

(An Overview of) The Basic Building Blocks of Cloud Computing

Renaud Lachaize & Thomas Ropars

Univ. Grenoble Alpes

M2 GI

January 2024

Outline

- General-purpose computing
- Storage
- Networking (very brief overview)

Outline

- **General-purpose computing**
 - Virtual machines
 - Containers
 - Physical machines

- Storage

- Networking

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 (3/6)

- The software layer in charge of supporting (system-level) virtual machines is called a “**Hypervisor**” or a “**Virtual Machine Monitor**”.
 - **Warning:** These two expressions are often use interchangeably. However, in some designs/documents, they correspond to different parts of the virtualization system.
- In order to achieve better virtualization performance (i.e., lower overhead), nowadays **most hypervisors partially rely on support from the physical hardware**.
 - For CPU and main memory virtualization
 - Also possibly for some types of I/O devices (e.g., high-speed network interfaces)

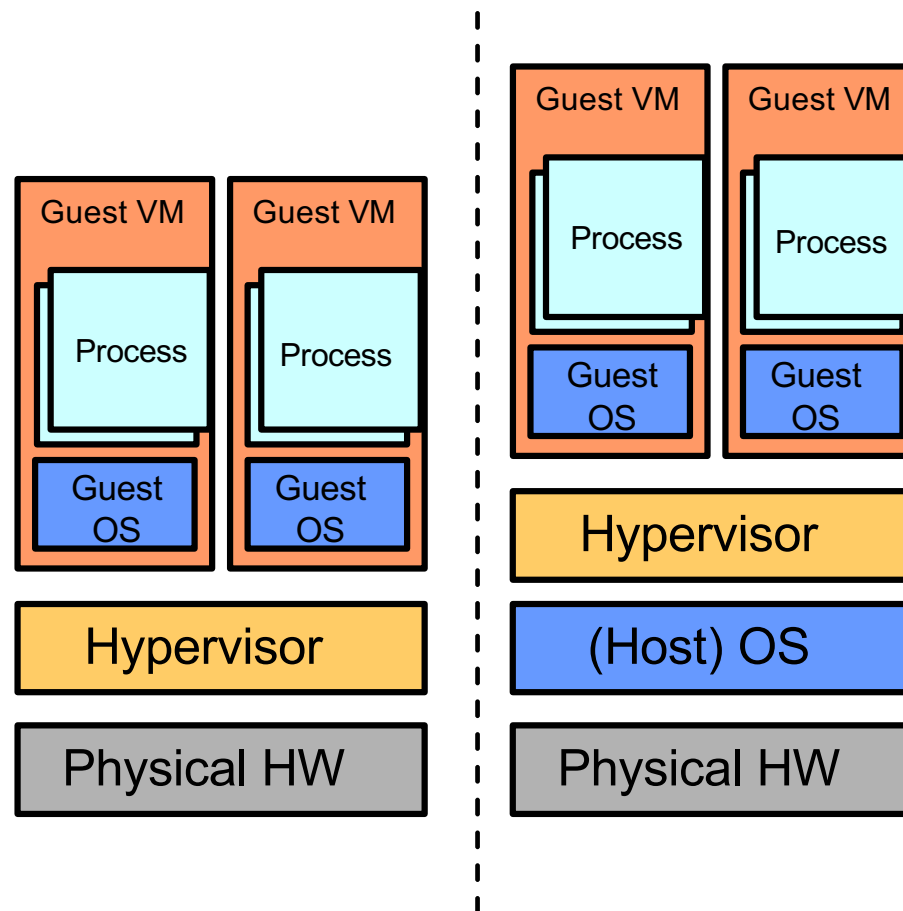
Virtual machines (4/6)

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 its 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.



Virtual machines (5/6)

Warning: Beware of confusions between the concept of “hypervisor” and the other types of software layers listed below.

- **Language-level virtual machines:**

- Examples: Java Virtual Machine (JVM), Python VM
- Provide higher-level abstractions
- Generally rely on a standard OS interface

- **Emulators:**

- Conceptually closer to hypervisors
- But with significant differences between the physical and virtual hardware (e.g., different ISA)
- Mostly used for backwards compatibility or cross-development/testing

- **Hardware simulators:**

- Aimed at precisely modelling the internal behavior of computer hardware (CPUs and/or devices)
- Very different trade-off in terms of speed vs. accuracy
- In contrast, hypervisors are only focused on externally-visible behavior of the hardware resources.

Virtual machines (6/6)

Most hypervisors rely on one or several forms of optimizations in order to improve the performance of the guest VMs.

- **Paravirtualization:**

- Principle: *Make the guest software “aware” of the fact that it is running in a virtualized platform.*
- Involves various degrees of modifications in the guest OS:
 - At least, installation of specific device drivers
 - Also possibly deeper modifications of the guest kernel
- Guest applications generally not modified

- **Direct device assignment:**

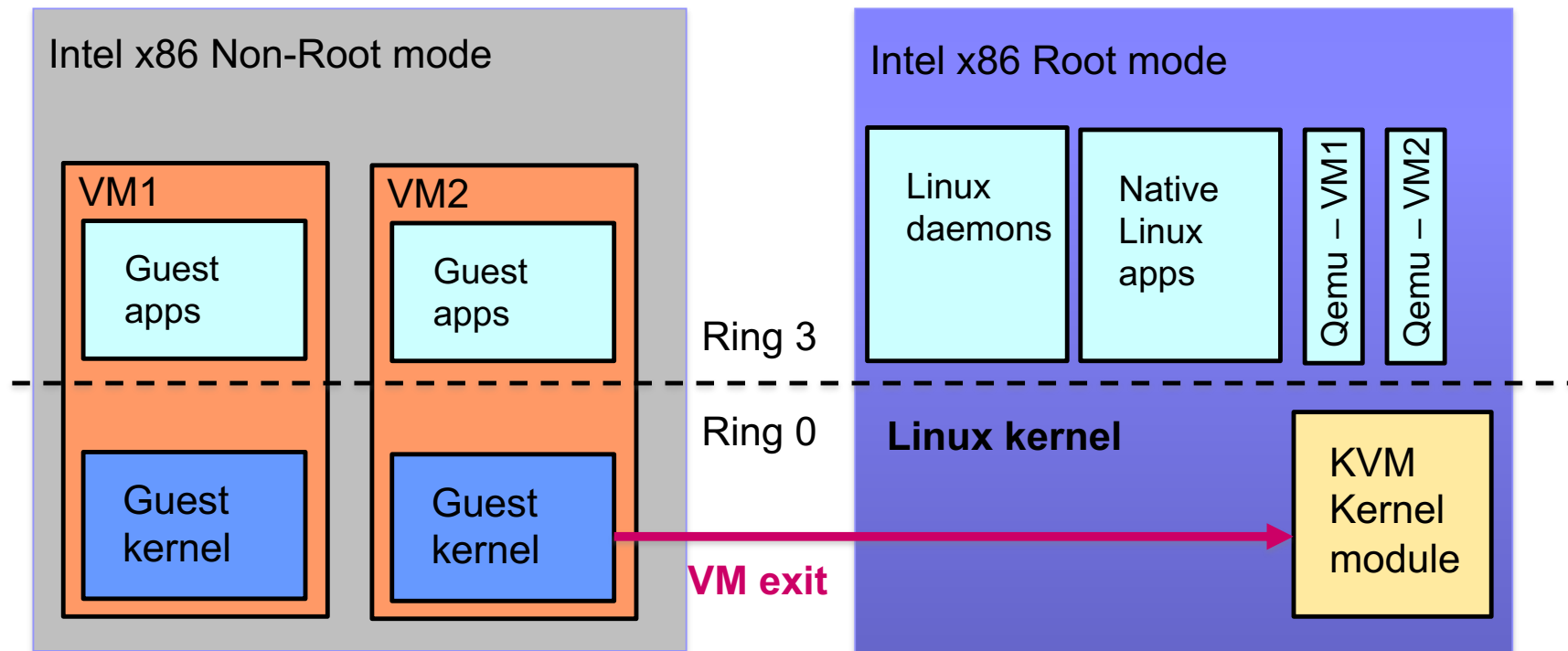
- Principle: *Hypervisor gives full/direct control of a given physical device to a given guest.*
- In practice, this usually relies on self-virtualizable hardware devices.

Hypervisor design

- There exists various hypervisor designs for server machines.
- Each design has **different characteristics and trade-offs regarding different dimensions**:
 - Virtualization performance
 - Resource footprint
 - Security and attack surface
 - Code maintainability and reuse
- Next, we will study **two mainstream open-source hypervisors**: Xen and Linux KVM,
 - In our illustration, we will focus on the versions designed for the Intel x86 architecture.
- **Some general trends**:
 - Hypervisor **code bases have become very large and feature-rich**. The attack surface and the number of security vulnerabilities is significant.
 - **More and more virtualization operations are offloaded to the hardware**, for performance and security reasons.

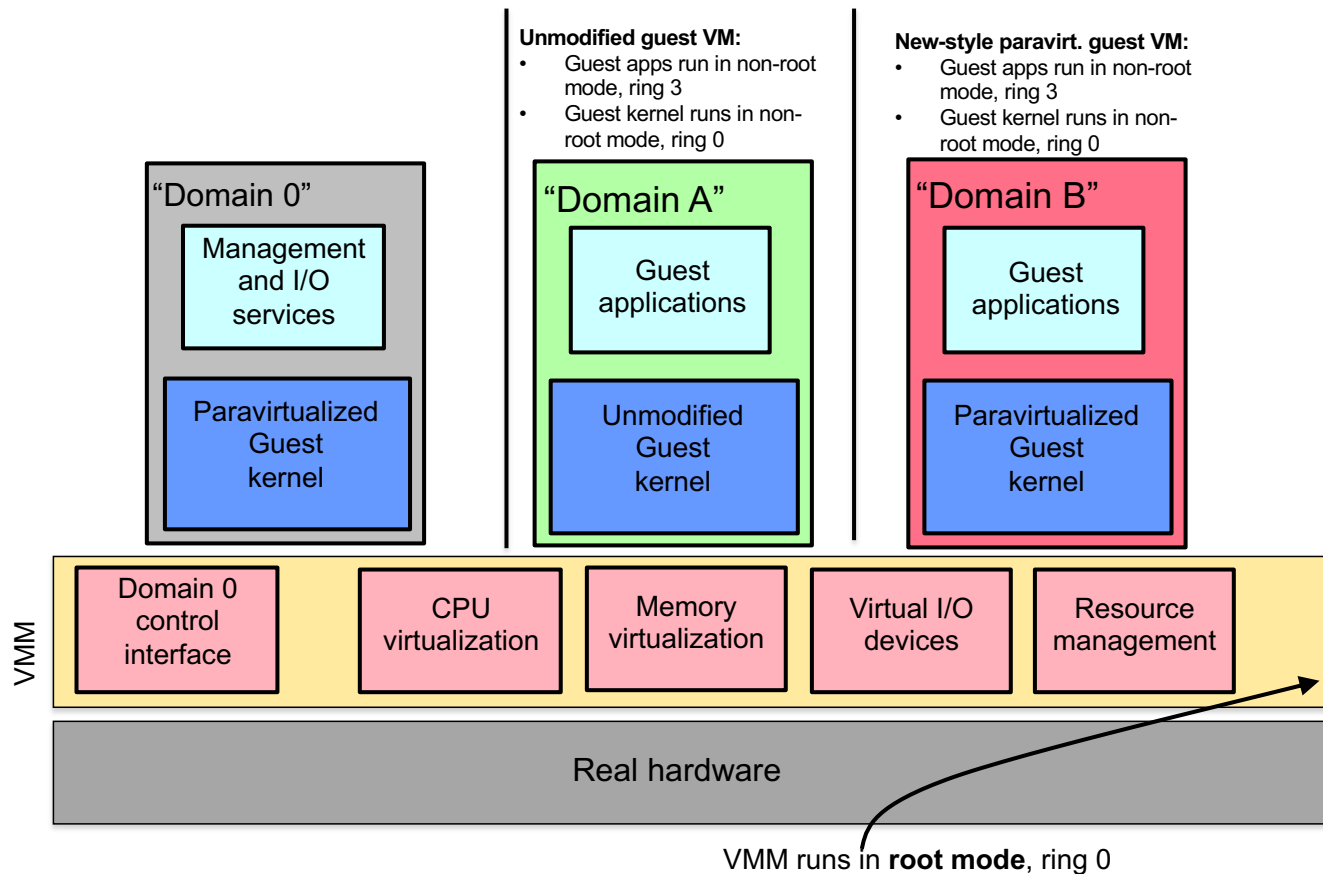
Hypervisor design – Example #1: Linux/KVM (Intel x86-64 version)

- KVM consists in a Linux kernel module + other software components allowing the Linux kernel to become a hypervisor.



Hypervisor design – Example #2: Xen (Intel x86-64 version)

- Xen consist in a native hypervisor + a specific VM (Linux-based, named “domain 0”) in charge of provide control and I/O services.

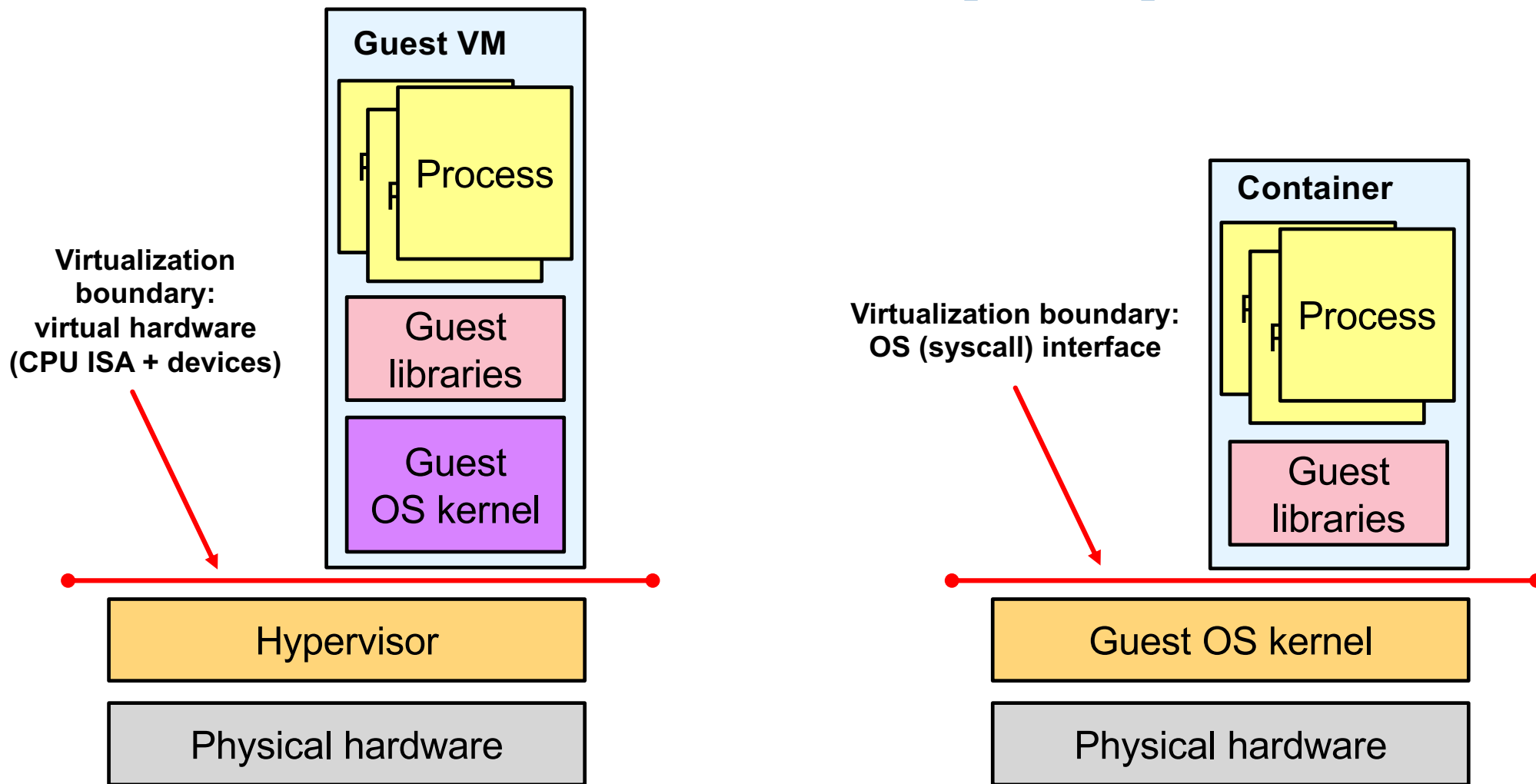


Containers (1/5)

- 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 (3/5)

On Linux, the concept of containers is not a unified facility provided by the operating system.

Rather, *the management of containers is achieved by **combining a set of independent facilities*** provided by distinct Linux kernel subsystems and tools, among which:

- **Namespaces**: for configuring the system resources that are visible by a given process (network interfaces/ports, users, PIDs, ...)
- **Control groups (“cgroups”)**: for enforcing resource allocation limits
- **Capabilities**: for controlling the operations that a given process/user is allowed to perform on various kinds of resources
- **Seccomp**: for filtering the legitimate system calls that a process can make

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).

OCI: Open Container Initiative (1/2)

- The OCI is (since 2015) “*an open governance structure for the express purpose of creating open industry standards around container formats and runtimes*”.
- So far, 3 main specifications have been produced (image, runtime, distribution).
- **Image format specification:**
 - “Defines the requirements for an OCI Image (container image), which consists of a manifest, an optional image index, a set of filesystem layers, and a configuration.”
- **Runtime specification:**
 - “Specifies the configuration, execution environment, and lifecycle of a container.” In particular, “defines how to properly run a container *filesystem bundle*” which fully adheres to the OCI Image Format Specification.”
- **Distribution specification:**
 - “Defines an API protocol to facilitate and standardize the distribution of content. It was launched in April 2018 to standardize container image distribution around the specification for the Docker Registry HTTP API V2 protocol, which supports the pushing and pulling of container images.”

OCI: Open Container Initiative (2/2)

- For more details:
 - I. Velichko. *What Is a Standard Container (2021 edition)*. <https://iximiuz.com/en/posts/oci-containers/>
 - B. Chen. *Open Container Initiative (OCI) Specifications*. 2019. <https://alibaba-cloud.medium.com/open-container-initiative-oci-specifications-375b96658f55>
 - A. Krajewska. *Container image formats under the hood*. 2020. <https://snyk.io/blog/container-image-formats/>
 - J. Webb. *Docker and the OCI container ecosystem*. LWN. 2022. <https://lwn.net/Articles/902049/>
 - OCI image spec: <https://github.com/opencontainers/image-spec/>
 - OCI runtime spec: <https://github.com/opencontainers/runtime-spec>
 - OCI distribution spec: <https://github.com/opencontainers/distribution-spec>

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.

For more details on the principles and origins of virtual machines & containers

- **On virtual machines:**

- A. Tanenbaum & H. Bos. *Modern operating systems (4th edition)*. Pearson education. 2014. Chapter 7: “Virtualization and the cloud”.
- E. Bugnion, J. Nieh, D. Tsafirir. *Hardware and Software Support for Virtualization*. Morgan & Claypool. 2017.

- **On containers:**

- A. El Amri. *The missing introduction to containerization*. 2019. <https://medium.com/faun/the-missing-introduction-to-containerization-de1fbb73efc5>
- I. Lewis. *Container runtime series (parts 1-4)*. 2017. <https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r>
- I. Velichko. *What Is a Standard Container (2021 edition)*. <https://iximiuz.com/en/posts/oci-containers/>
- E. Baker. *A comprehensive container runtime comparison*. 2020. <https://www.capitalone.com/tech/cloud/container-runtime/>
- J. Webb. *Docker and the OCI container ecosystem*. 2022. <https://lwn.net/Articles/902049/>
- A.Suda. *The internals and the latest trends of container runtimes*. 2023. <https://medium.com/nttlabs/the-internals-and-the-latest-trends-of-container-runtimes-2023-22aa111d7a93>

- **On both kinds of resources:**

- Allison Randal. *The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers*. ACM Computing Surveys. May 2020. <https://dl.acm.org/doi/abs/10.1145/3365199>

“Lightweight” virtual machines (1/3)

- Containers have thrived because they are lightweight ... but they are more difficult to run securely than virtual machines.
- Recently, new virtualization software has emerged with the following goals:
 - Reaching resource footprints ~ similar to those of containers
 - Reaching startup/shutdown times ~ similar to those of containers
 - Providing good I/O performance (without per-VM dedicated I/O devices)
 - Retaining the stronger isolation properties of virtual machines

“Lightweight” virtual machines (2/3)

Such “lightweight” virtual machines can use either:

- A stripped-down version of **a traditional guest OS** like Linux
- A “**unikernel**” (a.k.a. “**library OS**”)
 - A specialized guest OS that is tightly linked with the application (single binary)
 - There is no run-time boundary between the app and the guest OS
 - Provides **better performance and lower resource footprint**
 - **May complicate some other aspects**: administration, debugging, compatibility with multiprocess applications, ...

“Lightweight” virtual machines (3/3)

Examples (open-source projects):

- **Firecracker VMM** by Amazon Web Services
 - Used by AWS for secure and efficient execution of lightweight/ephemeral services (e.g., AWS Lambda and AWS Fargate)
 - Based on a modified version of the KVM hypervisor + a replacement of the QEMU emulator + restricted/simplified device set/model + implementation in a safe language (Rust) + container technologies
 - <https://firecracker-microvm.github.io>
 - A. Agache et al. Firecracker: Lightweight virtualization for serverless applications. In Proceedings of NSDI 2020. <https://www.usenix.org/conference/nsdi20/presentation/agache>
- **Kata containers** by Intel
 - <https://katacontainers.io>
- **LightVM project** (research prototype)
 - <http://sysml.neclab.eu/projects/lightvm/>
 - Research article: F. Manco et al. “My VM is faster (and safer) than your container”. In Proceedings on the 2017 ACM Symposium on Operating System Principles.

Unikernels

For more details, see the following references:

- <http://unikernel.org>
- A. Madhavapeddy and D. Scott. Unikernels: The rise of the virtual library operating system. *Communications of the ACM*. January 2014.
- A. Kivity et al. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*.
- Bryan Cantrill. Unikernels are unfit for production. January 2016. <https://www.joyent.com/blog/unikernels-are-unfit-for-production>
- S. Kuenzer. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of EuroSys 2021*. <https://unikraft.org>
- A. Raza et al. Unikernel Linux (UKL). *Proceedings of EuroSys 2023*. <https://dspace.mit.edu/bitstream/handle/1721.1/150839/3552326.3587458.pdf>

Physical machines (1/2)

- Also known as “**bare-metal instances**” or “**hardware as a service (HaaS)**”
- **Interface:**
 - a raw, dedicated physical machine, on which a user can freely deployed the host OS of his choice
 - ... through remote management software interacting with the machine’s firmware and BMC (baseboard management controller)
- **Popular in various contexts:**
 - Private clouds
 - High-Performance Computing (HPC) centers
 - Research & teaching testbeds
 - Also, more recently, in public clouds. For example:
 - AWS EC2 bare metal instances
 - Packet.com (specialized provider for bare metal cloud resources)

Physical machines (2/2)

- **Allow full machine reservation and reinstallation at small time scales.**
 - OS image installation/deployment time ~ a few minutes
 - Flexible reservations times (from minutes to days)
 - Public clouds support per-second billing
- **Useful for various needs:**
 - Benefits stemming from the **physical isolation of the machine** (removing potential interferences caused by multi-tenancy):
 - Improved security
 - Improved performance
 - Improved predictability (e.g., important for research and real-time workloads)
 - Also, benefits due to the **removal of virtualization**:
 - May have significant positive impact on some workloads (e.g., HPC, network intensive)
 - May enable/accelerate guest-level virtualization (removing the need for nested virtualization)
 - May enable support for non-virtualized hardware accelerators (e.g., GPUs)

Virtual & physical machines – Offerings in the public clouds

- **A large & growing diversity of hardware configurations**
 - Several main categories (with sub-categories and sub-parameters in each)
 - General purpose
 - Compute optimized
 - Memory optimized
 - Storage optimized
 - Accelerated computing
 - Custom configurations
 - Fixed CPU resources vs. burstable
- **Different pricing/reservation models**
 - On demand instances
 - Reserved instances
 - Spot/preemptible instances

Outline

- General-purpose computing
- **Storage**
 - Block storage
 - File storage
 - Object storage
 - Database storage
- Networking

Block storage

- Examples:
 - AWS Elastic Block Store, Google Persistent Disk
 - OpenStack Cinder Block Service
- Provides a block device abstraction (**akin to a hard disk in a single computer**).
- This is the **lowest abstraction level**. A block device is typically formatted and managed by a (local) file system.
- Such a block device:
 - Is accessed remotely (over the network), in the same datacenter
 - Can only be attached to a single virtual machine at a time
 - Generally supports snapshots
- **Use cases:** system volumes, data volumes (e.g., for databases)

File storage

- Examples
 - AWS Elastic File Service, Google Filestore
 - Network File System (NFSv3, NFSv4)
 - HDFS, Lustre
- Unlike a local file system stored on a block device, this abstraction **allows several client machines to simultaneously have access to the same file system**
- Typically supports a locking service to manage concurrent file accesses
- Various kinds of file systems are available
 - Various consistency and fault tolerance semantics
 - General purpose vs. optimized for specific types of workloads (e.g., big data, HPC)
- **Warning:** Completely different requirements than personal cloud storage systems (e.g., Dropbox, Google Drive)

Object storage

- (Sometimes also called “blob storage”)
- **Examples:**
 - Cloud services: AWS S3 (“Simple Storage Service), Google Cloud Storage
 - Open-source frameworks: OpenStack Swift, MinIO
- **Main goals:**
 - Scalability (number of data items, size of data items, number of clients)
 - Fault tolerance, durability and high availability (through redundancy)
 - Simplicity (simple to use & to scale)
 - Genericity (very diverse use cases: web sites/applications, multimedia, system images, backups, data lakes, analytics ...)
 - Cost effectiveness
 - **Typically available in the form of several storage “classes”**, corresponding to various requirements in terms of performance, durability, availability, cost.

Object storage – Main characteristics (1/2)

- Relies on the notion of "object"
 - Somewhat different from other traditional abstractions (block, file, database tuple).
 - Some similarities with the concept of key-value database (yet with some differences in features).
 - **An object is a generic container of bytes, with a variable size (potentially very big)**
 - **Versatile support for very different object sizes (from bytes to several TBs)**
 - Each object is also associated with a few kBs of metadata stored in a key-value format. Some of these metadata are system defined (e.g., last modification date). Others are flexible (user defined).

Object storage – Main characteristics (2/2)

- **Loosely structured namespace**

- Each object is identified by a globally unique, user-defined key (text string of variable length)
- Each object belongs to a given storage container named “bucket”
- The bucket namespace is flat (i.e., there is no hierarchy, unlike in a file system)

- **Optional features:**

- Versioning (per object)
- Access logging (per-object)
- Access control lists (typically per-bucket)
- Public access from the Internet, via a URL

Object storage – Interface (1/4)

- Main interface: accessible via a "Web service" interface (**HTTP-based, REST API**)
- (A client application can also mount a bucket as a file system ... but not fully compliant with POSIX file system semantics, and also some features of object storage will not be available)
- Main operations (for buckets, objects and metadata) : **"CRUD"**
 - Create, Read, Update, Delete
 - In the case of objects, "Update" = full overwrite
- Each object is uniquely identified with the combination of bucket name + key (+ optional version ID)
- Although a bucket is flat namespace, it is somehow possible to "recreate the notion of folder" by using of a common prefix in the name of various objects (e.g., "logs/")
- Data at rest can be encrypted on the client side or on the server side

Object storage – Interface (2/4)

- **Other operations:**

- Partial read of an object
- Conditional read of an object (if-modified-since, if-unmodified-since ...)
- Copy of an object
- Delete multiple objects at once
- Set object/bucket in “lock mode”, preventing future deletion or update (e.g., to enforce a given retention threshold)

Object storage – Interface (3/4)

- The AWS S3 API (although not standardized) has become a de-facto standard, implemented by most systems
 - Note however that, although this facilitates porting applications from one system to another, object storage systems are often not fully compatible and thus not always easily interchangeable, because advanced details/features have significant differences.
 - For example, see the following comparison between AWS S3 and Google Cloud Storage: <https://www.zenko.io/blog/four-differences-google-amazon-s3-api/>

Object storage – Interface (4/4)

- **Additional details** - For most object storage systems:
 - Updates to a single object are guaranteed to be atomic (no partial writes)
 - ... However, they may not become immediately visible (see description of consistency semantics on the next slides)
 - There are no atomicity/transactional guarantees across several objects
 - The system does not provide any object locking facility to manage concurrent requests.
 - Note that, given the weak consistency model provided by most object storage systems, it is generally not possible/practical to build a locking service on top of them.
 - Instead, if a locking service is needed, it is more appropriate to use an external locking service.

Object storage – Consistency (1/4)

- **Some Cloud-based object storage systems only provide limited/weak consistency guarantees.** This means, for example, that, in some situations, a read request may return an outdated view of the data.
- This is due to designed choices related to the management of replication, and the emphasis given to scalability.
- However, consistency guarantees of object storage are being improved (i.e., becoming stronger) due to the growing popularity and diversity of use cases for object storage, and the productivity/correctness benefits for application developers.

Object storage – Consistency (2/4)

- Terminology
 - **“Eventual consistency”** corresponds to a weak level of consistency. It only provides the following guarantees:
 - An attempt to read an existing object may return the current value of the object, or an older version, or NULL.
 - If we stop updating the object, all its copies (“replicas”) will eventually converge to the same state, after a finite amount of time (but we do not know the precise upper bound for this time interval).
 - In contrast, **“read-after-write”** semantics provide stronger consistency guarantees.

Object storage – Consistency (3/4)

- Case study #1: AWS S3 – Before December 2020
 - S3 generally provides read-after-write consistency
 - ... except in the following situations, which only guarantee eventual consistency:
 - Modification of an existing object (overwrite or delete) followed by a read
 - Attempting to read an object that does not exist, followed by a creation of the object, followed by a read of the object

Illustration (quotes from the old version of the S3 official documentation):

- *“S3 achieves high availability by replicating data across multiple servers within Amazon’s data centers. If a PUT request is successful, your data is safely stored. **However, information about the changes must replicate across S3, which can take some time, and so you might observe the following behaviors:***
 - *A process writes a new object to S3 and immediately lists keys within its bucket. Until the change is fully propagated, the object might not appear in the list.*
 - *A process replaces an existing object and immediately attempts to read it. Until the change is fully propagated, S3 might return the prior data.*
 - *A process deletes an existing object and immediately attempts to read it. Until the deletion is fully propagated, S3 might return the deleted data.*
 - *A process deletes an existing object and immediately lists keys within its bucket. Until the deletion is fully propagated, S3 might list the deleted object.”*

Object storage – Consistency (4/4)

- Case study #2: AWS S3 – Since December 2020

- “all S3 GET, PUT, and LIST operations, as well as operations that change object tags, ACLs, or metadata, are now strongly consistent. What you write is what you will read, and the results of a LIST will be an accurate reflection of what’s in the bucket.”

Illustration (quotes from the current version of the S3 official documentation):

- *“S3 achieves high availability by replicating data across multiple servers within Amazon’s data centers. If a PUT request is successful, your data is safely stored. **Any read (GET or LIST request) that is initiated following the receipt of a successful PUT response will return the data written by the PUT request.** Here are examples of this behavior:*
 - *A process writes a new object to Amazon S3 and immediately lists keys within its bucket. The new object appears in the list.*
 - *A process replaces an existing object and immediately tries to read it. Amazon S3 returns the new data.*
 - *A process deletes an existing object and immediately tries to read it. Amazon S3 does not return any data because the object has been deleted.*
 - *A process deletes an existing object and immediately lists keys within its bucket. The object does not appear in the listing.”*

- For more details:

- <https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel>
- <https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html>
- <https://blog.kylegalbraith.com/2021/01/12/the-s3-consistency-model-got-an-upgrade/>

Object storage – Wrap-up

- **Main strengths**

- Simplicity
- Versatility (use cases, scale, guarantees, access model ...)
- Cost

- **Main weaknesses**

- Some systems **only provide limited guarantees regarding data consistency and the management of concurrent operations**. This may not be appropriate for certain kinds of applications, in particular with the following characteristics:
 - Frequently modified objects and freshness requirements
 - Concurrent read-write accesses to the same objects
- Poorer performance (w.r.t. latency and bandwidth) compared to some other types of storage services (especially lower-level and/or more closely-coupled interfaces like block storage)

Databases (1/3)

- There are many types of databases for cloud-based systems
- **Main aspects to be considered:**
 - Managed (by the Cloud provider) or not
 - Data model (see details on the next slides)
 - Transactional features or not
 - Scalability (volume of data, number of requests) & sharding
 - Replication & fault-tolerance
 - Geographic scale (single datacenter vs. geo-replication)
 - Design trade-offs in terms of consistency, performance and availability

Databases (2/3)

There is a **growing diversity of database features**, due to :

- A growing number of data-intensive applications, with diverse requirements.
- A growing resort to storing derived data (i.e., multiple views/formats for the same initial piece of information), in order to facilitate/accelerate some types of data analysis.
- The rise of software architectures based on “micro-services” (a software design pattern that we will study in another lecture), which fosters the use of many small components and encourages the decentralization of the application data into several specialized datastores throughout a software system.

Databases (3/3)

We will give an overview of **several popular data models**:

- Key-Value
- Document
- Relational
- NewSQL
- Graph
- Column family
- Time series
- Streams and queues

Databases – Key/value

- Roughly speaking, akin to a very large and persistent hash table.
- The value is an opaque type.
- There is no schema.
- In some databases, the key can comprise multiple elements and support lookups based on the key prefix.
- Very basic but also usually very fast and scalable.
- Various goals:
 - (1) Many K/V databases are optimized for efficiently storing large volumes of data.
 - (2) Some K/V databases are specialized for storing important configuration data (metadata) and provide strong consistency and fault tolerance properties.
- Examples:
 - (1) Redis, Riak, AWS DynamoDB
 - (2) Etcd, Consul

Databases – Document

- Like a key/value DB, a document DB associates a primary key with a value. But the value must conform to a predefined format.
- This provides the ability to index the data items and to issue queries based on their values.
- However, a document DB does not rely on a schema. Each document can possibly have a different structure.
- A popular format for the document is JSON. Many documents are a composition of hashmaps (JSON objects) and lists (JSON arrays).
- Other popular formats are XML and BSON.
- Examples :
 - MongoDB, CouchBase
 - AWS DocumentDB, Google Cloud DataStore/FireStore

Databases – Relational

- Well-known, traditional model for SQL-based DBs.
- Data organized into two dimensional structures called tables (consisting of columns and rows).
- The inserted data items (“tuples”) must conform to the predefined schema of each table (number of columns and corresponding types).
- Most relational DBs support ACID transactions.
- Examples :
 - PostgreSQL, MySQL, MariaDB
 - AWS RDS, Google Cloud SQL (managed versions of the above ones)

Databases – NewSQL

- A new, emerging kind of relational databases.
- Support the main traditional features of relational DBs, like SQL queries and ACID transactions (sometimes with restrictions/variants).
- Are **specifically designed to provide high performance and/or high availability despite massive data scale and/or large geographic scale** (distribution/replication over multiple data centers).
- Examples :
 - CockroachDB
 - Google Cloud Spanner, Microsoft Azure CosmosDB

Databases – Graph

- The data model is based on two types of information: nodes (~ entities) and edges (relationships between nodes). Specific properties can be attached to each edge or node.
- Useful data model for analyses based on the relationships between data items.
- The storage format of such DBs is optimized for fast graph traversal (potentially much more efficient than joins between tables in a relational DB).
- Examples :
 - Neo4j, Dgraph
 - AWS Neptune

Databases – Column family

- Organizes data into rows and columns, like a relational DB. But :
 - Instead of tables, the DB uses structures named “column families”.
 - Each row defines its own format.
 - Rows can be sparse.
- Can also be described as a special key/value DB in which each key (row identifier) is associated with a dictionary of arbitrary attributes and their values.
- Good/useful for applications requiring great performance for row-based operations. (In contrast to joins or aggregations.)
- Examples :
 - Cassandra, HBase
 - AWS Redshift, Google Cloud BigTable

Databases – Time series

- A time series DB is optimized to store records based on time (e.g., telemetry data).
- Records are often small but plentiful.
- Such a DB often must support a very high number of write operations in real time (insertion of new records).
- Updates are rare, and deletions are performed in bulk.
- Such a DB often provides flexible support for data retention and down-sampling.
- Examples :
 - InfluxDB, Graphite
 - AWS Timestream

Databases – Streams & queues

- Store events and messages.
- There are significant differences between them.
- **Streams**
 - Data is stored as an immutable stream of events.
 - Typically used for data integration across heterogeneous services. Multiple services can consume data from the same stream without interfering on each other.
 - Example: Kafka
- **Queues**
 - Store messages/events that can be changed or removed.
 - Typically used for communication between services (e.g., dispatching requests and replies).
 - Example: RabbitMQ

Databases – For further details

- Prisma blog. Comparing Database Types: How Database Types Evolved to Meet Different Needs. <https://www.prisma.io/blog/comparison-of-database-models-1iz9u29nwn37>
- Martin Kleppmann. Designing Data-Intensive Applications. O'Reilly, 2017. (Chapters 2, 3, 4, 6)
- B. Scholl, T. Swanson, P. Jausovec. Cloud native: Using containers, functions, and data to build next generation applications. O'Reilly, 2019. Chapter 4: Working with data.

Outline

- General-purpose computing
- Storage
- **Networking**

Networking

Typical features/services include:

- Virtual Private Cloud (VPC):
 - IP subnet topology/setup
 - Virtual LANs (VLANs)
 - Network Address Translation (NAT) Gateways
 - Virtual Private Networks (VPNs)
 - Firewalls
 - Routers
- Domain Name System (DNS)
- Content delivery networks (CDNs)
- Application-level load balancers
- API gateways