Cloud Computing

Cloud native applications – Architecture and Orchestration

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2024

References

- These slides are mostly adapted from the slides of Renaud Lachaize

Main references (1/2)

- B. Scholl, T. Swanson, P. Jausovec. **Cloud native: Using containers, functions, and data to build next-generation applications**. O'Reilly, 2019.

- J. Garrisson, K. Nova. **Cloud-native infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment.** O'Reilly, 2017.

- B. Burns, J. Beda, K. Hightower, L. Evenson. **Kubernetes up & running (3rd edition)**. O'Reilly, 2022.
  - Freely available from VMware (registration required): https://k8s.vmware.com/kubernetes-up-and-running/

- B. Burns. **Designing distributed systems. Patterns and paradigms for scalable, reliable services**. O'Reilly, 2018.
  - Freely available from Microsoft (registration required): https://azure.microsoft.com/en-us/resources/designing-distributed-systems/

Main references (2/2)

- Kubernetes documentation: https://kubernetes.io/docs/home/
- Cloud Native Computing Foundation (CNCF) Web site: https://www.cncf.io
- Jordan Webb. The Container orchestrator landscape. LWN.net. August 2022. https://lwn.net/Articles/905164/
- How Kubernetes reinvented virtual machines. Ivan Velichko. August 2022. https://iximiuz.com/en/posts/kubernetes-vs-virtual-machines/

Outline

- ~~Origins and main characteristics~~
- **Microservices**
- Container orchestration
- Design patterns

Definition

By J. Lewis and M. Fowler

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- These services are built around business capabilities and independently deployable by fully automated deployment machinery.
- There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

The key concepts

- Architectural style
- Suite of small services
- Built around business capabilities
- Independently deployable
- Communicating with lightweight mechanisms
- Written in different programming languages
- Use different data storage technologies

Example of a media application

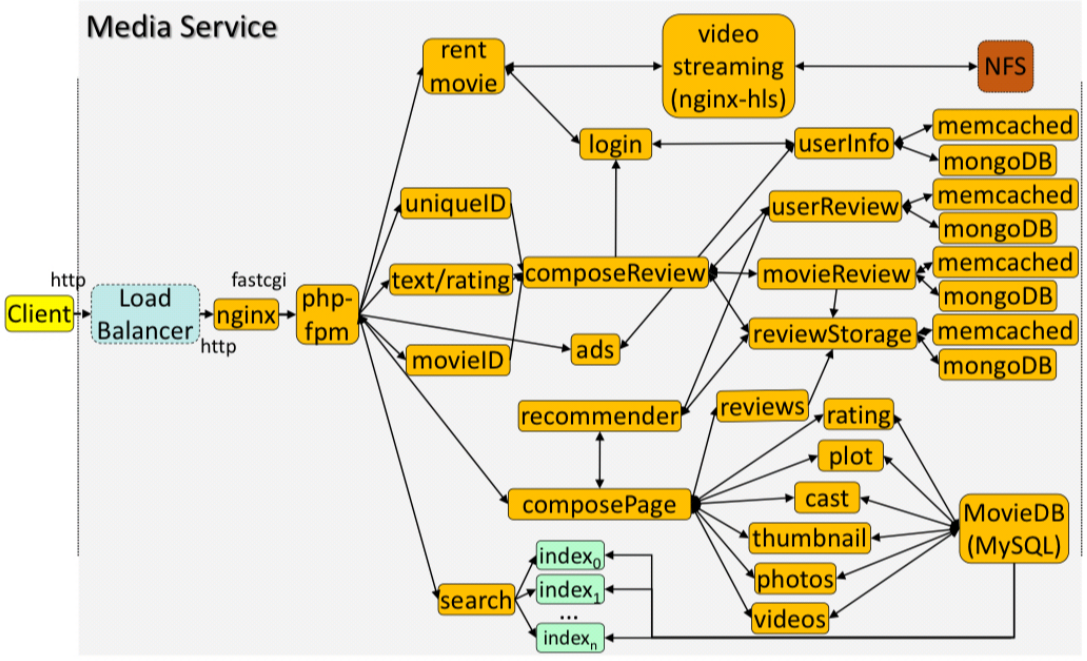


Figure 5. The architecture of the *Media Service* for reviewing, renting, and streaming movies.

See: Gan, Yu, et al. "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems.", ASPLOS 2019.

The key concept of microservices

- Independent deployability
- Owning their own state
- Alignment with organization + structured based on business domains
- Size

Advantages of microservice architectures

- Technology heterogeneity
- Robustness
- Scaling
- Ease of Deployment

Microservices – Challenges

- **Complexity** (e.g., due to increased distribution)
- API versioning and integration
- Data integrity and consistency
- Monitoring and logging
- Service discovery and routing
- Availability
- Performance issues (e.g., increased average and tail latencies)
- Achieving insightful and efficient end-to-end observability (distributed, large-scale debugging and profiling

Outline

- Origins and main characteristics
- Microservices
- **Container orchestration**
- Design patterns

We need to say a few words about Virtual Machines and Containers

Infrastructure building blocks for general-purpose computing tasks

- **Goals**: **Deploying, running, managing:**
  - *… arbitrary tasks*
  - *… made of arbitrary code*
  - *… in a flexible, convenient, secure, and efficient way (on a cloud platform).*

- Such an infrastructure relies on two major software building blocks:
  - Virtual machines
  - Containers

- Vocabulary: the above tasks are called **"guest code"** and are managed by the **"host" environment**.

Virtual machines (1/6)

- *A (system-level) virtual machine is an **efficient & isolated duplicate** of a real (physical) machine*
- Often abbreviated as “VMs” or “Guest VMs” or “Guests”
- Machine **resources** include CPU(s), main memory (RAM), I/O devices (disks, NICs, peripherals …)
- Goals:
  - **“Duplicate”:** code running in a VM cannot distinguish between real or virtual hardware
  - **“Isolated”:** Several VMs execute concurrently on the same machine without interfering with each other (at least w.r.t. safety and security considerations)
  - **“Efficient”:** VMs should execute at a speed close to that of real hardware

Virtual machines (2/6)

The resources exported by a virtual machine may or may not correspond to the ones of the underlying physical hardware.

Typically, in practice, on a Cloud server:

- **Regarding the functional interface:**
  - The VM exports the same CPU model as the one of the physical machine (*same ISA: Instruction Set Architecture*).
  - The VM may or may not export the same types of I/O devices as the one of the physical machine.
- **Regarding the amount of available resources:**
  - A VM typically exports fewer resources than the total of physical resources.
  - Two main reasons:
    - Concurrent execution of multiple VMs (with decent performance)
    - Virtualization overhead

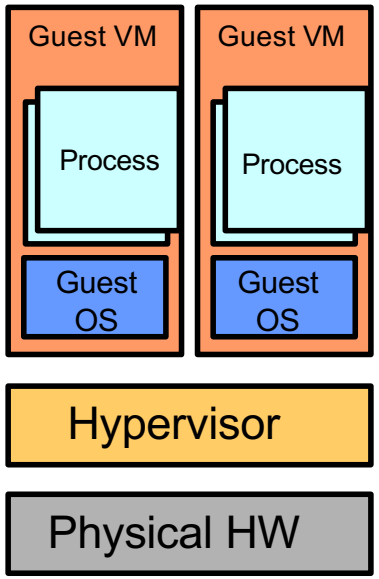
Virtual machines (4/6)

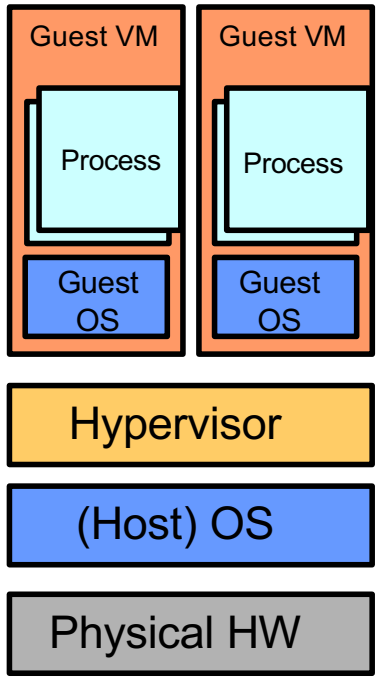
Where is the hypervisor in the software stack?

- There are several possibilities.
- On server systems, for performance reasons, the hypervisor is typically the lowest software layer (which directly controls the hardware).

In any case:

- Each guest VM has it own (guest) OS instance.
- Different VMs may have similar or different guest OSes.
- The expression “host software” corresponds to the layer(s) below the guests.



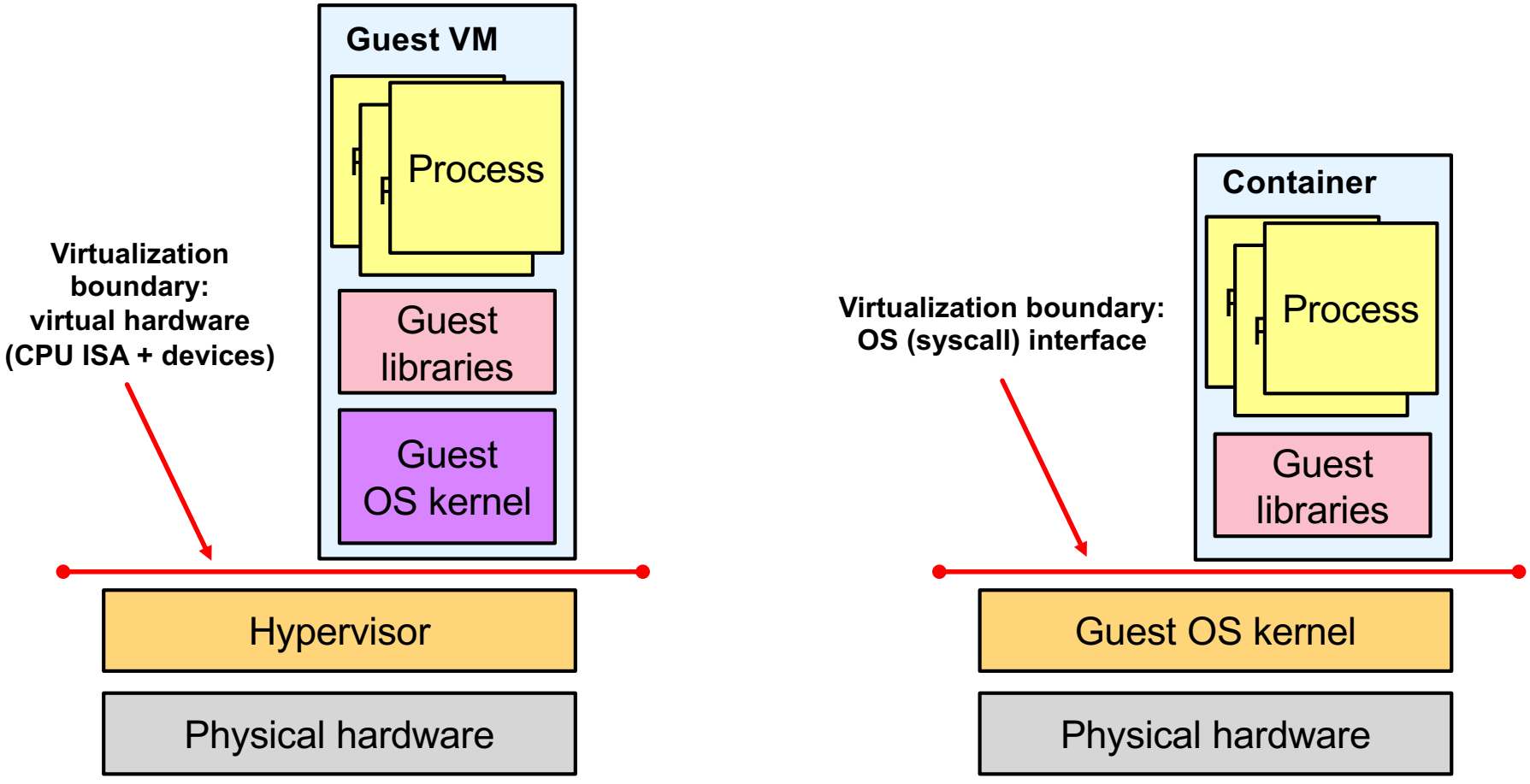


Containers (1/5)

- *Unlike (system-level) virtual machines,* ***containers support virtualization at the level of the OS interface*** *(rather than at the hardware interface).*
- Hence, they are also known as **"OS-level containers"** or **"OS-level virtualization"**.

- Roughly speaking, a container is akin to a "process group" in a traditional OS … yet with more isolation guarantees regarding security, performance and software configuration.
- Different containers running on the same machine **share the same underlying host kernel**. There are no guest kernels.

Containers (2/5)



Containers (4/5)

The containers ecosystem also includes **tools and facilities to simplify the management of container images** (i.e., the files to be included in the file system within a container).

- **Application packaging**
  - Management of executables, libraries, and configuration files
  - Management of version numbers and dependencies
  - Layered file system, allowing to define new images based on existing ones, in a simple and space-efficient way
- **Distribution and sharing of images**
  - Repository (“hub”) of existing images
  - Facilitated by the fact that most container images are lightweight (and layered)

Containers (5/5)

In practice, the management of containers is addressed via a set of software tools, which encompass **different needs**:

- Building **container images**, managing images, sharing and downloading images
- Managing **container instances**, running containers

There exists several tools with roughly similar features, like Docker and Podman.

The above-mentioned tools are themselves based on several modular building blocks, among which:

- **"Low-level runtimes"** focused on the machinery for running containers. Example: *runc* (used by Docker).
- **"High-level runtimes"** focused on support for download/managing container images and running a container from an image. Example: *containerd* (used by Docker).

Virtual machines and Containers (1/3)

Virtual machines and containers **share a set of common design goals**:

- **Deployment:**
  - Notion of “virtual appliance”: Encapsulating code (applications, libraries, …) & configuration information to make software components more portable across machines and hosting environments.

- **Security isolation (a.k.a. “sandboxing”):**
  - Preventing “guest” code from performing unauthorized actions and accessing unauthorized data
  - In particular, preventing unwanted interactions with:
    - The (code and data of the) host and the other guests
    - The hardware resources of the machine (including the I/O devices)

- **Performance isolation:**
  - Precisely controlling the amount of low-level resources granted to each guest.
    - Avoiding/mitigating interferences between guests
    - Avoiding resource exhaustion/saturation (denial of service)
    - Possibly differentiating QoS between guests

Virtual machines and Containers (2/3)

However, virtual machines and containers also **have significantly different characteristics regarding some aspects:**

- **Dependencies for portability:** Hardware interface vs. OS interface (ABI)
- **Memory and disk footprint:** Containers are more lightweight.
- **Startup and shutdown latency:** Containers are faster.
- **I/O performance:** Depending on the chosen setups, VMs and/or containers may have non-negligible overheads for network- or disk-sensitive workloads. (There is no clear performance hierarchy between the two).
- **Syscall performance:** Same remark as above for syscall-intensive workloads.
- **Security:** VMs are arguably more secure. However, VM and container technologies both have a large attack surface.
- **Live migration (across physical hosts):** VMs have more mature/robust support.
- **Support for stateful (vs. stateless) workloads:** VMs have more mature/robust support.

Virtual machines and Containers (3/3)

These two technologies are not necessarily antagonist and mutually exclusive.

- In public clouds, containers are often/typically deployed within virtual machines.

- Modern “container orchestration” systems are agnostic regarding the actual container implementation and can use VMs as a replacement.
  - For example, in the Kubernetes orchestrator, the CRI (*container runtime interface*) specification is also compatible with virtual machines.

- Many recent facilities integrated in host operating systems can be leveraged by both technologies.
  - For example, on Linux, the seccomp and eBPF subsystem available for secure and efficient sandboxing & monitoring of guest code.

Container orchestration (1/2)

- **Vocabulary warning**: Not to be confused with the expression “service orchestration”, which is often used with a somewhat different meaning.

- A **container orchestrator** is:
  - **a software system**
  - … in charge of **simplifying** (through automation) **the management of a fleet of container-based applications**
  - … **on a cluster** of (virtual or physical) machines.

Container orchestration (2/2)

The **main duties** of a container orchestrator include:

- The **provisioning** and **deployment** of containers
- The setup of the **network configuration** of the containers

- The **placement decisions** for the containers on the cluster nodes

- **Health monitoring** (for containers and nodes) and appropriate reactions when needed

- **Load balancing** between containers
- **Autoscaling** decisions (mostly for horizontal scaling but also possibly for vertical scaling)

Container orchestration – Case study: Kubernetes

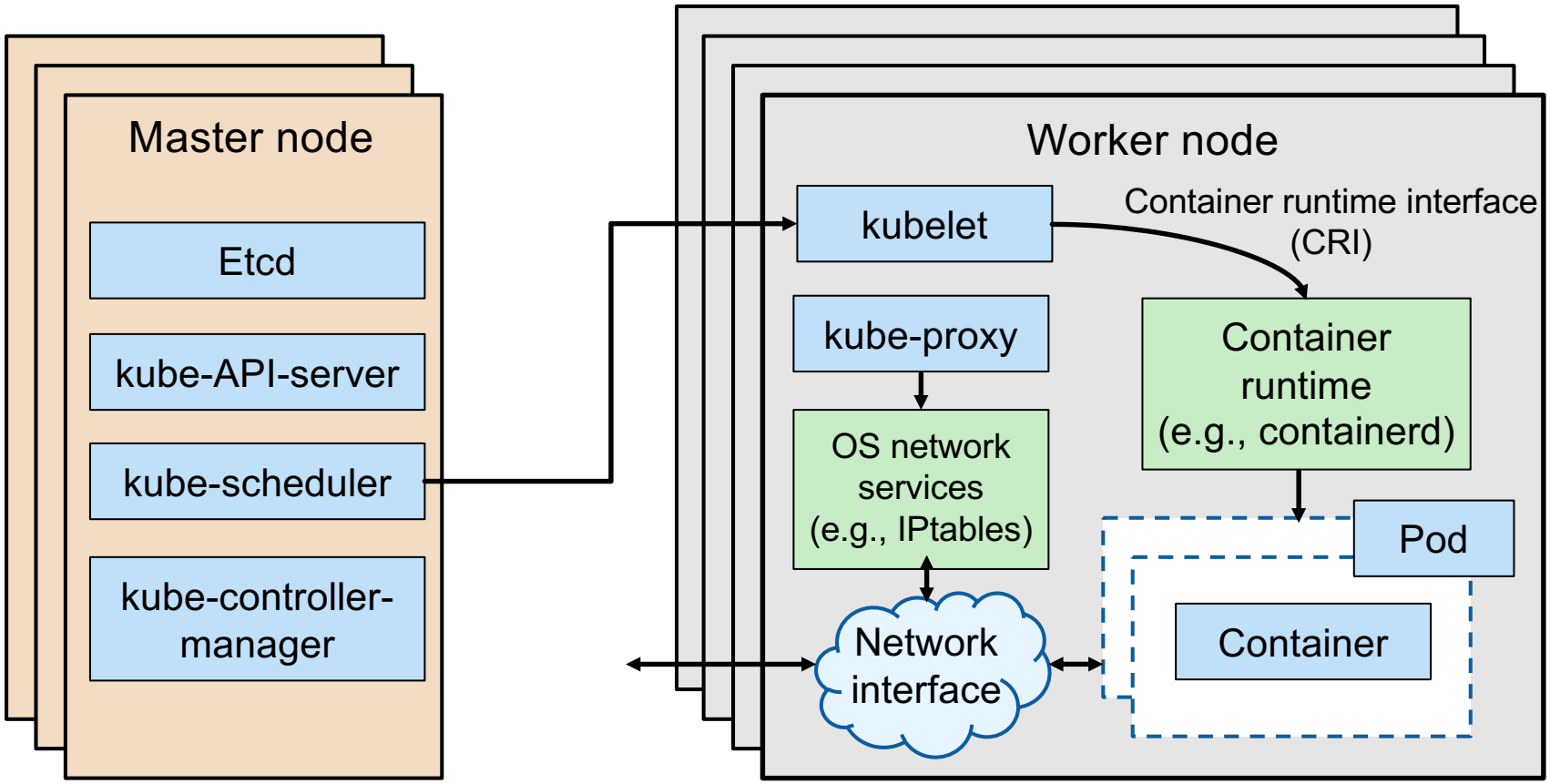
- Kubernetes: currently **the most popular container orchestrator**
- Also known under the abbreviation “k8s”
- A project initially developed by Google and influenced by the design of previous (closed, internal) cluster/container management systems used within the company.
- Released as open source in 2014. Now managed by the Cloud Native Computing Foundation (CNCF).
- Implemented in the Go language.

- A good/short introductory reference (for the description of the design principles):
  - B. Burns et al. Borg, Omega and Kubernetes: Lessons learned from three container-management systems over a decade. ACM Queue. January-February 2016. https://static.googleusercontent.com/media/research.google.com/fr//pubs/archive/44843.pdf

Kubernetes components

- Kubernetes is a distributed system based on 3 main categories of components.
  - **Master components:** Provide the **"control plane"** of the cluster.
    - Make global (cluster-wide) decisions. E.g., regarding the scheduling/placement of tasks, the number of replicas for a given service, the response to specific events like failures.
    - Are typically deployed on dedicated nodes for better robustness.
  - **Node components:** Provide the **"data plane"** of the system. Act as local agents on the cluster resources :
    - For health and performance monitoring
    - For handling the orders issued by the control plane (e.g., running a new container)
    - For setting up the network connectivity of the containers
  - **Addons**: Provide additional services, such as:
    - DNS services
    - High-level user interfaces (e.g., management/monitoring dashboards)

Kubernetes components overview



Kubernetes components – Details (1/5)

- **Master components**
  - **Kube-apiserver**: the **front end for the Kubernetes control plane**
    - Exposes the API of the control plane
    - Designed for horizontal scaling (load balancing)
  - **Etcd**: a strongly consistent and highly available **key-value database**
    - Used to store all the cluster configuration/metadata
  - **Kube-scheduler**: monitors the creation of new “Pods” and selects nodes for them
    - Placement decisions for Pods are based on various factors, including: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality …

Kubernetes components – Details (2/5)

- **Master components (continued)**
  - **Kube-controller-manager**: runs (most of) the Kubernetes controllers
    - Logically, each controller is a distinct process
    - However, to reduce complexity, all controllers are compiled into a single binary and run in a single OS process
    - Some controllers can also run externally
  - **Cloud-controller-manager**: runs controllers that interact with the underlying cloud provider
    - Allows the cloud provider's code and the Kubernetes code to evolve independently of each other

Kubernetes components – Details (3/5)

Remarks:

- Master components are **replicated on multiple master nodes** for failover and high availability.

- **"Managed Kubernetes" offerings** typically offload the burden of hosting/administrating these master components from the end-users (cloud tenants).
  - Examples : Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), AWS EKS.

Kubernetes components – Details (4/5)

- **Node components**: Run on every compute (worker) node of the cluster
  - **Kubelet**: local agent in charge of running containers in "Pods"
    - Reads and applies the Pods specifications
    - Checks the health of the containers
    - Does not manage the containers that were not created by Kubernetes
  - **Kube-proxy**: local network proxy
    - Maintains the network configuration/rules on the cluster nodes (regarding communication with other cluster nodes and external nodes), in order to support the Kubernetes "service" concept.
    - Typically relies on the local OS services (e.g., packet filtering) to implement traffic forwarding.
  - **Container runtime**: in charge of executing containers
    - The Kubernetes CRI (Container Runtime Interface) provides an abstract interface that allows plugging various runtime implementations (e.g., Docker, containerd, cri-o, …)

Kubernetes components – Details (5/5)

- **Add-ons**: Some examples
  - **DNS**: Serves DNS records for Kubernetes services
    - Containers started by Kubernetes automatically include this DNS server for their DND lookups
  - **Web UI** (Dashboard)
    - General-purpose Web-based user interface for Kubernetes clusters
    - Allows managing and troubleshooting the Kubernetes cluster, and applications running on the cluster
  - **Container resource monitoring**
    - Records generic time-series of metrics about containers in a centralized database
    - Provides a user interface to browse the collected data
  - **Cluster-level logging**
    - In charge of collecting and saving container logs to a central store with a search/browse interface.

Kubernetes – Main concepts (1/4)

- **Pods**:
  - **A pod is an abstraction encapsulating a set of containerized components.**
  - A pod is the basic scheduling unit for Kubernetes.
  - There are two main setups in practice:
    - Pod with a single container
    - Pod with multiple containers that work together
  - **All the containers within a pod are guaranteed to be deployed on the same machine and can share resources.**
  - Each pod runs a single instance of an application (there is no replication/sharding within a pod).
  - Networking:
    - Each pod is assigned a unique IP address.
    - Containers within the same pod share the same network namespace (including network ports).
    - Containers within the same pod can communicate using localhost.
  - Storage:
    - A pod can specify a set of shared storage volumes, which can be accessed by all the containers within the pod.

Kubernetes – Main concepts (2/4)

- **Service**:
  - In Kubernetes, a “service” is an “endpoint” abstraction allowing to expose an application (running as a set of pods) as a network service.
  - Useful to abstract the fact that the pod instances providing a given functionality can vary over time (e.g., due to scaling or load balancing).
- **Volume**:
  - On-disk files in a container are ephemeral (they disappear upon termination of the container), which can be limiting in certain setups/situations.
  - Volumes survive container crashes/restarts and support sharing between containers in a pod.
  - The lifetime of a volume corresponds to the lifetime of the enclosing pod.
  - Many types of backing storage systems are supported.
- **Persistent volume:**
  - Unlike a standard volume, a persistent volume can exist/survive independently from the pod(s) that may consume it. The same persistent volume may be used successively by different pods.

Kubernetes – Main concepts (3/4)

- **Namespace**:
  - Kubernetes allows using multiple virtual clusters on the same physical cluster.
  - Such a configuration is useful for environments in which there are many users spread across multiple teams and projects.
  - The notion of “namespaces” is used to define such virtual clusters.
    - Provides a scope for naming resources (a resource name must be unique within a given namespace).
    - Supports resource quotas to control the division of resources.

Kubernetes – Main concepts (4/4)

- **Controllers**:
  - A concept inspired from other fields like robotics and automation/control theory.
  - A **"control loop":**
    - is an infinite loop whose purpose is to **regulate the behavior of the system**.
    - continually monitors the current state of the system (here, a cluster) and acts accordingly **in order to reach a desired state** (analogy: a thermostat).
  - **Kubernetes uses many different controllers**, each in charge of a specific aspect of the cluster state.
    - A set of **built-in controllers** (running inside the kube-controller-manager) supporting the important core behaviors.
    - Also possibly additional, **user-defined controllers** (that can run within Kubernetes, as a set of pods, or outside).

Kubernetes – Some examples of controllers (1/2)

- **DaemonSet**: **ensures that all (or some) nodes of the cluster run a copy of a Pod.**
  - For example, useful to run a storage daemon or a logging/monitoring daemon on every node.

- **ReplicaSet**: **maintains a stable number of replica pods running at any given time.**
  - Typically used to enforce a given replication factor in order to provide high availability guarantees.

- **Deployment**: **provides declarative updates for Pods and ReplicaSets.**
  - A ”deployment object” is used to describe a desired state.
  - Then the deployment controller changes the actual state at a controlled rate, in order to reach the desired state.
  - Useful for many purposes: rolling out and monitoring ReplicaSets, scaling ReplicaSets, updating Pods, rolling back to earlier deployment versions, …

Kubernetes – Some examples of controllers (2/2)

- **StatefulSet**: **manages applications that have one or more of the following requirements:**
  - Stable, unique network identifiers
  - Stable, persistent storage
  - Ordered, graceful deployment and scaling
  - Ordered, automated rolling updates