UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Advanced Cloud Computing

Lecturer:
Prof. Ruggero Lot

Author:
Andrea Spinelli

January 15, 2025

github.com/Spina02 andreaspinelli2002@gmail.com

Abstract

As a student of Scientific and Data Intensive Computing, I've created these notes while attending the **Advanced Cloud Computing** - module of *HPC and Cloud Computing* course - to deepen my understanding of modern cloud infrastructure. This document represents my journey through the advanced concepts of cloud computing, focusing particularly on containerization and orchestration at scale.

The course material I've documented is structured around two main sections:

- **Foundation Technologies:** My exploration into the core technologies that make containerization possible, including Linux kernel namespaces, resource limitations, networking, and layered filesystems. Through hands-on experiments, I've worked to understand how these components create isolated environments for applications.
- **Orchestration with Kubernetes:** My practical experience with Kubernetes, documenting the essential resources needed for deploying applications at scale. I've covered topics from basic concepts like replica management to more complex subjects such as deployment strategies, autoscaling, and pod networking with various CNI solutions like Calico, Flannel, and Cilium.

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in cloud computing.

The content focuses on both theoretical foundations and practical implementations, documenting the challenges and solutions I've encountered while learning these advanced concepts. My goal in sharing these notes is to provide a practical perspective on cloud computing from a student's point of view, hopefully making these complex topics more approachable for others who are on a similar learning path.

Contents

1 Networking Foundations

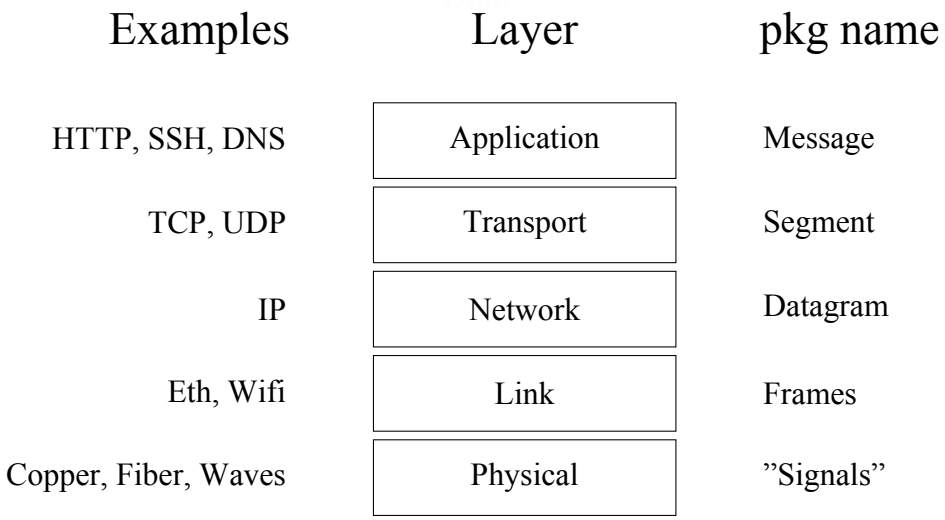
1.1 Introduction to Networking

Networking is the practice to connect multiple devices together to share resources and informations. It is the foundation of the internet and also the backbone of cloud computing. When a device needs to communicate with another one it splits data into packets with an header and sends them over the network. The packets are then reassembled at the destination device. This process is called packet switching.

1.1.1 The 5-Layer Network Model

The network model is characterized by layered architecture which splits networking into different levels of abstraction, each one with a specific purpose. This architecture is called the network model and it is used to standardize network communications.

The **OSI network model** is the most common model used in networking and it is composed by 7 layers. The **5-layer model** is a simplified version and it is composed by the following layers:



Each layer in the network model combines software and hardware differently. Application and Transport layers run primarily as software on end devices and servers, enabling flexible protocol updates. The Network layer uses a hybrid approach with software routing and hardware-accelerated forwarding. Physical and Data Link layers are mainly hardware-based, implemented in NICs and transmission media for optimal performance.

While this layered architecture promotes modularity and easier maintenance, it can introduce overhead due to data encapsulation and occasional duplication of functionality across layers. Some modern approaches, like Software-Defined Networking (SDN), attempt to optimize these trade-offs by allowing more flexible layer interactions.

Physical Layer (L1)

The physical layer is the lowest layer of the network model and it is responsible for the physical connection between devices. It defines the hardware and the physical medium used to transmit data. The physical layer is responsible for the transmission of raw data bits over a physical medium.

Data Link Layer (L2)

The data link layer is responsible for reliable point-to-point delivery of data frames between directly connected nodes. It handles addressing using MAC (Media Access Control) addresses, error detection, and flow control. This layer breaks data into frames and ensures reliable transmission over a single network segment. Common protocols at this layer include Ethernet for wired networks and Wi-Fi (802.11) for wireless networks.

The data link layer also manages access to the shared physical medium through protocols like CSMA/CD (Carrier Sense Multiple Access with Collision Detection) for Ethernet or CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) for wireless networks. For more information on MAC addressing, refer to Section 1.2.1.

Network Layer (L3)

The network layer manages the routing and forwarding of data across the network. It ensures that packets are sent from the source to the destination, potentially through multiple routers. The network layer uses IP addresses to determine the best path for packet delivery. For a more detailed analysis of IP addressing, refer to Section 1.2.1.

Transport Layer (L4)

The **transport layer** is responsible for end-to-end communication between applications. It includes two primary protocols:

- **UDP** (*User Datagram Protocol*): A connectionless, unreliable protocol typically used for applications requiring fast, continuous data transmission, such as audio and video streaming. It does not guarantee delivery or order but is faster than TCP.

  UDP uses only the **destination IP** and **destination port**. It is simpler than TCP, as it does not establish a persistent connection. This makes UDP faster but less reliable, with no need for tracking the source address once the data is sent.

- **TCP** (*Transmission Control Protocol*): A connection-oriented, reliable protocol used for most types of data transfer. It ensures data is delivered in order and without errors, making it suitable for tasks like file transfers and web browsing.

  TCP requires more detailed parameters, including both the **source IP** and **source port** in addition to the **destination IP** and **destination port**. These allow for two-way communication and reliable delivery of data, as TCP establishes and maintains a connection between sender and receiver.

Application Layer (L5)

The application layer is the top layer in the network protocol stack, where user-facing applications interact with the network. It provides protocols, such as HTTP, SMTP, and FTP, to facilitate

services like web browsing, email, and file transfer. This layer handles high-level communication between software applications and enables them to exchange data over a network.

The **Domain Name System** (DNS) translates human-readable hostnames (e.g. www.example.com) into IP addresses that computers use to identify each other on the network. It acts as the internet's phonebook, ensuring users can access websites and services without needing to remember numerical addresses. A **socket** is an endpoint for communication between two machines over a network. It combines an IP address and a port number to establish a connection, allowing applications to send and receive data.

Socket creation: Client

```
# Socket creation
clientSocket = Socket(AF_INET, SOCK_STREAM)
# socket connection opening
clientSocket.connect((server_name, server_port))
```

Socket creation: Server

```
# Socket creation
clientSocket = Socket(AF_INET, SOCK_STREAM)
# socket start listening
clientSocket.bind(('', server_port))
clientSocket.listen(1)
```


1.1.2 Data Encapsulation

Data encapsulation is the process of wrapping data with necessary protocol information before network transmission. Each layer of the network model adds its own **header** (and sometimes **trailer**) to the data from the layer above, creating a packet that can be properly routed and delivered (Figure 1.1). This encapsulation ensures that data is transmitted accurately and efficiently across the network, with each layer handling specific aspects of the communication process. The actual data in a packet is called **payload**.

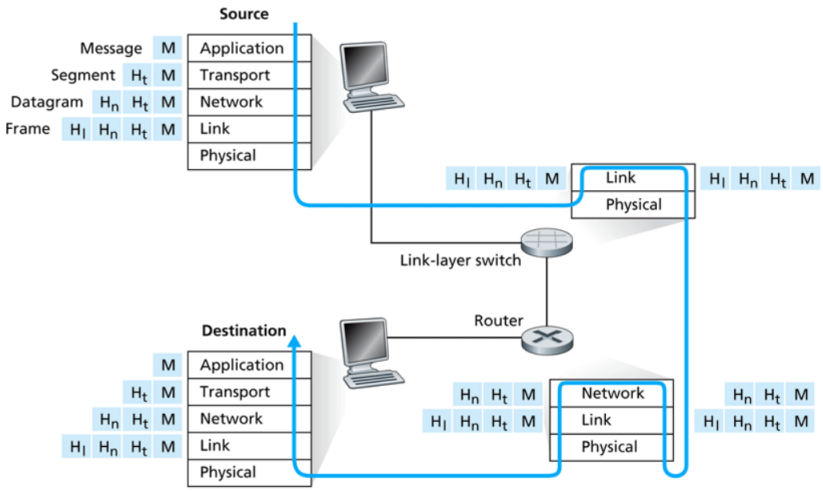


Figure 1.1: Encapsulation path

Each layer's headers (and trailers) contain data specific to the protocol operating at that layer. For example, the network layer header includes source and destination IP addresses, while the transport layer header includes source and destination port numbers. They can also include error detection bits that allow the receiver to determine if any bits in the message have changed during transit. This layered approach allows each protocol to focus on its specific function, ensuring modularity and ease of troubleshooting.

1.2 Network Addressing

Efficient network communication relies on a robust addressing mechanism to ensure data packets are delivered to their intended destinations. The process begins with setting the correct destination address, followed by forwarding the packet through routers using MAC addresses at Layer 2. Routers, equipped with forwarding tables, determine the optimal path for each packet, mapping destination addresses to specific link interfaces.

In layer 3, the Internet model adopts a "best-effort" delivery approach. This design does not guarantee timing, order, or delivery of packets, but it prioritizes flexibility and scalability across vast and interconnected networks. Additional controls, such as ensuring reliability and sequencing, are delegated to higher layers within the protocol stack, allowing the network layer to focus on efficient routing and addressing.

1.2.1 MAC and IP Addressing

MAC addressing

A MAC address is a unique identifier for each network interface on a device. These addresses operate at the link layer (L2), enabling local communication between devices. The address FF:FF:FF:FF:FF:FF is used for broadcasting to all devices on a network.

The Address Resolution Protocol (ARP) is used to map an IP address to a MAC address on a local network. When a device needs to communicate with another device within the same network segment, it sends an ARP request to determine the corresponding MAC address.

IP Addressing

The **Internet Protocol (IP)** is a fundamental protocol within the Internet protocol suite, responsible for addressing and routing packets of data so they can traverse interconnected networks and reach their intended destinations. Operating at the network layer (L3), IP ensures that data sent from a source device is directed toward the correct recipient through various networking devices such as routers and switches.

The IPv4 datagram (Figure 1.2) includes important fields like the version, header length, flags for fragmentation, source and destination IP addresses, and other optional fields. It allows for routing and addressing across the network. Key fields include:

- Version (IPv4 or IPv6).
- Flags for managing fragmentation.
- Source and destination IPs that identify the sender and receiver.
- Upper-layer protocol (such as TCP or UDP) used to transport the data.

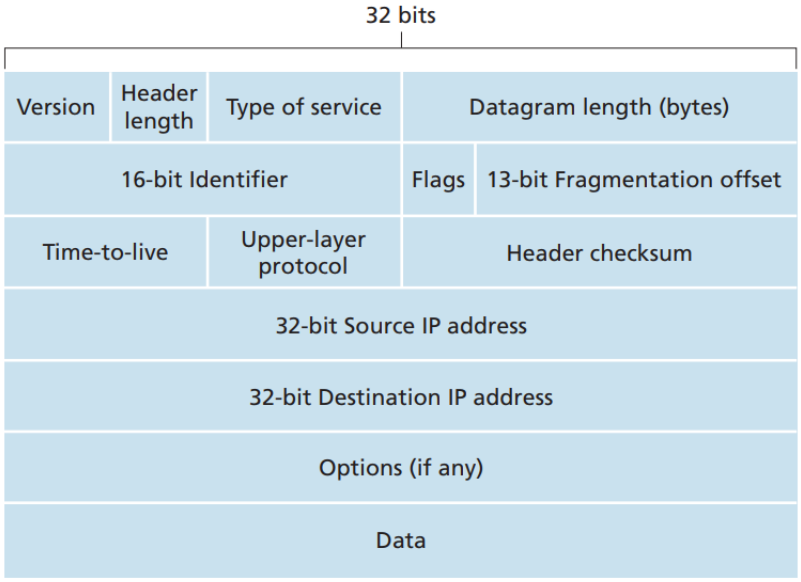


Figure 1.2: IPv4 datagram structure

An IPv4 address is a 32-bit number divided into four octets, each representing a decimal number (e.g., 172.16.254.1). The address is used to uniquely identify a device on a network, typically paired with a subnet mask (e.g., /24) to define the network range.

Observation: *IPv6*

While these notes focus on IPv4 addressing and datagrams, it is important to acknowledge the growing significance of **IPv6** in modern networking. IPv6, with its 128-bit address space, was designed to address the limitations of IPv4, particularly the exhaustion of available addresses. Although not covered here, further exploration of IPv6 is recommended for a comprehensive understanding of contemporary network protocols.

1.2.2 Hostnames and DNS

Hostnames are human-readable labels assigned to devices on a network, facilitating easier identification and access compared to numerical IP addresses. For example, `www.example.com` is more memorable than its corresponding IP address.

The **Domain Name System (DNS)** is a hierarchical service that translates hostnames into IP addresses, enabling users to access resources using familiar names. When a user enters a hostname into a browser, a DNS query resolves it to the appropriate IP address, allowing the connection to be established seamlessly. DNS can be vulnerable to attacks like DNS spoofing and cache poisoning. Security measures like DNSSEC help ensure DNS response integrity and authenticity.

1.2.3 Network Ports

Network ports are logical endpoints used by the Transport Layer protocols (TCP and UDP) to differentiate between multiple services running on a single device. Each port is identified by a unique number ranging from 0 to 65535.

Well-known ports (0-1023) are reserved for common services such as HTTP (`80`), HTTPS (`443`),

and SSH (`22`). Registered ports (1024-49151) are assigned to user applications and services, while dynamic or private ports (49152-65535) are typically used for temporary communications initiated by client applications.

Proper management of network ports is crucial for both functionality and security. Open ports can be potential entry points for unauthorized access, making it important to monitor and control port usage through firewalls and security policies.

Warning: *Port Security*

Unsecured open ports can be exploited by malicious actors to gain unauthorized access or disrupt services. Regularly auditing open ports and implementing strict firewall rules are essential practices to protect network integrity.

1.2.4 Core Network Services

CIDR

Classless Inter-Domain Routing (CIDR) notation enhances IP address allocation by using the format `a.b.c.d/x`, where `x` indicates the number of bits in the network prefix. Introduced to improve the scalability of IP addressing, CIDR allows for more flexible and efficient use of IP address spaces compared to the traditional class-based system. By aggregating IP addresses into variable-length blocks, CIDR reduces the size of routing tables and optimizes routing efficiency.

Example: *CIDR Notation Example*

Consider the network 192.168.1.0/24:

- Network address: 192.168.1.0
- Subnet mask: 255.255.255.0
- First usable IP: 192.168.1.1
- Last usable IP: 192.168.1.254
- Broadcast address: 192.168.1.255
- Total IPs: 256 (254 usable)

Another example with 192.168.0.0/16:

- Network range: 192.168.0.0 to 192.168.255.255
- Subnet mask: 255.255.0.0
- Total IPs: 65,536 (65,534 usable)

NAT

Network Address Translation (NAT) enables multiple devices on a local network to share a single public IP address. By translating private IP addresses to a public address using a translation table, NAT conserves the limited pool of public IP addresses and provides a layer of security by obscuring internal network structures from external entities.

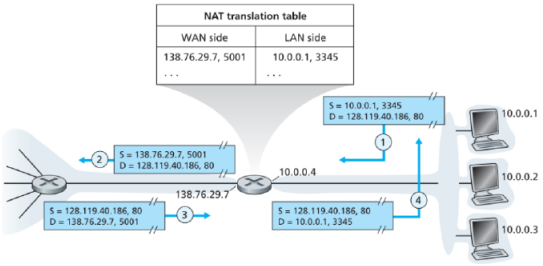


Figure 1.3: Network Address Translation (NAT) Process

Observation: *NAT and IPv6*

While NAT has been instrumental in conserving IPv4 addresses, the adoption of **IPv6** eliminates the need for NAT by providing a vast address space. IPv6 allows for direct end-to-end connectivity, simplifying network configurations and improving performance.

DHCP

Dynamic Host Configuration Protocol (DHCP) automates the assignment of IP addresses and other network configurations to devices on a network. As shown in Figure 1.4 when a device connects, it sends a DHCP request, and a DHCP server responds with an available IP address, subnet mask, default gateway, and DNS server information.

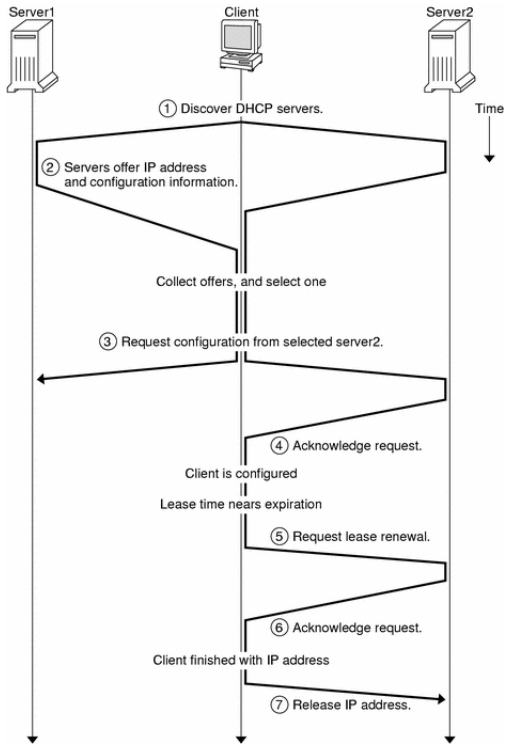


Figure 1.4: DHCP Process Flow

DHCP simplifies network management by dynamically allocating IP addresses, reducing the likelihood of address conflicts, and allowing for efficient utilization of IP address pools.

2 Virtualization and Isolation

2.1 Linux Namespaces

Linux Namespaces enable lightweight isolation by creating separate views of system resources for groups of processes. This capability underpins modern containerization, ensuring that processes running in one namespace remain unaware of those in another. By confining access to system resources, namespaces allow multiple containerized workloads to share a single kernel while maintaining logical separation.

Process ID (PID)

The PID namespace gives each container its own process numbering scheme, starting with PID 1. Within this namespace, processes are invisible to those in the host system or other namespaces. This setup allows containers to manage processes as though they are on a dedicated machine, enabling features like container migration or suspension without affecting the global PID space.

UNIX Time-sharing System (UTS)

The UTS namespace isolates hostname and domain name information, letting containers adopt distinct network identities. Updating the hostname inside one UTS namespace will not affect the host or other containers, ensuring clarity in multi-tenant or multi-instance environments.

Inter-Process Communication (IPC)

The IPC namespace confines shared memory regions, semaphores, and message queues to the container, preventing them from overlapping with the host or other containers. This mechanism is essential for complex applications that rely on inter-process communication but also need to remain isolated from external processes.

TIME

The Time namespace allows processes to operate under a virtualized system clock. This feature is particularly valuable in scenarios where an application requires time-dependent testing or simulation without altering the host system's actual clock. Processes within this namespace perceive a separate notion of system time, enabling careful control of time-related behaviors.

Mount (MNT)

The Mount namespace enables containers to have distinct filesystem views. Changes to mounts inside the namespace, such as mounting or unmounting filesystems, do not propagate to the host or other containers. This strategy is vital for secure sandboxing, as it prevents containers from altering critical host filesystem paths.

Control Groups (CGROUPS) Control Groups (`cgroups`) work in tandem with namespaces to manage resource usage at a granular level. Administrators or container engines can specify memory, CPU, or I/O constraints on groups of processes, ensuring that rogue or heavily loaded containers cannot starve other processes on the system.

Definition: *Linux Namespace*

A Linux Namespace is a kernel-level isolation mechanism that partitions process views of system resources (e.g., PIDs, networks, filesystems), providing each namespace with a restricted environment that does not interfere with other namespaces or the host system.

2.2 The /proc Filesystem and Resource Isolation

The /proc Filesystem

The `/proc` filesystem acts as a dynamic interface to the kernel's internal data structures. It provides real-time information about running processes, CPU features, and memory statistics. Containers typically rely on this pseudo-filesystem to query system resources and modify runtime parameters. Since each container can mount a tailored view of `/proc`, they perceive only the resources and processes available in their respective namespaces rather than those of the host.

Resource Isolation

Comprehensive resource isolation in Linux emerges from the combination of namespaces and `cgroups`. Namespaces enforce logical isolation, so that filesystem operations and process visibility remain confined to each container. `cgroups` impose physical limits, preventing containers from consuming excessive CPU time, memory, or other resources. This synergy enables secure, scalable environments capable of running numerous containers side by side without significant interference.

2.3 Virtual Machines and Hypervisors

2.3.1 Virtual Machines (VMs)

Virtual Machines (VMs) allow multiple operating systems to run in complete isolation on a single physical machine by emulating hardware through software. Each VM includes its own kernel, resource management, and filesystem, behaving as if it were a standalone system. This design provides robust security boundaries and reliability, making VMs attractive for scenarios requiring strong isolation, such as running untrusted code or simulating entire environments for testing.

However, VMs often incur higher resource usage and slower startup times when compared to container-based solutions. Because each VM replicates an entire operating system, they demand more memory and processing overhead, which can limit efficiency in environments where large numbers of instances need to run concurrently.

Definition: *Virtual Machine*

A Virtual Machine is a software-based emulation of a physical computer, featuring its own virtualized hardware, kernel, and operating system. This approach provides strong isolation between different workloads on the same host but can lead to increased resource consumption due to the replication of the underlying OS layers.

2.3.2 Hypervisors (KVM, QEMU)

Hypervisors form the foundation of virtualization by allocating and managing the hardware resources that VMs consume. They serve as an abstraction layer between the guest operating systems and

the physical hardware, ensuring isolation and controlling how resources like CPU, memory, and storage are shared.

Definition: *Hypervisor*

A hypervisor is the software or firmware responsible for creating and running virtual machines, isolating their execution environments from each other and from the underlying hardware.

Modern Linux-based hypervisors include *KVM* (Kernel-based Virtual Machine), which takes advantage of hardware virtualization extensions (e.g., Intel VT-x, AMD-V) to provide efficient performance. Meanwhile, *QEMU* is a powerful software emulation tool often paired with KVM to handle additional device emulation tasks, enabling a versatile and configurable virtualization stack. Together, KVM and QEMU provide a solid environment for running multiple VMs on Linux systems without significant overhead.

2.4 Containers

Containers have revolutionized the way applications are developed, deployed, and managed by providing a lightweight and portable environment that encapsulates an application along with its dependencies. Unlike Virtual Machines (VMs), containers share the host operating system's kernel, enabling high efficiency and rapid deployment.

2.4.1 Container Definition

A **container** is the runtime instantiation of a *container image* from a *repository*. It operates as a standard Linux process, typically created using the `clone()` system call instead of `fork()` or `execvp()`. Containers are further isolated using Linux namespaces and control groups (`cgroups`), along with security mechanisms like SELinux or AppArmor, ensuring that each container remains isolated from others and the host system.

Definition: *Container*

A container is a lightweight and portable environment that encapsulates an application together with its necessary dependencies, all running atop the host operating system's kernel instead of a separate guest OS. This design allows for efficient resource utilization and rapid startup times, making containers ideal for microservices and cloud-native applications.

2.4.2 Container Image and Repository

A *container image* is a packaged set of file system layers and metadata that defines how to run a container. It is typically downloaded from a *registry server* and used locally as a mount point when starting containers. While the term *container image* is often used interchangeably with *repository*, it is important to distinguish that a repository comprises multiple image layers and associated metadata, such as `manifest.json`.

Definition: *Container Image*

A container image is a packaged set of file system layers and metadata describing how to run a container. It is typically downloaded from a remote registry and mounted locally to launch container processes.

Common image formats include *Docker*, *Appc*, *LXD*, and the *Open Container Initiative (OCI)* format, with OCI being the most widely adopted standard. Commands like `docker pull` or `podman pull` are used to retrieve images from repositories by referencing their repository names and optional tags.

Observation: *Repository vs. Image*

While *images* and *repositories* are often used interchangeably, it is crucial to understand their distinction. A **repository** is a collection of related image layers and metadata, identified by a unique name and optional tags. For example, the command `docker pull registry.access.redhat.com/rhel7:latest` fetches an image from the specified repository, which may contain multiple layers representing incremental changes.

2.4.3 Container Engines

A **container engine** is software that orchestrates the lifecycle of containers. It accepts user requests through interfaces like command-line tools, manages container images by pulling them from registries, and initiates or terminates containers using underlying *container runtimes*.

Definition: *Container Engine*

A container engine is a software layer that orchestrates container lifecycle operations, including image retrieval, container creation, and container management, typically leveraging an underlying container runtime for the actual process isolation.

Examples of container engines include *Docker*, *Podman*, *RKT*, *CRI-O*, and *LXD*. Cloud providers often implement their own container engines tailored to their platforms. For instance, Docker provides a comprehensive stack involving `docker-cli`, `docker-compose`, and the `dockerd` daemon, which interacts with `containerd` and `runc` to manage container processes.

2.4.4 Container Runtimes and System Calls

The **container runtime** is the low-level component within a container engine responsible for the creation and execution of container processes. Following the OCI Runtime Specification, `runc` serves as the reference implementation, while alternatives like `crun`, `railcar`, and `katacontainers` offer additional features or optimizations.

Definition: *Container Runtime*

A container runtime is the lower-level component within a container engine that manages the actual creation and execution of container processes. It interfaces directly with the operating system to apply namespaces, cgroups, and other isolation mechanisms.

Key *system calls* utilized by container runtimes include:

- `clone()`: Provides fine-grained control over what is shared between the parent and child processes, including namespaces.
- `fork()`: Duplicates the calling process to create a new process.
- `execvp()`: Replaces the current process image with a new one.

These system calls, combined with Linux features like `cgroups` and namespaces, form the foundation of container isolation and resource management.

2.4.5 Docker Stack vs. Podman Stack

Historically, **Docker** popularized containerization with a user-friendly CLI and a monolithic daemon (`dockerd`). The Docker architecture includes:

- `docker-cli` or `docker-compose`
- `dockerd` (container engine)
- `containerd` (manager)
- `containerd-shim`
- `runc` or a similar runtime

In contrast, **Podman** offers a daemonless experience, eliminating the need for a central background daemon like `dockerd`. Instead, `podman-cli` interacts directly with utilities such as `conmon` and `runc`, enabling rootless container execution without the overhead or security concerns associated with a central daemon.

2.4.6 Kubernetes Integration

For large-scale orchestration, **Kubernetes** utilizes the Container Runtime Interface (CRI) to interact with various container runtimes. While Docker was traditionally the default runtime, many Kubernetes deployments are transitioning to **CRI-O** or **containerd** to better align with the CRI specifications. Kubernetes leverages standard APIs to manage container deployment, networking, and scaling across multiple nodes, ensuring seamless operation within a cluster.

2.4.7 Capabilities and User Namespaces

Linux **capabilities** enhance security by dividing the traditional superuser privileges into distinct, manageable units (e.g., `CAP_NET_ADMIN`, `CAP_SYS_BOOT`). This allows containers to be granted only the specific capabilities they require, minimizing potential security risks.

Definition: *Capability*

A capability is a distinct unit of privilege that can be independently enabled or disabled for a process. This granularity allows for more secure and controlled permission management compared to the all-or-nothing superuser model.

User namespaces extend the capability concept by allowing a process to appear as root within the container while being an unprivileged user on the host system. This separation ensures that even if a process inside the container has root-like privileges, it does not pose a security threat to the host environment.

Capabilities are tied to user namespaces, meaning each namespace has its own set of capabilities. A child namespace cannot be granted more capabilities than the creating process possesses in its own namespace, ensuring controlled privilege escalation and enhancing overall security.

2.4.8 Mount Namespace Propagation

The **Mount namespace** plays a critical role in container security and filesystem management. It isolates mount points, ensuring that changes to the filesystem within a container do not affect the host or other containers. Linux defines several mountpoint propagation modes to control how mount events are shared across namespaces:

- `shared`: A mount belongs to a group of peers. Any changes propagate to all members.
- `slave`: One-way propagation where the master propagates changes to the slave, but not vice versa.
- `private`: Does not receive or forward any propagation events.
- `unbindable`: Cannot be bind-mounted.

These propagation modes provide flexibility for advanced scenarios, allowing certain filesystem changes to be visible across multiple namespaces while others remain strictly isolated.

2.5 Differences Between VMs and Containers

Virtual Machines (VMs) and Containers are both technologies that provide isolation and resource management, but they achieve these goals through fundamentally different approaches, resulting in distinct advantages and trade-offs.

Definition: ***Container***

A container is a lightweight and portable environment that encapsulates an application together with its necessary dependencies, all running atop the host operating system's kernel instead of a separate guest OS.

Virtual Machines (VMs) offer full hardware emulation, including their own operating systems and kernels. This allows VMs to provide strong isolation between different workloads, as each VM operates independently of the host and other VMs. VMs are ideal for scenarios requiring complete isolation and the ability to run different operating systems on the same physical hardware. However, this level of isolation comes at the cost of higher resource consumption and longer startup times, as each VM duplicates the overhead of an entire operating system.

Containers, in contrast, share the host operating system's kernel, providing a much more lightweight form of isolation. Containers encapsulate only the application and its dependencies, avoiding the need to replicate the entire OS. This results in significantly lower resource usage and faster startup times, making containers highly efficient for deploying microservices and cloud-native applications. Containers can be easily scaled and managed, fitting seamlessly into modern DevOps practices and continuous deployment pipelines.

Furthermore, containers offer greater portability across different environments, as they bundle all necessary components to run the application consistently from development to production. This consistency reduces the "it works on my machine" problem and streamlines the deployment process.

In summary, while **VMs** are suited for applications requiring full isolation and diverse operating systems, **containers** excel in environments demanding high efficiency, rapid scaling, and portability. The choice between VMs and containers depends on the specific requirements of the workload, balancing the need for isolation against resource efficiency and deployment flexibility.

Definition: *Virtual Machine vs. Container*

Virtual Machines provide complete hardware virtualization with their own operating systems, offering strong isolation but higher resource usage. Containers share the host OS kernel, delivering lightweight and efficient process-level isolation suitable for scalable and portable applications.

2.6 Mount Namespace Propagation

The Mount namespace is closely related to container security and convenience. It isolates mount points, meaning a container's filesystem changes do not affect the host or other containers. Linux also defines mountpoint propagation modes such as `shared`, `slave`, `private`, and `unbindable`, indicating how changes in one namespace might (or might not) be reflected in others. This flexibility supports advanced scenarios where certain filesystem changes must be visible across namespaces while others remain isolated.

2.7 Conclusion

By integrating these definitions and concepts from the slides, we can see how containers build on top of Linux kernel features like namespaces and cgroups to provide efficient and flexible process-level isolation. Container images simplify distribution by bundling application code and dependencies, while container engines and runtimes manage the lifecycle of these images as runnable processes. In contrast, Virtual Machines replicate entire operating systems, thereby delivering stronger hardware-level isolation but incurring heavier resource usage. Ultimately, the right choice depends on workload requirements: containers excel at high density and rapid deployment, whereas VMs may be more suitable where robust isolation or hardware emulation is paramount.