another without too much difficult. For achieving this we have two types of migrations:

- **Offline:** vm is stopped, moved and restarted.
- **Live:** vm is transferred as it is not stopped while migrations is happening (without generating a service discontinuity).

Live migration consists in the following steps:

1. Start migration: VM and the destination host are determined (can be done either by human or by automatic strategies such as load balancing)
2. Transfer memory: send VM memory to destination node (this ensure service continuity) in rounds. In first round all memory is copied and then until dirty portion of memory is small enough to handle final copy, it recopies data changed w.r.t. last round.
3. Suspend VM: send CPU and network states and suspend the source VM.
4. Activate destination host: reload states and recover execution of programs, then redirect network connection to the new VM. Old VM is removed from source host.

3.3 Software level virtualization

This kind of virtualization, also called **containerization**, can be seen as an evolution of *chroot* mechanism, it leverages on multi-programming techniques at OS level, thus implying no VMM and no HW emulation is required, instead the result consists in multiple VMs sharing same host operating system. Fundamentals for containerization are: **namespaces**, **CGroups** and **UnionFS**.

3.3.1 Namespace

A namespace wraps global resource (e.g. CGroups, IPC, Network, Mount, PID, etc.) in an abstraction so that processes within a namespace perceive that they have their own instance of it. Changes to a global resource are visible only to processes that are member of the namespace.

3.3.2 CGroups

It's a Linux kernel feature that allows processes to be organized in hierarchical groups whose usage of resources can be monitored and/or limited. The kernel's cgroup interface is provided through pseudo-filesystem called cgroupfs.

3.3.3 UnionFS

Allows administrator to keep files physically separated but logically unified into a single view. Each physical filesystem or directory is called "*branch*", while the UnionFS result is called "*merge*".

To each branch is assigned a priority and a branch with higher priority overrides branches with lower one.

This filesystem operates on directories, so if a directory exits in two branches then the content of the union directory is the combination of these two. It also automatically removes any duplicated directory entry by keeping only the file content and attributes of the higher priority branch.

3.4 Docker

Docker is a set of PaaS products that use OS-Level virtualization and containers run directly within the host machine's kernel. On the latter is installed the **Docker Engine**, which is an application that provides CLI interfaces to interact with the server (called **Docker daemon**).

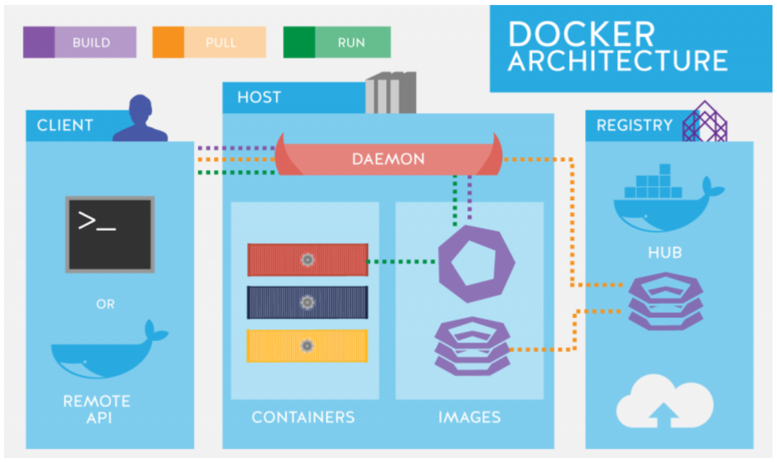


Figure 5: Docker architecture

3.4.1 Storage options

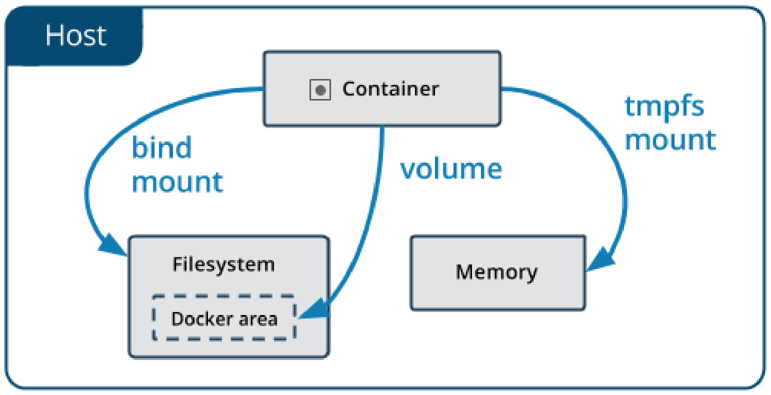


Figure 6: Type of mounts

- **Container writable layer:** all files created inside a container are stored in it. Data does not persist when container no longer exists and it can be difficult to share data with other containers.
- **Volume:** created and managed by docker (isolated from the host) is stored within a directory on the docker host.
- **Bind mount:** a file or directory on host machine is mounted into a container. Dependence on the filesystem makes this system non-portable, but faster than volumes.
- **tmpfs:** a memory area outside the container writable layer, which is shared with host or is in the container namespace. It's temporary and only persisted in the host memory. Available only on Linux hosts and not sharable among containers.

3.4.2 Network options

With docker is possible to connect container each other or connect container to non-Docker workloads by using the following network drivers:

- **bridge:** default network driver, used when applications run in standalone containers that need to communicate.
- **host:** remove network isolation between container and host and uses host's networking directly (for standalone containers).
- **overlay:** connect multiple docker daemons together, allowing communication between containers or services on different daemons.

- **macvlan:** allows to assign a MAC address to a container, making it appear as a physical device in the network.
- **none:** disable all networking.

4 Automatic Computing

Data-centers are large scale complex systems containing thousands of servers spread around the world. These structures comes with several challenges that cannot be performed only by humans, ranging from **handling HW/SW** (e.g. failures, updates, other operations) for guaranteeing **Service Level Agreements** (SLA). **Autonomic Computing** is a solution to this problem: field humans only if needed by making the system capable to manage itself.

4.1 Autonomic Computing

Given an high-level objective from system administrator, autonomic computing systems can manage themselves by performing a **MAPE** cycle of actions:

- **Monitor:** check system state (e.g. retrieve cpu load)
- **Analyze:** perform some analytic on monitored data (e.g. aggregation, prediction, comparison)
- **Plan:** decide which action should be taken for keeping the system in the desired state
- **Execute:** execute the planned action

Autonomic computing systems have the following properties:

- **self-configuration:** ability to configure and reconfigure itself under various circumstances (e.g. perform software installation and updates).
- **self-healing:** ability to recover from failures (e.g. deal with os crashes).
- **self-protection:** ability to detect and mitigate attacks (e.g. prevent attack, deal with a successful attack).
- **self-optimization:** ability to constantly monitor for optimal solution (e.g. auto-scaling).

4.2 Auto-scaling

Scalability measures the trend of performance with an increasing workload and a system is **scalable** if under that increasing workload it can maintain its performance by allocating new resources. Scaling mechanisms can be performed by:

- Replication of application (usually done on monolithic applications)

- Replication of an application components (usually done on micro-services applications)

Auto-scaling mechanisms usually define 3 things: *scaling condition* (e.g. an event triggered by a threshold exceeded), *scaling action* (e.g. scale in/out, scale up/down), *scaling amount* (e.g. add 3 VMs).

4.2.1 Auto-scaling conditions

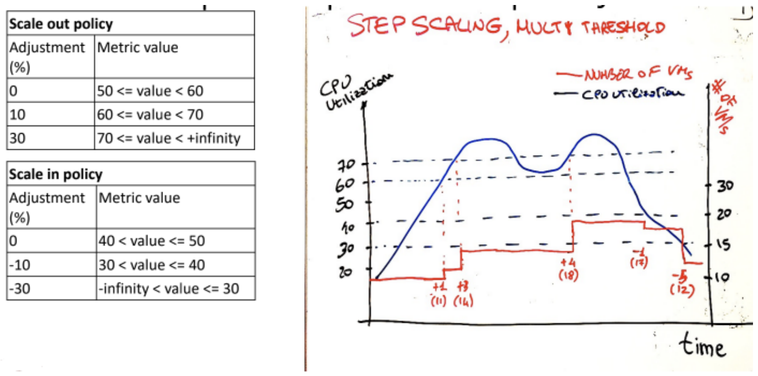
Auto-scaling algorithms can be:

- **Reactive:** take action *while* the performance are decreasing. This kind of algorithms can be based on thresholds (heuristic based, so no optimal solution) or combination of them, or either based on a mathematical model of the system which provides the scaling action that should be performed (could provide optimal solution).
- **Proactive:** predict performance metrics and takes decision few minutes ahead.

and they can be based on either a "**simple scaling**" approach or a "**step scaling**" one. The difference stands in the fact that in simple scaling we have a *cool down* time just after the scaling action has been performed, while in the other one there's no cool down. This results in lower reactivity in simple scaling and in a "*ping-pong*" problem in step scaling (e.g. increasing the number of VMs and then immediately decreasing them, and so on).

Instances that are just being started and still cannot perform at their maximum potential are considered in "*warm-up*" state. At a given time t, the "performance metric value" used by the scaling condition is calculated by taking the average of the performance metric value of all the instances that are **not in warm-up state**.

$$V_t = \frac{1}{N_t} \sum_{i=1}^{N_t} PM(VM_i)$$



Scale out policy	
Adjustment (%)	Metric value
0	50 <= value < 60
10	60 <= value < 70
30	70 <= value < +infinity

Scale in policy	
Adjustment (%)	Metric value
0	40 < value <= 50
-10	30 < value <= 40
-30	-infinity < value <= 30

Figure 7: Example of scaling

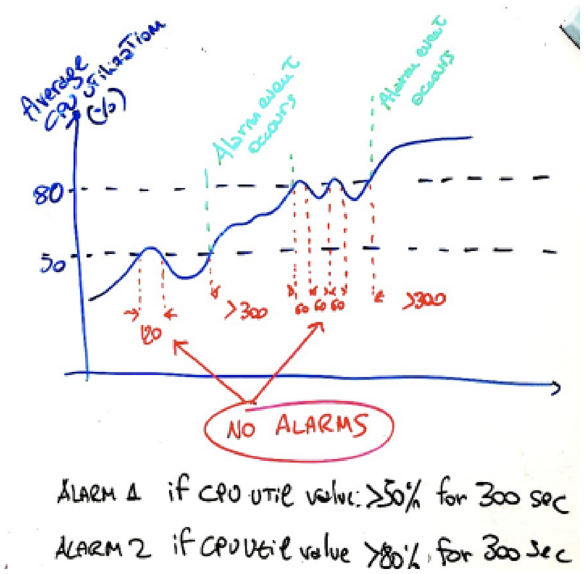


Figure 8: Example of scaling

4.2.2 Auto-scaling amount

Also called scaling policy, it defines the amount of instances that should be up. It can be defined in 3 ways:

- **Change in capacity:** define the number of instances *to add*.
- **Exact capacity:** define the number of instances that should be up after the scaling.

- **Percentage:** like the first one, but defined using percentages over the actual number of instances.

4.3 Auto-scaling in AWS

The idea is to define 3 things:

1. **Auto-scaling group:** group of EC2 instances that shares similar characteristics.
2. **Launch configuration:** EC2 template used by Auto-scaling group for new EC2 instances.
3. **Scaling policy:** policy that determine scaling action

Scaling plan and launch configurations are associated to a group, which for the latter can be defined several scaling policies (at least one must be defined).

4.4 Orchestration with Ansible

Data-centers have specialized platforms for automating tasks like provisioning, configuration, patching and/or monitoring. An example of these platforms is **Ansible**, a state-less orchestration tool which does not require to install any software on the remote machine (i.e. **agent-less**), not bringing any kind of overhead when management is not running. It runs following a "*push model*", it means that control instructions (management instructions) are sent to remote machines through existing technologies like *SSH* on Linux, or *WinRM* on Windows. Fundamentals components of Ansible are:

- **Host inventory:** set of hosts on which are applied automation tasks
- **Playbook:** a series of "plays", each of which is a set of tasks that can target one, many or all the hosts in the inventory. A playbook is defined through a `.yaml` file and it comes with a variety of features that allows to express conditional execution of tasks, gather system variables from remote machines.
- **Module:** a task to be performed, in practice a small piece of code for doing a specific task. Each module is run only if needed, and this needing condition can be expressed by using preconditions.

4.5 Kubernetes

Also referred to as K8s is a platform for managing containerized workloads and services, so it basically does the following:

- Service discovery (find services actually running) and load balancing.
- Automatically mount storage (storage orchestration).

- Automatically bring container into, so called, "desired" state.
- Distributes containers between nodes so as to have the best use of resources
- Restarts, kill, replace and does not advertise faulty containers
- Secret and configuration management (e.g. secret keys, env variables)

A K8s cluster can be split in to two macro entities category: **control plane** and **k8s nodes**.

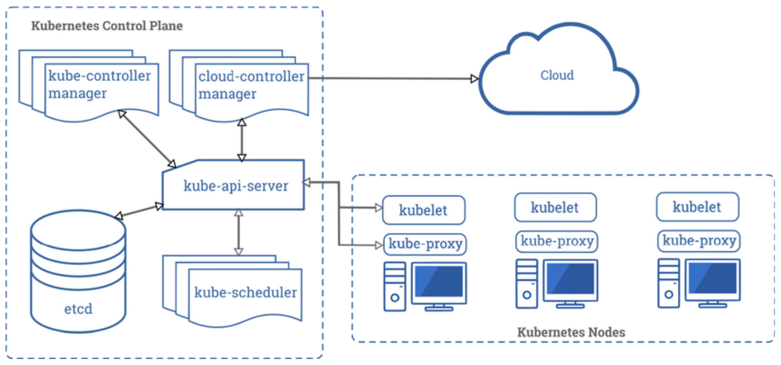


Figure 9: Kubernetes cluster components

Kubernetes is also based on three main types of objects:

- **K8s objects:** persistent entities used by k8s for representing the state of the cluster (e.g. which containerized apps are running and on which nodes, resource available, policies, and so on)
- **K8s pods:** smallest deployable units of computing that can be created and managed in k8s. It's a group of one or more containers with shared storage and network resources. The shared context is a set of Linux namespaces.
- **Node condition** (e.g. condition type, status, reason, ..).

4.5.1 K8s Nodes

Nodes are virtual or physical machines on which to run containerized workloads, they are member of a k8s cluster and are managed by the control plane. Nodes can either manually register or self-register themselves to control plane (API-Server).

Once a node is created the control plane checks for its validity and returns the node status containing information like: address, capacity and node condition (e.g. condition type, status, reason, ..).

A node is generally composed by 3 components:

1. **kubelet:** controls that container described in PodSpec are running in a pod and their status is healthy.

2. **kube-proxy:** allow communication to cluster's pods from outside or inside the cluster

3. **Container run-time:** container engine (e.g. docker)

4.5.2 Control plane components

Control plane usually runs on a dedicated server or VM and it includes the following components:

- **kube-api-server:** front-end of the k8s control plane which can be horizontally scaled.

- **etcd:** key-value store used for storing all cluster data.

- **kube-scheduler:** watches for new pods with no assigned node, and selects best node for them to run on, according to their requirements for resources. So basically the scheduler first finds feasible nodes for a pod (filtering process) and then runs a scoring phase for choosing the best node among feasible nodes. It's possible to implement a custom scheduler.

- **kube-controller-manager:** runs controller processes such as node controller, job controller, ...

- **cloud-control-manager:** link k8s cluster with a specific cloud provider's API

5 Cloud data storage

Multiple forms of data storage are used in cloud systems (e.g. distributed file systems or databases), the common goals are: massive **scaling** on demand, high **availability** and **simplified developing** and **deploying** methodologies. For achieving this several things are needed to take into account.

5.1 Atomicity

It's the property which guarantee that multi-step operations are executed as if they were single-step operations (i.e. without any interruption). Atomicity needs HW support which must provide two basic primitives operations: **test-and-set** and **compare-and-swap**. These two allows us to implement critical section handling mechanisms like *semaphores*, *lock* or *monitors*.

Two kinds of atomicity:

- **All or nothing:** two phases, the pre-commit phase which consists in a set of operations that are either all finish or none at all (strategy for masking failures during execution).
- **Before or after:** result of every read or write is the same as if that read or write occurred either before or after any other read or write (necessary for coordination of concurrent activities).

5.2 Storage models

Storage model describe the layout of the data structure of the physical model (disk or solid state) and is desirable to have read/write coherence, before-or-after and all-or-nothing atomicity. Two models:

- **Cell storage model:** assumes that storage consists of fixed size cells and that each object fits exactly one cell. Read/write coherence is not guaranteed.
- **Journal storage model:** stores records consisting of multiple field and it keep tracks of all the versions of all the variables stored (log). It consists of two components: the **manager** and the **cell storage**. The user does not directly access to the storage, but instead it can request to the manager to: create a new action, read/write a cell or commit/abort an action. An online transaction is basically an all-or-nothing action which record the action into the journal storage and then apply the change in the cell storage overwriting the previous version.

5.3 Google file system

GFS has been designed after a careful analysis of the file characteristics and of the access models:

- File size ranging from few GB to 100s of TB. GFS uses thousands of inexpensive components to provide petabyte of storage.
- Files are often *read sequentially* and random writes operation are rare compared to appending operations (this allow them to use HDDs over expensive SSDs).
- Response time is not a main requirement, data are processed in bulk.
- High reliability to hw/sw failures or human errors.
- Relaxed consistency model to simplify the system implementation.

GFS architecture is accessed by multiple clients and consists of a single **master** and multiple **chunk-servers**. Files are stored in chunk-servers as Linux files and are organized in chunks of fixed size (64 MB). This choice gives improvements over performances such as: reduced time-to-search for a specific file, increase likelihood of locality principle, optimize performances for large files. For reliability, each chunk is replicated on 3 chunk-servers.

The master maintain file system metadata such as the namespace, access control information, mapping from files to chunks, current location of chunks. Having a single master simplify the system design, but its involvement should be minimized in order to avoid SPOF or bottlenecks. Master handle file creation, but instead it does not handle any kind of read/write operation (i.e. clients never r/w data through master). Information regarding where each chunk replica is actually stored is retrieved during start-up phase or whenever a new chunk-server join the cluster.

Clients do not cache file retrieved by chunk-servers (because of their big size), but instead they cache metadata provided by the master when they ask for a specific file (they ask for chunk location).

In case of failure, master recovers its file system state by replaying the operation log (supported by checkpoints for speeding up this process).

5.4 Hadoop filesystem (HDFS)

Hadoop provides a distributed filesystem and a framework for analysis and transformation of big dataset using *MapReduce paradigm.* HDFS is based on a Master/Slave model, in which the master (called **NameNode**) stores metadata, and application data are stored in slaves servers called **DataNodes**. File content is split into large blocks of fixed size (64-128 MB) which are replicated on multiple DataNodes for reliability (similar concept to chunk in GFS).

5.5 NoSQL datastore

Cloud applications are often based on "*online transaction processing*" (**OLTP**), which usually requires low response time. This could be achieved in two ways: *(1)* by using a caching system and/or *(2)* by scaling those applications. Scalability (horizontal scalability) in OLTP is a big problem since issues of consistency among distributed copies can surely happen, especially by using a relational database; that because those kind of db *must* preserve **ACID properties**.

Non-relational database models (**NoSQL**) allows us to relax on the consistency property, allowing data to be inconsistent and by introducing the concept of "**eventual consistency**" which means that data at some future point in time will be consistent, instead of enforcing it when data is "committed" (something like a lazy approach).

5.5.1 Google Big Table

It's a storage system designed and developed by Google which provides high scalability, availability and performance and it's meant to be used in *throughput-oriented*, *batch-processing* and *latency-sensitive* jobs.

Big table is a sorted map which corresponds to the following mapping function:

$$f : string \times string \times timestamp \rightarrow string$$

or in another way:

(row: `string`, column: `string`, time: `int64`) $\rightarrow$ `string`.

Big table row is an arbitrary string of maximum 64KB, but usually it ranges from 10 to 100 bytes. Read and write operations are atomic and rows are partitioned into **tablets**, the unit base for load balancing. Keep in mind that reading small ranges is very efficient, so users should exploit this for improving locality.

Big table column grouped into sets called **column families**. A column key structure is typically *family:qualifier*, and data stored in the same family are of the same type.

Big table timestamp allows to version control cell's data. Versions are indexed by timestamp, which are sorted in decreasing temporary order so that the most recent version is the first to be read. The assignment can be performed by either the big table itself, or by the client application (collision handling required). Versions are garbage collected by only keeping last n versions of a cell and by keeping only new-enough versions (e.g. latest 7 days).

Big table architecture is based on GFS so it's a master/slave architecture in which a BigTable cluster can store multiple tables and each table is composed by several tablets. The architecture:

- **Master** is responsible for assigning tablets to tablet servers, detect add/delete of tablet servers in the cluster, balancing tablet-server load and handle schema changes.
- **Tablet servers** can be dynamically added or removed to accommodate the workload and each tablet server is responsible to manage a set of tablets, handle read and write requests, split tablets that has grown too much

Two other building blocks of this system are:

1. **SSTable:** file format used to store BigTable data. Contains sequence of blocks and corresponding block index.

2. **Chubby:** provides distributed lock mechanisms based on files and directories. It's used for ensuring that there's at most one master active, store bootstrap location of BigTable data, discover tablet server and finalize their death, store BigTable schema information, store access control lists.

5.5.2 Amazon's DynamoDB

Highly available, scalable and distributed key-value store model, used for managing state of services which require very high reliability. It's based on optimistic replication techniques (i.e. replicas can diverge but at some point in time they will be consistent), which are used for increasing the availability and decreasing latency. Changes are allowed to propagate to replicas but conflicts must be detected and resolved, usually using a consensus algorithm. A smart way of resolving conflicts (used by DynamoDB) is to always allow writes and then conflicts are resolved at read time, so only when needed.

Key design principles incremental scalability, symmetry, decentralization, heterogeneity.

System interface consists of two main functionalities:

1. `Get(key)`: locate object replicas associated with the key, returning a single object or a list of versions of it.
2. `Put(key, context, object)`: locate where the object should be placed based on the associated key.

Partitioning algorithm we can see the cluster as an hash table in which each node is a bucket. Given a key, to determine item location, the item's key is hashed to yield it's position. The uniform distribution is given by a good hash function. Virtual nodes concept is introduced for nodes that become unavailable: load handled by this node is distributed across all other nodes until a new node join the system.

Replication for achieving high availability data is replicated on multiple nodes, let's say N hosts. Data is then replicated to N-1 clockwise successor nodes in the ring. The ring is given by assigning a range of hash values to each node.