UNIT III

Virtualization: Approaches to Virtualization, Hypervisors, Virtualization to Cloud Computing, **Programming Models for Cloud Computing:** MapReduce, Cloud Haskell, Software Development in Cloud, Different Perspectives on SaaS Development, **New Programming Models Proposed for Cloud:** Orleans, BOOM and Bloom, Grid Batch, Simple API for Grid Applications.

Chapter Overview: Virtualization in Cloud Computing

Virtualization is a fundamental enabling technology for delivering various cloud computing services. It enhances **scalability**, **flexibility**, and **resource utilization** of the underlying infrastructure, while simplifying administrative tasks for IT personnel. Through **resource sharing**, virtualization also contributes to **green IT initiatives** by reducing energy consumption and optimizing hardware usage.

This chapter introduces the concept of virtualization and explains its benefits in the context of cloud computing. It discusses the different types of resources that can be virtualized, including **processor, memory, storage, and network resources**. Further, the chapter presents various approaches to virtualization, such as:

- **Full Virtualization**
- **Hardware-Assisted Virtualization**
- **Paravirtualization**

Additionally, the role of **hypervisors** (types, functions, and security concerns) is examined in detail. Toward the end, the chapter highlights the key differences between **cloud computing and virtualization**, and explains how virtualization serves as the backbone for cloud computing to deliver on-demand services effectively.

Virtualization -Introduction

1. Need for Virtualization

Modern computing environments require **large-scale infrastructure** to handle complex workloads. Traditionally, organizations purchased **new physical hardware** whenever extra resources were needed. However, this approach created problems:

- **High CapEx (Capital Expenditure):** Big upfront investments for servers, storage, and networking equipment.
- **High OpEx (Operational Expenditure):** Ongoing costs for power, cooling, maintenance, and management.
- **Low Resource Utilization:** A physical server might only use **10–20% of its capacity**, leaving most of the hardware idle.

- **Poor ROI (Return on Investment):** Huge spending but little benefit from underutilized systems.

As a result, organizations needed a solution that would **increase utilization, reduce cost, and improve flexibility**. This led to the widespread adoption of **virtualization**.

2. What is Virtualization?

Virtualization is a **technology that enables a single physical infrastructure to act like multiple independent logical systems**.

- It allows multiple **operating systems (OSs)** and applications to run on a **single physical machine**.
- A **software layer called Hypervisor (Virtual Machine Monitor)** manages the sharing of hardware resources among these virtual environments.
- Each virtual environment (called a **Virtual Machine – VM**) behaves like a real physical computer, with its own OS, applications, and storage.

👉 **Definition:** Virtualization is the process of creating a virtual version of computing resources (like servers, storage, memory, or networks) to enable resource sharing and better utilization.

3. Benefits of Virtualization

Virtualization provides several advantages:

1. **Efficient Resource Utilization** – Hardware resources are shared and used effectively.
2. **Cost Savings** – Reduces CapEx and OpEx by consolidating multiple servers into fewer physical machines.
3. **Increased ROI** – Better use of purchased infrastructure improves return on investment.
4. **Flexibility & Scalability** – Virtual machines can be created, modified, or deleted easily.
5. **Ease of Administration** – IT staff can manage multiple virtual systems from a central console.
6. **Green IT Support** – Fewer physical servers = reduced energy consumption, less space, lower carbon footprint.
7. **Improved Disaster Recovery** – Virtual machines can be backed up, cloned, or moved across systems quickly.

4. Types/Forms of Virtualization

Virtualization is not limited to hardware. Different forms include:

1. **Processor Virtualization** – Running multiple virtual CPUs on one physical CPU.
2. **Memory Virtualization** – Combining multiple memory resources into one pool, or allocating memory to VMs as needed.
3. **Storage Virtualization** – Abstracting physical storage (disks) into logical volumes for flexible usage.
4. **Network Virtualization** – Creating virtual networks (VLANs, VPNs) independent of physical hardware.

5. **Operating System Virtualization** – Running multiple isolated OS environments on the same hardware.
6. **Data Virtualization** – Allowing applications to access and process data without worrying about its physical location.
7. **Application Virtualization** – Running applications in virtualized containers independent of the underlying OS.

5. Before vs After Virtualization

Aspect	Before Virtualization	After Virtualization
Hardware Usage	One server → One OS + applications	One server → Multiple OSs + applications
Resource Utilization	Low (10–20%)	High (70–90%)
Costs (CapEx/OpEx)	Very High (new hardware for every need)	Lower (shared resources, fewer servers)
Scalability	Slow – need to buy and install new hardware	Fast – create new virtual machines easily
Experimentation/Testing	Requires separate hardware for each test	Multiple test environments on the same machine
Green IT	More power, space, cooling required	Less power and space, energy-efficient

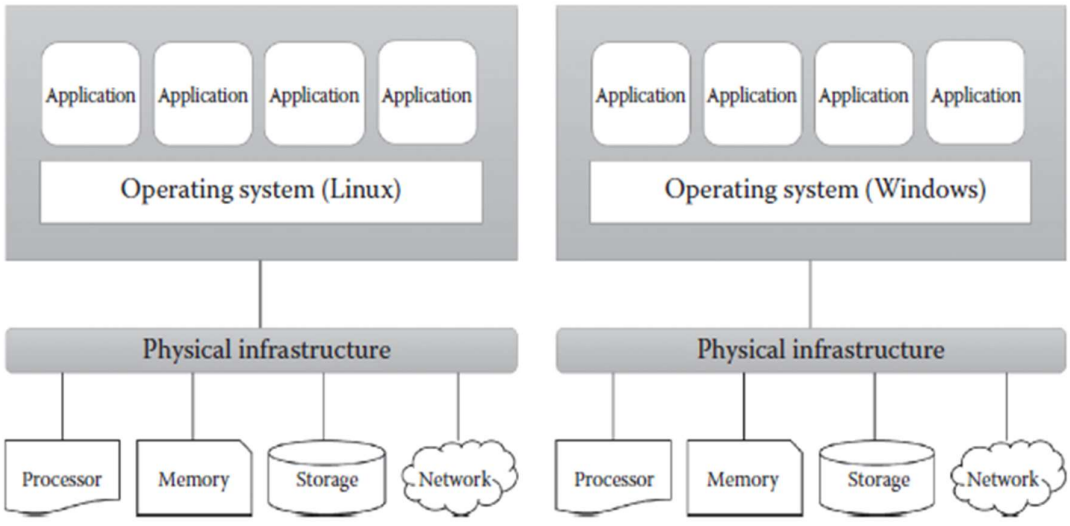


FIGURE 7.1
Before virtualization.

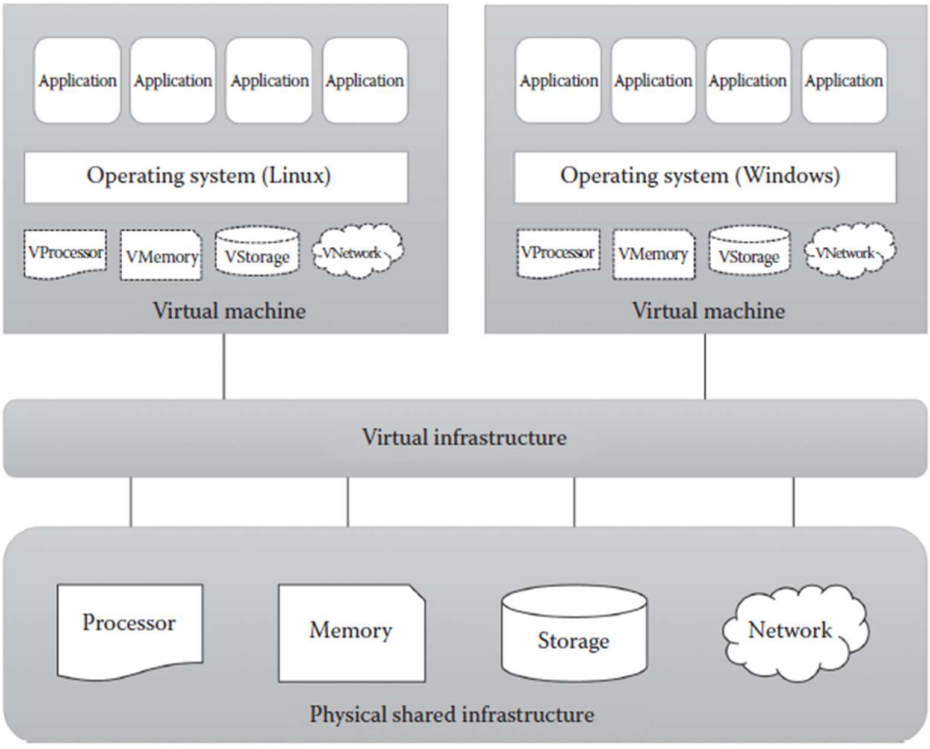


FIGURE 7.2
After virtualization.

6. Virtualization and Cloud Computing

- **Virtualization ≠ Cloud Computing**
  - Virtualization is a **technology** (creates virtual resources).
  - Cloud computing is a **service model** (delivers IT resources on demand using virtualization).
- **Relationship:**
  - Virtualization is the **foundation of cloud computing**.
  - Without virtualization, cloud computing cannot provide **scalability, resource pooling, and multi-tenancy**.
  - Example: In a public cloud (like AWS, Azure, GCP), virtualization enables multiple users to share the same physical data center infrastructure securely.

2 Virtualization Opportunities

Virtualization is the process of **abstracting physical resources into virtual resources** that can be allocated to **Virtual Machines (VMs)**.

The major resources that can be virtualized include **processor, memory, storage, network, data, and applications**.

Below are the different virtualization opportunities:

1 Processor Virtualization

- Allows VMs to share virtual processors that are created from the **physical processors** of the system.
- The **virtualization layer (hypervisor)** abstracts physical CPUs into **virtual CPUs (vCPUs)**.

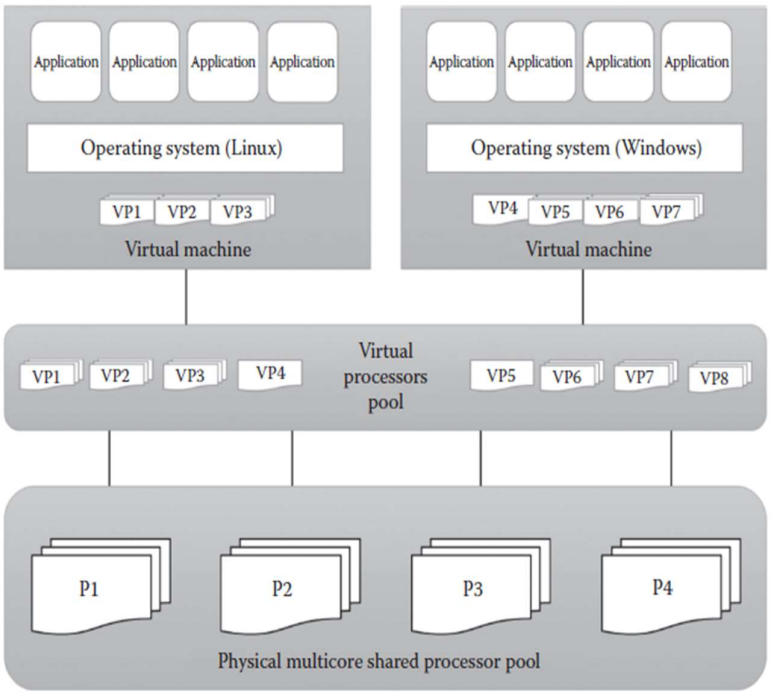


FIGURE 7.3
Processor virtualization.

- **How it works:**
  - A hypervisor manages the scheduling of physical processor time across multiple VMs.
  - Each VM "thinks" it has its own processor.
- **Where used:**
  - Data centers and cloud environments.
  - Can be implemented from a **single server** or even from **distributed servers** (resource pooling).

2 Memory Virtualization

- Provides VMs with **virtual main memory** mapped from the underlying **physical memory**.
- Similar to the concept of **virtual memory** in OS.

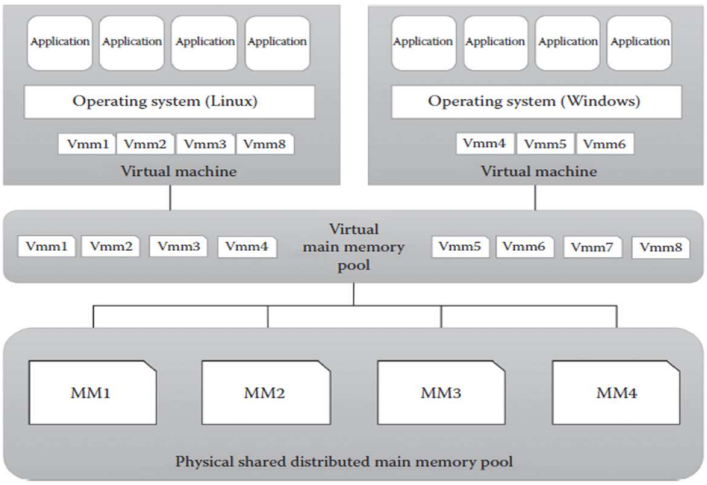


FIGURE 7.4
Main memory virtualization.

- **How it works:**
  - Maps **virtual page numbers → physical page numbers**.
  - Supported by modern x86 processors.
- **Advanced usage:**
  - Hypervisors can consolidate unused memory from different servers into a **virtual memory pool**.
  - This pool is dynamically allocated to VMs when required.
- **Benefit:**
  - Increases flexibility and ensures no memory remains idle.

3 Storage Virtualization

- Combines multiple **physical storage devices** into a pool of **logical storage (virtual disks)**.
- Each VM sees only the logical storage, not the underlying hardware.

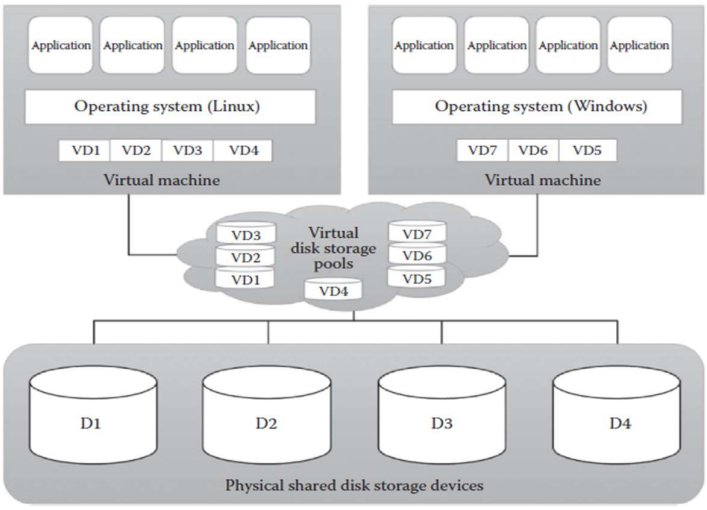


FIGURE 7.5
Storage virtualization.

- **Uses:**
  - Efficient utilization of physical storage.
  - Data **backup and replication**.
  - Ensures **high availability** of data.
- **Technologies used:**
  - Hypervisors
  - **Storage Area Networks (SANs)**
  - **Network-Attached Storage (NAS)**

4 Network Virtualization

- Creates a **virtual network** by abstracting physical networking components (routers, switches, NICs).
- Managed by virtualization software.

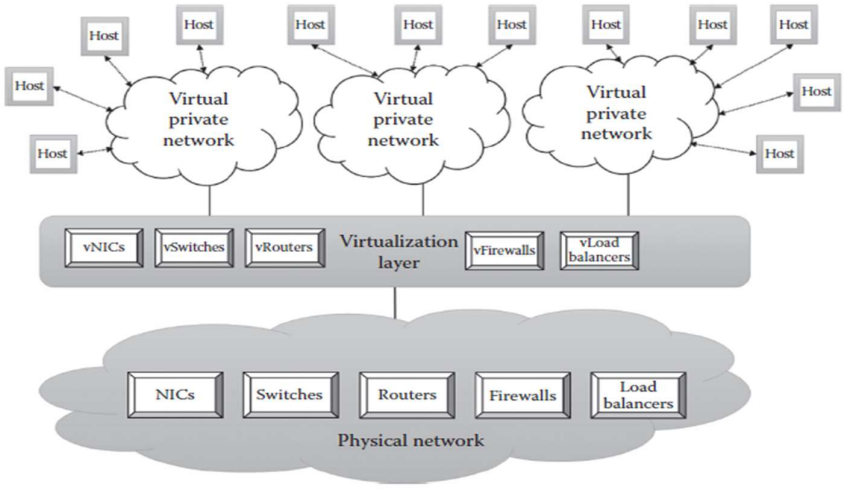


FIGURE 7.6
Network virtualization.

- **Features:**
    - Virtual networks act as a **software-based entity** that includes both network hardware and software.
    - Enables **communication between VMs** on the same physical network.
- **Types of VM Network Access:**

1. **Bridged Network** – VM directly connected to physical network.
2. **NAT (Network Address Translation)** – VM shares the host's IP address.
3. **Host-Only Network** – VM communicates only with the host system.

5 Data Virtualization

- Provides access to data **without concern for type or physical location**.
- Aggregates heterogeneous data from different sources into a **single logical data view**.

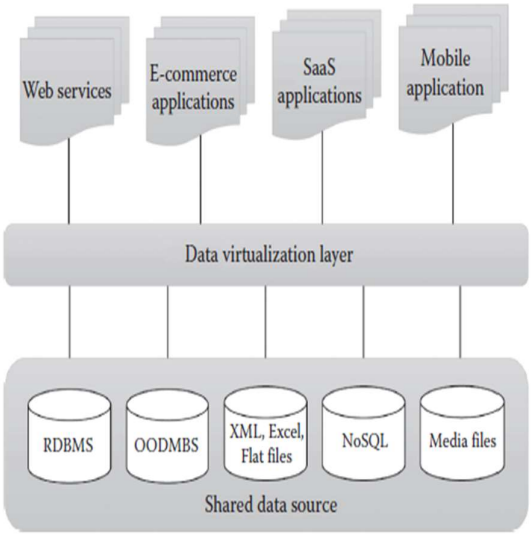


FIGURE 7.7
Data virtualization.

- **Features:**
    - Hides complexity of data type and location from applications.
    - Ensures **single point of access** to distributed data.
- **Uses:**
    - **Data integration**
    - **Business Intelligence (BI)**
    - **Cloud computing** (data available to SaaS, mobile apps, portals, etc.)

6 Application Virtualization

- Allows users to run applications **without installing them locally**.
- Applications are **developed, hosted, and virtualized on a central server**.

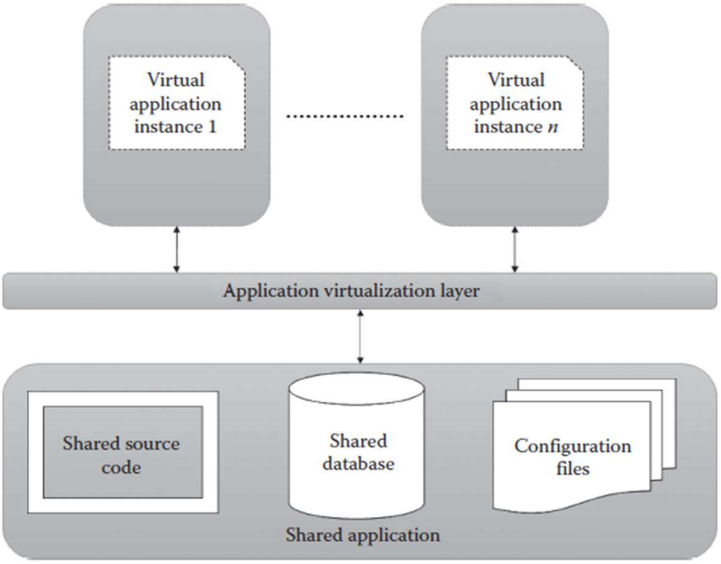
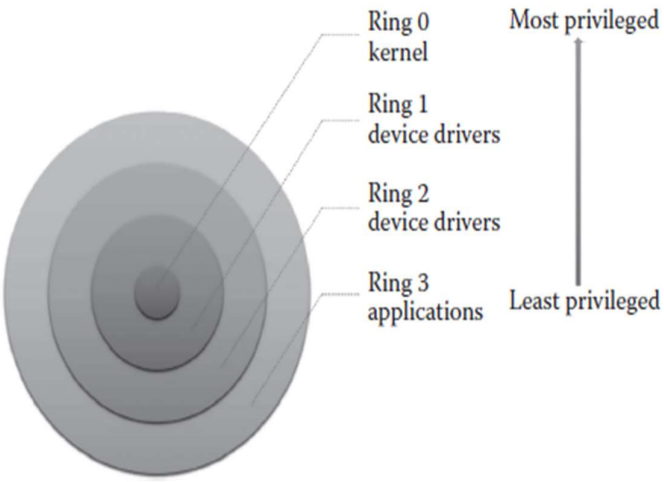


FIGURE 7.8
Application virtualization.

- **How it works:**
  - Users are provided with an **isolated virtual copy** of the application.
  - Simplifies deployment (no need for client-side installations).
- **Relation to Cloud:**
  - Key enabling technology for **Software as a Service (SaaS)**.
- **Examples:**
  - Microsoft Office 365, Google Workspace, Virtual Desktop applications.

Approaches to Virtualization

Before understanding the approaches, we need to know about **protection rings in Operating Systems (OSs).**



🔹 **Protection Rings in OSs**

- Protection rings define **privilege levels** to separate trusted OS code from untrusted applications.
- **Ring 0** → Most privileged (Kernel of OS, full hardware access).
- **Ring 1 & Ring 2** → Device drivers.
- **Ring 3** → Least privileged (user applications).
- Purpose: Prevent malicious or faulty applications (at Ring 3) from directly accessing physical resources.

👉 In virtualization, depending on the approach, the **hypervisor (VMM)** and **guest OS** are placed at different rings.

1 Full Virtualization

In **full virtualization**, the **guest operating system (OS)** is **completely unaware** that it is running in a virtualized environment. It assumes it has full control of the hardware, but in reality, the **hypervisor (VMM)** intercepts and manages all privileged instructions.

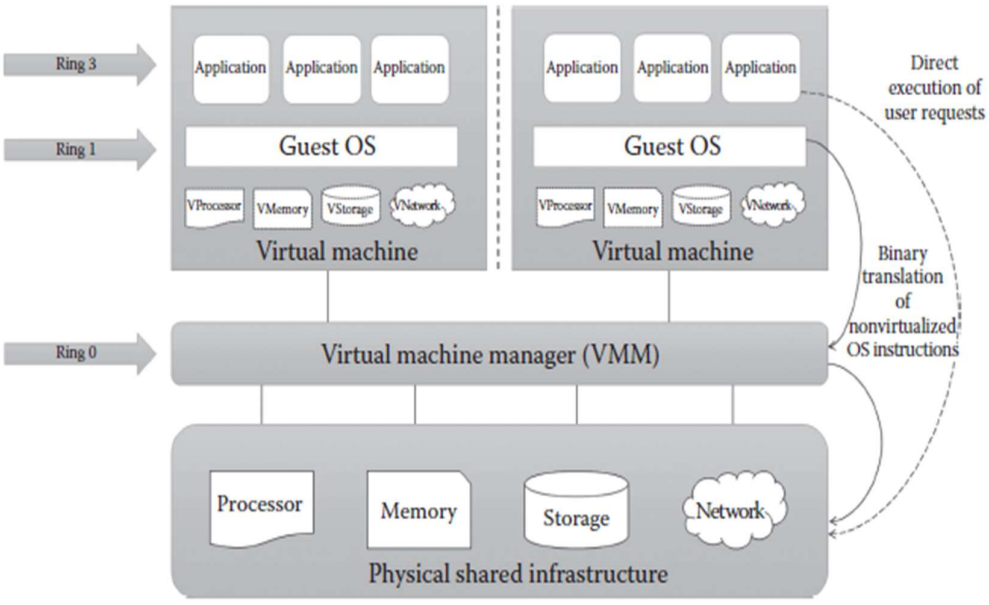


FIGURE 7.10
Full virtualization.

🔹 **Protection Ring Mapping**

- **Ring 0 (Most privileged):**

  Hypervisor / VMM runs here.

  - Has complete control over CPU, memory, storage, and I/O.
  - Acts as the middle layer between guest OS and hardware.

- **Ring 1:**

  Guest OS kernel runs here.

  - Unlike on a physical machine (where OS runs in Ring 0), here it gets **demoted**.
  - This ensures that the hypervisor remains in control.
  - Guest OS cannot directly interact with hardware; all privileged operations must go through the hypervisor.
- **Ring 3 (Least privileged):**

  User applications.

  - Behave normally, as if running on a real machine.
  - Applications are completely unaware of virtualization.

🔹 **Key Techniques Used**

1. **Binary Translation**
   - Used for privileged instructions issued by the guest OS kernel.
   - Since the guest OS cannot execute such instructions directly in Ring 1, the hypervisor translates them into **safe equivalent instructions** that achieve the same result but without compromising isolation.
2. **Direct Execution**
   - Used for non-privileged user-level instructions (applications).
   - Applications in Ring 3 can execute directly on the CPU without modification, which improves performance.

✅ **Pros:**

- Strong isolation and security between VMs.
- Multiple OSs can run simultaneously.
- Guest OS can be migrated easily to native hardware.
- Easy to use (no modification of guest OS).

❌ **Cons:**

- Binary translation adds overhead, reducing performance.
- Requires proper hardware-software combinations.

2 Paravirtualization

Paravirtualization is a virtualization technique in which the **guest OS is aware** that it is running in a virtualized environment. Instead of being fully abstracted from the hardware, the guest OS is **modified** to interact with the hypervisor directly using **hypercalls** (special calls similar to system calls).

This approach provides **partial simulation** of the hardware and reduces overhead compared to full virtualization.

📌 **Protection Ring Mapping**

- **Ring 0 (Privileged):**

  Modified Guest OS Kernel

  - Unlike full virtualization (where guest OS runs in Ring 1), here the **guest OS kernel is allowed to run in Ring 0**.
  - It communicates directly with the hypervisor via **hypercalls**.
- **Ring 3 (Least Privileged):**

  User Applications

  - Run unmodified.
  - Like in full virtualization, they believe they're running on a real machine.

📌 **Key Concept: Hypercalls**

- Hypercalls = **special calls from guest OS to the hypervisor**, similar to system calls from applications to the OS.
- Replace **non-virtualizable instructions** in the guest OS.
- Allow **direct communication** with the hypervisor, eliminating the need for **binary translation**.

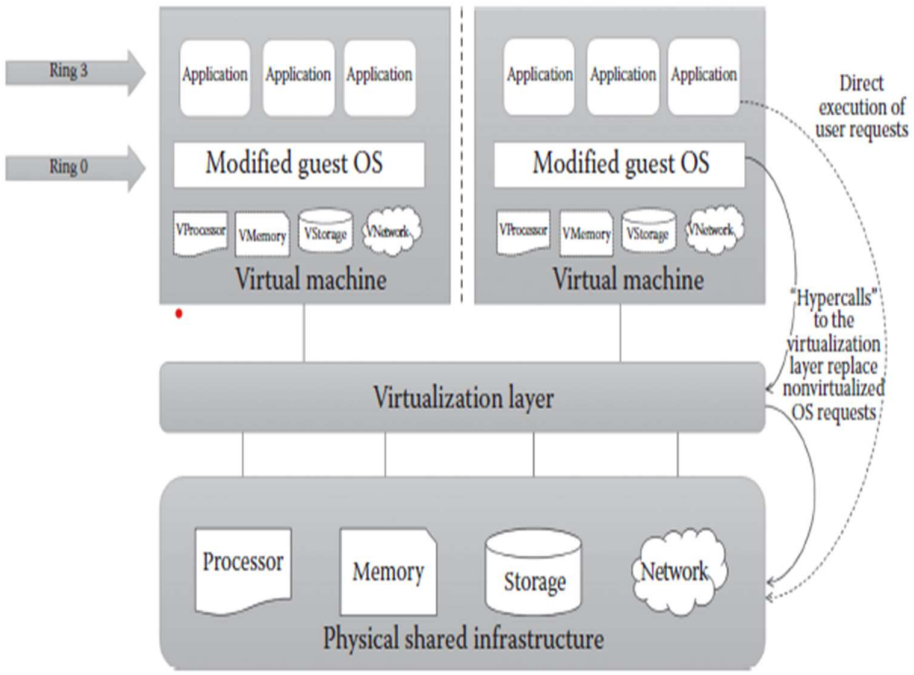


FIGURE 7.11
Paravirtualization.

✅ **Pros:**

- No binary translation → Higher efficiency and performance.
- Easier to implement compared to full virtualization.

❌ **Cons:**

- Requires **guest OS modification** → not always possible.
- Modified OS cannot run on physical hardware directly.
- Lack of backward compatibility → harder migration across different hosts.

3 Hardware-Assisted Virtualization

Hardware-Assisted Virtualization is a virtualization technique in which the **CPU itself provides built-in support for virtualization**. Instead of relying on **binary translation (full virtualization)** or **guest OS modification (paravirtualization)**, modern processors (Intel and AMD) include special **virtualization extensions** that allow the hypervisor to run **below Ring 0** at a special privilege level.

This eliminates much of the overhead seen in the other two approaches.

📌 **Hardware Support**

- **Intel VT-x (Virtualization Technology):**

  Uses **Virtual Machine Control Structures (VMCS)** to store guest state.
- **AMD-V (AMD Virtualization):**

  Uses **Virtual Machine Control Blocks (VMCB)** to manage virtualization.

These hardware extensions allow direct trapping of privileged instructions to the hypervisor, without binary translation or OS modification.

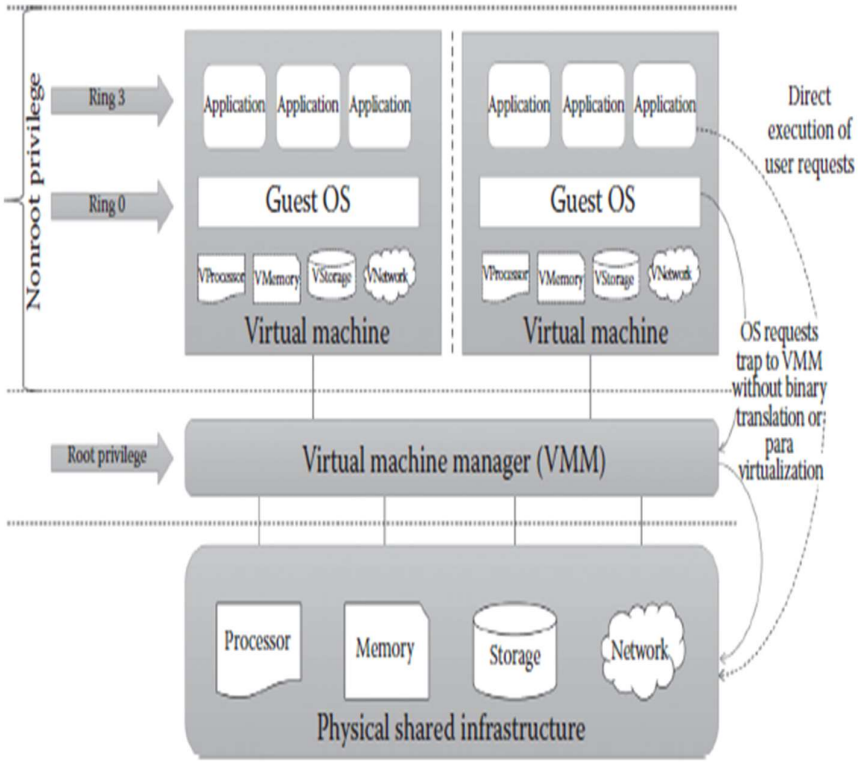


FIGURE 7.12
Hardware-assisted virtualization.

📌 **Protection Ring Mapping**

- **Root Mode (Ring -1):**

  Hypervisor (VMM)

  - Runs at a higher privilege level than even the OS kernel.
  - Has complete control of the hardware.
- **Ring 0:**

  Guest OS kernel

  - Runs in **Ring 0 as usual**, without being demoted.
  - Still under control of the hypervisor.
- **Ring 3:**

  User Applications

  - Run unmodified as if on a normal machine.

📌 **How It Works**

1. A **user application (Ring 3)** makes a system call.
2. The **guest OS kernel (Ring 0)** executes normally.
3. If the guest OS issues privileged instructions:
   - The **hardware traps** these instructions directly to the **hypervisor (Ring -1)**.
   - No binary translation or hypercalls are needed.
4. The hypervisor validates and executes the instructions.

✅ **Pros:**

- Removes inefficiencies of other approaches.
- Guest OS does not require modification.
- Higher performance and better compatibility.

❌ **Cons:**

- Dependent on specific hardware (Intel VT-x, AMD-V).
- Slight hardware overhead compared to native execution.

---

4 Hypervisors

◇ **Virtual Machines (VMs) & Hypervisors**

📌 **Why VMs?**

- Traditionally, there was a **one-to-one relationship** between hardware and OS.
- This led to **underutilized resources** (e.g., a powerful server running only one OS and one app).
- With **virtualization**, many VMs can share a single physical server → **better resource utilization, lower cost, and energy efficiency (Green IT).**

What is a Hypervisor?

A **Hypervisor** (also called **Virtual Machine Monitor – VMM**) is a software or firmware layer that:

- Sits **between the physical hardware and the VMs**.
- Provides **virtual infrastructure** such as:
  - vCPU (virtual CPU)
  - vMemory (virtual RAM)
  - vNICs (virtual network cards)
  - vStorage
  - vI/O devices
- Enables multiple OSs to run on the same physical server.

Types of Hypervisors

1. Type 1 Hypervisor (Bare-Metal / Native Hypervisor)

- Runs **directly on hardware** (no host OS).
- Has direct access to CPU, memory, and devices.
- **Efficient and secure** → less overhead, more suitable for **servers and data centers**.
- Used for **heavy workloads and high-security environments**.

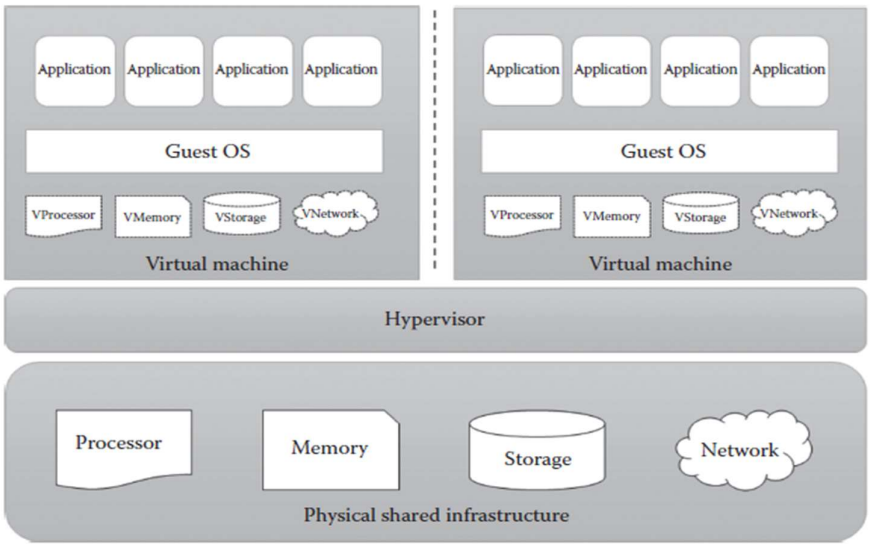


FIGURE 7.13
Type 1 or bare metal hypervisor.

Examples:

- **Microsoft Hyper-V (Server edition)**
- **Citrix XenServer**
- **VMware ESXi**
- **Oracle VM Server for SPARC**

☑ **Use case:** Cloud providers (AWS, Azure, Google Cloud) use Type 1 hypervisors to host thousands of VMs.

2. Type 2 Hypervisor (Hosted / Embedded Hypervisor)

- Runs **on top of a host operating system**.
- The host OS manages the hardware, and the hypervisor works like an **application**.
- **Less efficient** than Type 1, since there's an extra host OS layer.
- If the **host OS crashes → all VMs crash** too.
- Best for **desktops, laptops, and testing environments**, not for production servers.

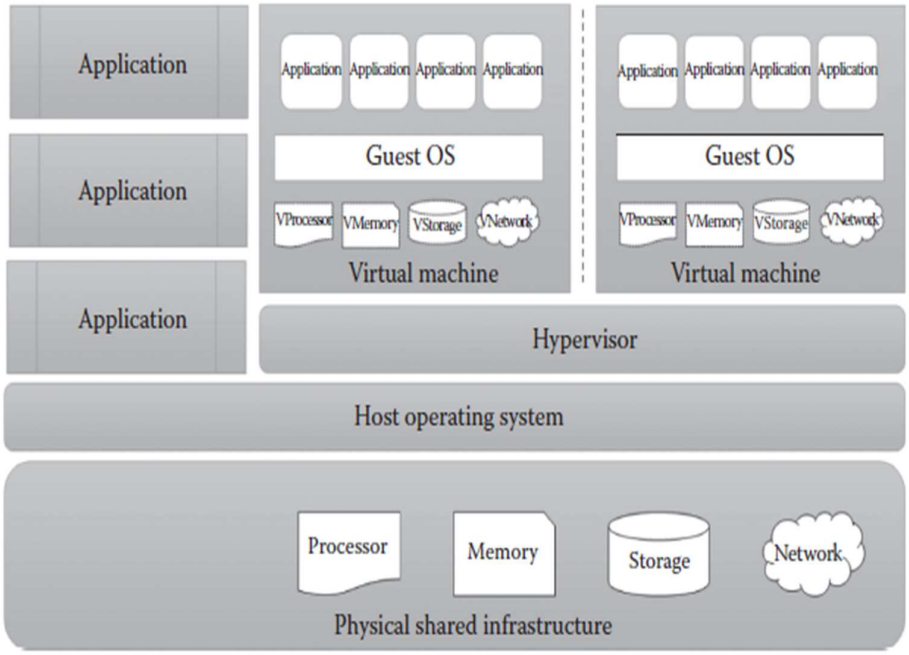


FIGURE 7.14
Type 2 or hosted hypervisor.

◇ **Examples:**

- **VMware Workstation**
- **Oracle VirtualBox**
- **Parallels Desktop (Mac)**

☑ **Use case:** A developer runs **Linux VMs inside Windows** using VirtualBox for software testing.

- **Disadvantages:**
  - Extra overhead → lower efficiency.
  - If the **host OS crashes**, all VMs also fail.
- **Best for:**
  - **Client systems** where performance is less critical (testing, development).

2 Security Issues in Hypervisors

Since hypervisors manage multiple VMs and have **direct access to hardware**, they are a **prime target for attackers**.

Main attack vectors:

1. **Attack through Host OS (Type 2 hypervisors):**
    - Exploit vulnerabilities in host OS.
    - If host OS is compromised → attacker gains control of hypervisor.

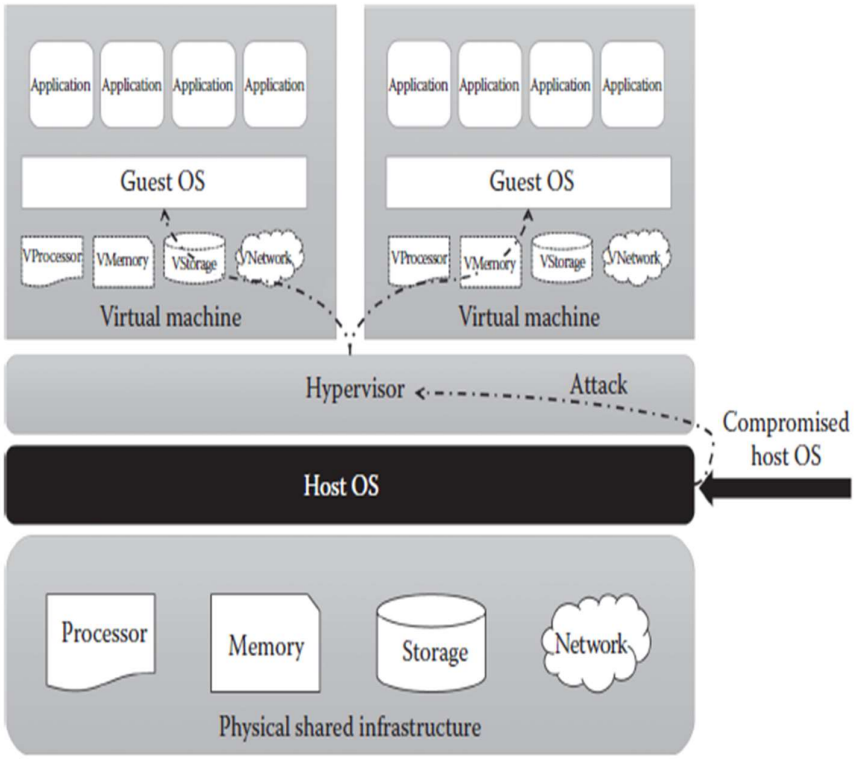


FIGURE 7.15
Attack through the host OS.

    - **Possible attacks:**
        - **Denial of Service (DoS):** Deny virtual resources to new VMs.
        - **Data theft:** Steal confidential VM data.
2. **Attack through Guest OS (Type 1 & Type 2):**
    - Malicious guest OS or VM sends harmful requests to hypervisor.
    - **Possible attacks:**
        - Unauthorized access to other VMs.
        - Abuse hardware resources (resource exhaustion).

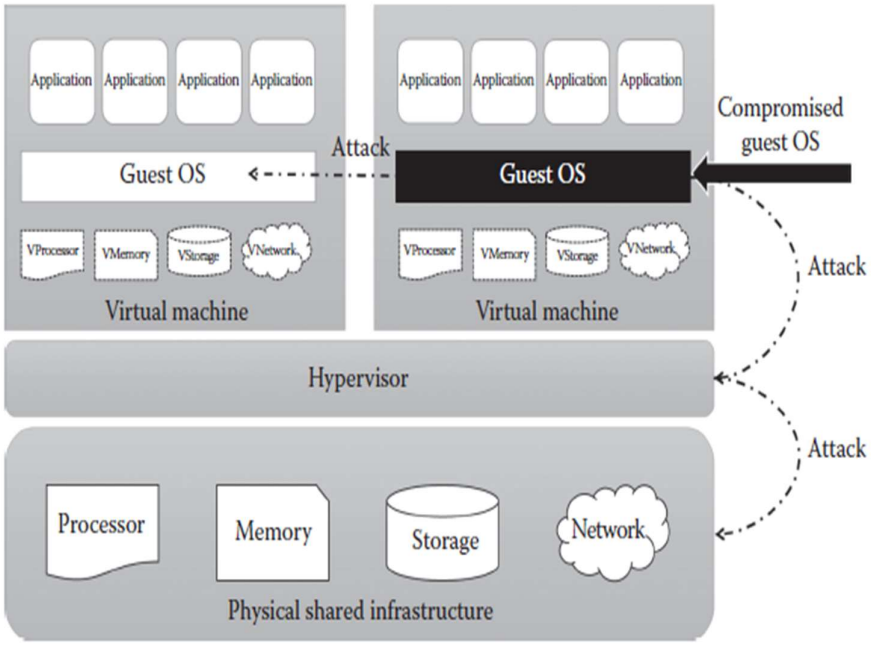


FIGURE 7.16
Attack through the guest OS.

Recommendations for Securing Hypervisors

To protect hypervisors from attacks:

- **Regular updates:** Keep hypervisor and host OS patched.
- **Disable unused hardware resources** to reduce attack surface.
- **Least privilege principle:** Restrict hypervisor and guest OS privileges.
- **Monitoring tools:** Deploy intrusion detection/prevention in hypervisor.
- **Strong guest isolation:** Prevent compromised VM from affecting others.
- **Mandatory access control (MAC):** Enforce strict policies on access rights.

---

5. From Virtualization to Cloud Computing

Many people mistakenly think **virtualization** and **cloud computing** are the same.

➡️ **Reality:** They are **different** technologies, but **virtualization is an enabler of cloud computing**.

We can compare them on several parameters:

1. Type of Service

- **Virtualization:**
  - Primarily provides **infrastructure-level services** (servers, storage, networking).
  - Focused on resource utilization within data centers.
- **Cloud Computing:**
  - Provides **Infrastructure (IaaS), Platform (PaaS), and Software (SaaS)** services.
  - Covers complete IT service delivery.

2. Service Delivery

- **Virtualization:**
  - Not designed for **on-demand** service delivery.
  - Services are provisioned and managed manually.
- **Cloud Computing:**
  - Services are **on-demand** and accessed anytime by end users.
  - Example: AWS, Azure, GCP provide resources instantly when requested.

3. Service Provisioning

- **Virtualization:**
  - Requires **manual intervention** by IT administrators to allocate resources.
- **Cloud Computing:**
  - Supports **automated, self-service provisioning** for end users.
  - Example: A developer can launch a VM or database instance from a portal without admin help.

4. Service Orchestration

- **Virtualization:**
  - Cannot orchestrate or compose multiple services automatically.
- **Cloud Computing:**
  - Supports **service orchestration and composition**.
  - Providers even offer automated orchestration tools (e.g., Kubernetes, AWS CloudFormation).

5. Elasticity

- **Virtualization:**
  - Limited flexibility.
  - Starting/stopping VMs is **manual** and scaling is **difficult**.
- **Cloud Computing:**
  - Offers **elasticity and auto-scaling**.
  - Resources can be **added/removed dynamically** as per workload.

6. Target Audience

- **Virtualization:**
  - Mainly for **service providers or IT owners** (to improve hardware utilization and reduce costs).
- **Cloud Computing:**
  - Targets both **service providers** (higher ROI) and **end users** (pay-per-use, no infrastructure ownership).

Cloud computing delivers IT resources as a service using **virtualization** as the enabling technology.

The three major service models are:

1. **Infrastructure as a Service (IaaS)**
2. **Platform as a Service (PaaS)**
3. **Software as a Service (SaaS)**

1 Infrastructure as a Service (IaaS)

- IaaS provides **virtualized infrastructure resources** (compute, storage, network) to customers on a **pay-per-use** basis.

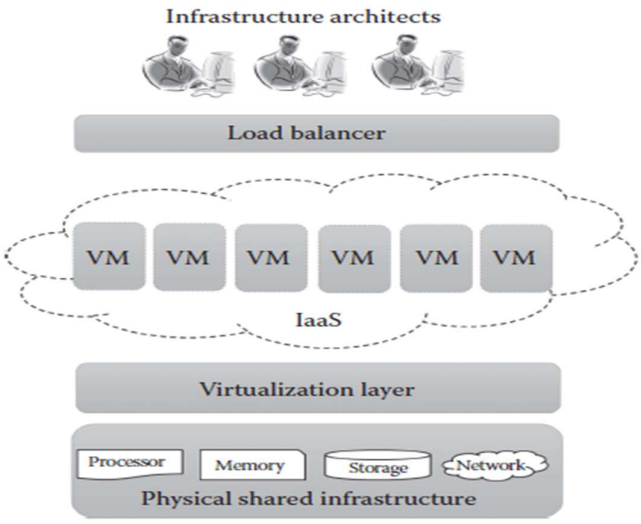


FIGURE 7.17
IaaS.

- **How it works:**
  - Uses **processor, memory, storage, and network virtualization**.
  - Hypervisors (mainly **Type 1**) abstract the physical resources into virtual resources.
  - Delivered in the form of **Virtual Machines (VMs)**.
- **Features:**
  - Customers can create, configure, and manage VMs.
  - Provides **virtual CPUs, memory, storage, and networks**.
  - Load balancers distribute traffic across multiple servers for scalability.
- **Examples of IaaS providers:**
  - Amazon Web Services (EC2)
  - Microsoft Azure IaaS
  - OpenStack
  - CloudStack
  - Eucalyptus

✅ **Use case:** Hosting servers, running enterprise applications, test and development environments.

2 Platform as a Service (PaaS)

- PaaS provides a **virtual development platform** that enables users to **develop, test, and deploy applications** without worrying about the underlying infrastructure.

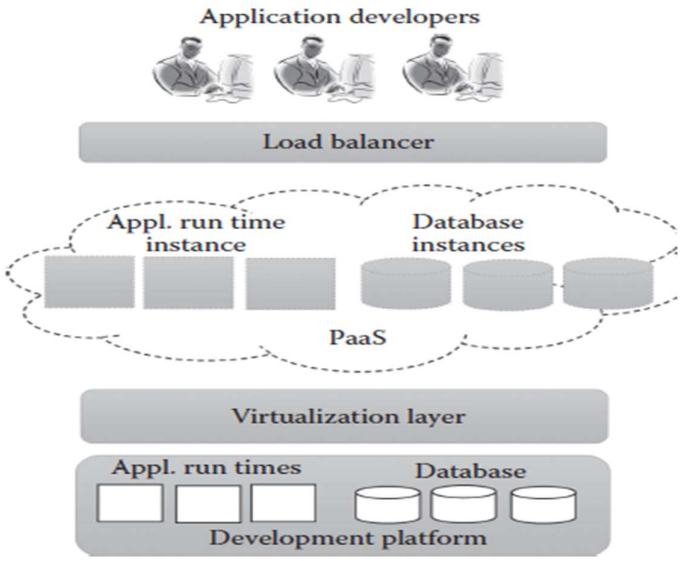


FIGURE 7.18
PaaS.

- **How it works:**
  - Service providers offer **programming languages, runtime environments, databases, middleware, and libraries** virtually.
  - Developers can build applications using online platforms accessed via **Web UIs, REST APIs, or WebCLI**.
  - Supports both **online and offline development** (integration with IDEs like Eclipse).
- **Features:**
  - Developers don't need to install software tools locally.
  - Application deployment depends on **cloud deployment model**:
    - **Public cloud** → off-premise hosting.
    - **Private cloud** → on-premise hosting.
  - Uses **OS-level and software-level virtualization**.
  - Supports **scalability** via load balancing.
- **Examples of PaaS providers:**
  - Google App Engine
  - Microsoft Azure App Services
  - RedHat OpenShift
  - Force.com (Salesforce platform)

✅ **Use case:** Application development and deployment (web apps, APIs, business apps).

3 Software as a Service (SaaS)

- SaaS provides **software applications** over the internet, hosted in the provider's data center.

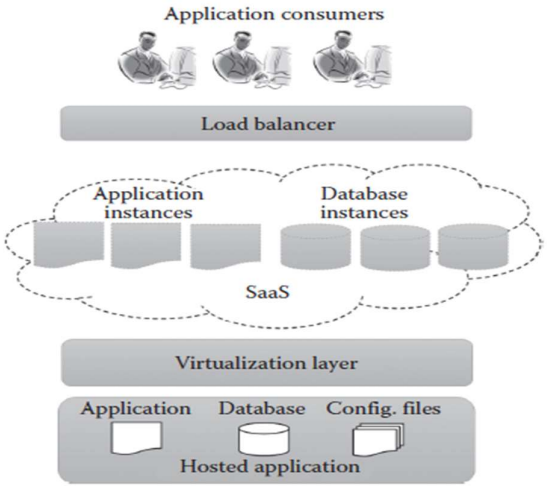


FIGURE 7.19
SaaS.

- **How it works:**
  - Applications are accessed through a **web browser**; no need to install locally.
  - Uses **application-level virtualization** to deliver apps.
  - Supports **multitenancy** → multiple users share the same software instance securely.
- **Features:**
  - Subscription-based pricing (not licensed software).
  - Highly **scalable** using software load balancers that replicate application/database servers.
  - Supports multiple application and database instances for high availability.
- **Examples of SaaS applications:**
  - Google Docs, Google Drive
  - Microsoft Office 365
  - Salesforce CRM

✅ **Use case:** Email, office suites, collaboration tools, customer relationship management (CRM).

🌟 Beyond IaaS, PaaS, SaaS

Other cloud services enabled by **virtualization**:

- **Network as a Service (NaaS)** → using network virtualization.
- **Storage as a Service (STaaS)** → using storage virtualization.
- **Database as a Service (DBaaS)** → using database virtualization.

Programming Models for Cloud Computing

1. Introduction to Programming Models

A **programming model** is a specific way, method, or approach followed in programming. It defines how software is designed, how problems are solved, and how the code is executed on underlying hardware or platforms.

There are several programming models available in general—such as **procedural, object-oriented, event-driven, functional, and parallel programming**—each having its own **advantages and disadvantages**.

Programming models form the **foundation of software and application development**. The properties of each model determine its **suitability for specific use cases** and its **impact on performance, scalability, and maintainability**.

📌 **Example:**

- An **object-oriented programming model** (Java, C++) is best suited for large-scale enterprise applications where reusability and modularity are important.
- A **parallel programming model** (OpenMP, MPI) is used in scientific applications like climate simulations or DNA sequencing, where massive computations need to run in parallel.

2. Evolution of Programming Models

As technology evolved, **programming models also evolved** to meet new challenges. Each evolutionary step added new functionality:

- From **machine-level programming** → to **structured programming** → to **object-oriented programming** → to **parallel/distributed programming** → to **cloud-specific programming models**.

However, **no single programming model can solve all problems**. For domain-specific challenges, new models are often developed.

📌 **Example:**

- Big Data analytics required new models like **MapReduce**.
- Real-time IoT data processing uses **streaming models** (Apache Kafka, Apache Flink).
- Serverless computing uses **Function-as-a-Service (FaaS) models** like AWS Lambda.
-

3. Cloud Computing and Its Impact

Cloud computing has transformed the computing era by enabling **on-demand services** in computing, storage, and networking without requiring organizations to invest in costly infrastructure (CAPEX).

Why cloud is popular:

- **On-demand access** (pay for what you use)
- **Scalability and elasticity** (resources scale automatically)
- **Market-driven growth** (adopted by Amazon, Google, Microsoft, etc.)
- **Global adoption** (businesses migrate applications to cloud for flexibility and cost savings)

📌 **Real-world Example:**

- **Amazon Web Services (AWS)** offers computing (EC2), storage (S3), and database (RDS) services on-demand.
- **Microsoft Azure** provides enterprise cloud solutions for banking, healthcare, and government services.

4. Service Models of Cloud

Cloud's architecture is built on **three service models**:

1. **Infrastructure as a Service (IaaS):** Provides raw infrastructure (servers, storage, networks) on-demand.
    - 📌 Example: AWS EC2, Google Compute Engine, Azure Virtual Machines.
2. **Platform as a Service (PaaS):** Provides a development platform with tools and frameworks to build applications.
    - 📌 Example: Google App Engine, AWS Elastic Beanstalk, Heroku.
3. **Software as a Service (SaaS):** Provides software as a service to users, running on the cloud.
    - 📌 Example: Salesforce CRM, Microsoft 365, Dropbox.

👉 Among these, **SaaS is most important** for end-users since it delivers ready-to-use software. However, **to design SaaS applications, a suitable programming model must be chosen**.

5. Properties of Cloud Applications

Cloud applications differ from conventional ones because they must support:

- **Scalability:** Handle growing workloads efficiently.
    - 📌 Example: Netflix scaling servers during new movie releases.
- **Concurrency:** Multiple users access the application simultaneously.
    - 📌 Example: Google Docs allows thousands of users to edit documents at once.
- **Multitenancy:** One instance of the application serves multiple users or organizations.
    - 📌 Example: Salesforce CRM supports thousands of companies on a shared platform.
- **Fault Tolerance:** Continue functioning despite failures in hardware/software.
    - 📌 Example: AWS automatically replicates data across regions to avoid downtime.

6. Programming Models for Cloud

There are **two main approaches** in cloud programming models:

A. Extending Existing Programming Models

Adapting traditional models with modifications for cloud environments.

- **MapReduce (Big Data processing):** Google BigQuery, Hadoop.
- **Message Passing Models:** Amazon SQS for Uber ride-matching.
- **Service-Oriented Architecture (SOA):** PayPal API used by e-commerce platforms.

B. Developing New Programming Models

Models built specifically for cloud applications.

- **Serverless (FaaS):** AWS Lambda, Azure Functions.
- **Dataflow Models:** Apache Beam for real-time streaming analytics.
- **Actor Model:** Microsoft Orleans, used in Halo gaming for millions of players.

🌐 **Extended Programming Models for Cloud**

Cloud computing introduces **unique challenges** that are not present in traditional computing systems. While many programming models exist for application development, **not all of them can be directly migrated to the cloud** because the cloud has certain **distinct properties** such as scalability, concurrency, multitenancy, and elasticity.

To overcome this, researchers and developers often **extend existing programming models** so that they can handle cloud-specific requirements instead of inventing everything from scratch.

🔑 **Key Properties That Matter in Cloud Programming Models**

1. **Scalability**
    - The system's ability to **scale up (add more resources)** when demand increases or **scale down (reduce resources)** when demand decreases.
    - Why important: Cloud must support **millions of dynamic users**, from a few during off-peak hours to millions during peak usage.
    - Example:
        - **Netflix on AWS** – During peak streaming times, extra servers are automatically provisioned, and during low-traffic hours, resources are scaled down.
2. **Concurrency**
    - The ability of a system to handle **multiple tasks/users simultaneously** without slowing down or crashing.

- Why important: Cloud services often need to serve **thousands or millions of clients** at the same time.
- Example:
  - **Google Docs** – Multiple people editing the same document in real time is possible because concurrency is supported by the programming model and infrastructure.

3. **Multitenancy**
   - A single cloud application can serve **multiple customers (tenants)** at the same time, keeping data secure and isolated.
   - Why important: Ensures cost efficiency by sharing the same resources across many users.
   - Example:
     - **Gmail** – Millions of users share the same Google infrastructure, but each inbox is private and secure.

◆ **Why Extend Existing Models Instead of Creating New Ones?**

- Many **traditional programming models** already have concepts of **parallelism, distribution, and data partitioning**, which overlap with cloud requirements.
- Instead of reinventing the wheel, it is often more efficient to **adapt and extend** these models for cloud environments.
- For example:
  - **MapReduce** was originally created for large-scale data processing at Google, and later extended for cloud environments (Hadoop, Spark, etc.).
  - **Actor Model** (used in distributed systems like Erlang or Akka) was extended to cloud environments to handle concurrency at scale.

◇ 1. MapReduce

MapReduce is a **parallel and distributed programming model** created by Google to handle **large-scale data processing**. It divides work into two main functions:

1. **Map Function** – Breaks down the input data into smaller pieces and processes them in parallel.
2. **Reduce Function** – Combines the results from the map stage and produces the final output.

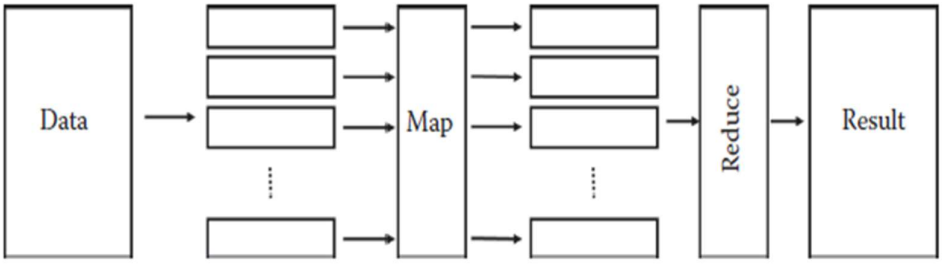


FIGURE 8.1
MapReduce.

🛠 Step-by-Step Working of MapReduce

1. **Input Data**
   - A huge dataset (structured, semi-structured, or unstructured) is stored across distributed machines.
   - Example: Web server logs of 1 billion users accessing Amazon.
2. **Splitting the Data**
   - The large dataset is divided into **chunks (splits)**.
   - Example: Logs are divided based on dates, regions, or users.
3. **Map Phase**
   - Each chunk is processed in **parallel** by the **map function**.
   - The map function transforms input into **key-value pairs (k, v)**.
   - Example:
     - Input: "user1 clicked productA".
     - Map Output: (productA, 1)
4. **Shuffling & Sorting**
   - The system groups all key-value pairs with the same key together.
   - Example: (productA, 1), (productA, 1), (productB, 1) → grouped as (productA, [1,1]) , (productB, [1]).
5. **Reduce Phase**
   - The **reduce function** aggregates values for each key.
   - Example:
     - Input: (productA, [1,1])
     - Reduce Output: (productA, 2)
6. **Final Result**
   - The combined output is written back as the **final result**.
   - Example: ProductA clicked 2 times, ProductB clicked 1 time.

✅ **Real-Time Examples of MapReduce**

1. **E-commerce Analytics**
    - **Input:** Customer clickstream logs.
    - **Map:** Extract (product, 1) each time a product is viewed.
    - **Reduce:** Count total clicks per product.
    - **Result:** Top 10 most-viewed products on Amazon.

2. **Social Media Trends**
    - **Input:** Tweets from Twitter.
    - **Map:** (hashtag, 1) for each hashtag.
    - **Reduce:** Sum counts per hashtag.
    - **Result:** Trending hashtags (#WorldCup, #Election2025).
3. **Fraud Detection in Banking**
    - **Input:** Transaction logs.
    - **Map:** Emit (userID, transaction_amount)
    - **Reduce:** Aggregate per user, detect anomalies (sudden huge spikes).

🔑 **Why MapReduce Fits Cloud?**

- **Scalability:** Runs across thousands of machines (Hadoop, Spark on AWS/Azure).
- **Concurrency:** Many map tasks run **in parallel**.
- **Fault-tolerance:** If one machine fails, tasks are reassigned automatically.
- **Multitenancy:** Multiple users can run jobs on shared infrastructure.

🔹 **I) Map Function**

- Input: A **large dataset** divided into small chunks.
- The **map function** processes each chunk and converts it into **key/value pairs (k, v)**.
- It usually works on a **single document (or record)**.

Formula:

$$Map(Key_1, Value_1) \longrightarrow List(Key_2, Value_2)$$

- $(Key_1, Value_1)$: Original input data.
- $(Key_2, Value_2)$: Intermediate pairs generated.

Example (Word Count):

- Input sentence: "Cloud computing is powerful. Cloud is scalable."
- Map function generates:
  - (Cloud, 1), (computing, 1), (is, 1), (powerful, 1), (Cloud, 1), (is, 1), (scalable, 1).

🔹 **II) Reduce Function**

- Input: A **list of intermediate key/value pairs** produced by the Map step.
- Before reduce, data undergoes **shuffle + sort + merge** → groups same keys together.
- The **reduce function** aggregates values for each unique key.

Formula:

$$Reduce(Key_2, List(Value_2)) \longrightarrow (Key_2, Value_3)$$

- Key_2: Word (like "Cloud").
- $List(Value_2)$: All values (counts) associated with that word.
- $Value_3$: Final aggregated result.

Example (Word Count):

- Input to reduce:
  - (Cloud, [1, 1])
  - (computing, [1])
  - (is, [1, 1])
  - (powerful, [1])
  - (scalable, [1])
- Reduce Output:
  - (Cloud, 2)
  - (computing, 1)
  - (is, 2)
  - (powerful, 1)
  - (scalable, 1)

✅ Final result = Count of each word in the document.

🚀 **Real-Time Applications of MapReduce**

1. **Word Count (as shown)** – Text mining, log file analysis.
2. **Search Engine Indexing (Google):** Map = scan web pages, Reduce = build keyword index.
3. **E-commerce (Amazon):**
   - Map = Extract product views (product, 1)
   - Reduce = Count total views (product, total)

4. **Social Media Analytics (Twitter, Facebook):** Trending hashtags, sentiment analysis.
5. **Fraud Detection (Banking):** Group and analyze suspicious transactions per user.

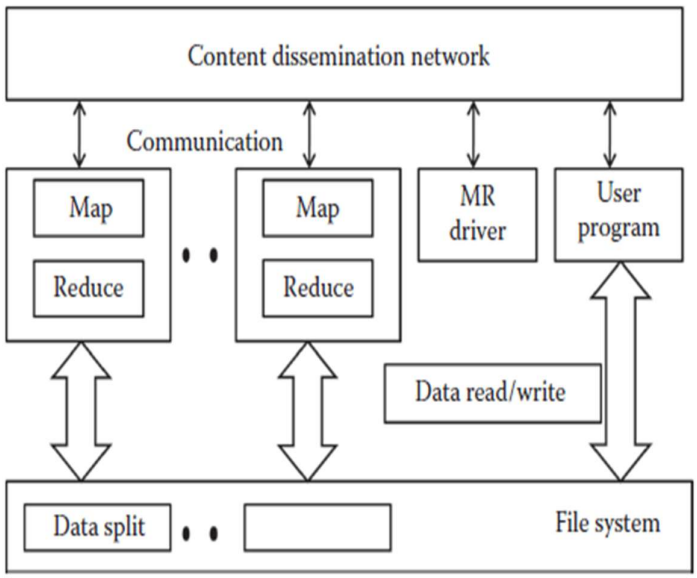
◇ 2 CGL-MapReduce

CGL-MapReduce was introduced by **Ekanayake et al.** to handle **data-intensive scientific applications** (like climate modeling, genomics, physics simulations).

The **key difference**:

- **Hadoop MapReduce** = Uses **File System (HDFS)** for communication.
- **CGL-MapReduce** = Uses **Streaming via Content Dissemination Network (CDN)**, avoiding file-system overhead.

This makes it **faster** and **more suitable for real-time scientific data**.

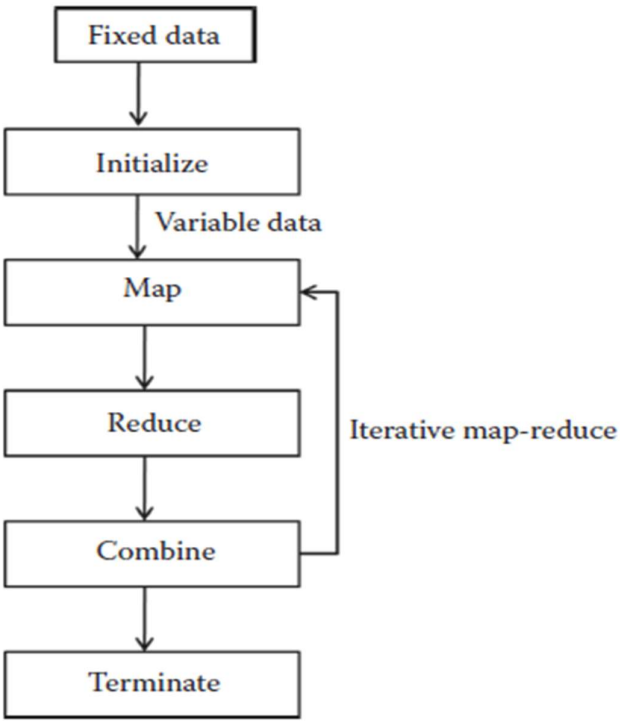


◆ Components of CGL-MapReduce

1. **Data Split & File System**
   - Input data is divided into chunks (splits).
   - But unlike Hadoop MR, **results are not always stored back into the file system**.
2. **Map Worker**
   - Processes the input split.
   - Generates intermediate **key/value pairs**.
   - **Directly streams output** to the **Reduce worker** (no need to write to HDFS).
3. **Reduce Worker**
   - Receives mapped data (via CDN streaming).
   - Performs aggregation/reduction (e.g., word count, summation).
   - Produces the final output.

4. **Content Dissemination Network (CDN)**
    - The **communication backbone** (uses **NaradaBroker**).
    - Handles streaming of intermediate results from **Map → Reduce → MR Driver → User Program**.
5. **MR Driver**
    - The **master controller**.
    - Assigns tasks to Map and Reduce workers.
    - Interacts with user program instructions.
6. **User Program**
    - Provides instructions (e.g., word count, data analysis job).
    - Interacts with MR Driver.
    - Reads/writes final data to/from the file system if required.

🔹 **Steps of CGL-MapReduce**



1. Initialize Stage (Fixed Data Input)

- Starts **MapReduce worker nodes** and configures the MapReduce task.
- Configuration is a **one-time process** → can be reused for multiple iterations.
- Fixed (static) data is loaded here (e.g., dataset schema, constants).

✅ **Real-time example**:

In a **climate simulation**, the **geographical boundaries** and **initial climate constants (like latitude, longitude, elevation)** are fixed and initialized once.

2. Map Stage (Variable Data Input)

- MRDriver starts **map computation**.
- **Variable (changing) data** is passed into map tasks.
- Results from one iteration can be reused in the next → enabling **iterative computation**.
- Uses **streaming** to send map results directly to **reduce workers** (no HDFS).

✅ **Real-time example**:

In **DNA sequencing**, each map worker gets a fragment of DNA sequence and outputs key/value pairs → (sequence ID, nucleotide pattern).

3. Reduce Stage

- After all map tasks finish, reduce workers aggregate the mapped results.
- Reduce workers start execution when initialized by MRDriver.
- Output is streamed directly to the **user program** (or to the combine stage if iterative).

✅ **Real-time example**:

In **weather prediction**, reduce workers aggregate mapped results → e.g., average temperature for each region.

4. Combine Stage

- Collects all reduced results and prepares them:
  - **Single-pass MapReduce** → results combined directly.
  - **Iterative MapReduce** → results fed back into the next **map stage** for further processing.

✅ **Real-time example**:

In **machine learning (clustering like K-means)**:

- Map = assign points to nearest cluster center.
- Reduce = compute new cluster centers.
- Combine = pass updated centers to next iteration until convergence.

5. Terminate Stage

- The final stage, triggered by the **user program**.
- All worker nodes shut down.
- Final results delivered to the user or stored in a file system if needed.

✅ **Real-time example**:

In **astronomy data analysis**, once all telescope data is processed iteratively, final celestial maps are produced and workers are terminated.

◇ Cloud Haskell: Functional Programming Model

1. Functional Programming Basis

- Cloud Haskell is based on **Haskell**, a **pure functional programming language**.
- In functional programming:
  - Functions behave like **mathematical functions** → output depends only on input.
  - **Immutability**: once data is created, it cannot be changed.
  - Functions are **idempotent** → can be retried safely without side effects.

✅ **Real-time example**:

If you request the **current exchange rate** (e.g., USD → INR), the function always returns the same output for the same input. Even if the process crashes midway, it can restart and recompute the same result.

2. Message-Passing Interface (MPI)

- Cloud Haskell avoids shared memory.
- All communication happens through **message passing** between processes.
- Makes it **safer** (no accidental overwriting of shared variables).

✅ **Real-time example**:

Like **WhatsApp messaging** – each conversation is independent. Even if one chat thread crashes, it doesn't affect others.

3. Concurrency Model

- Cloud applications handle **many independent requests** from users.
- Cloud Haskell ensures **safe concurrent execution** → multiple tasks can run at once without interfering.

✅ **Real-time example**:

In an **online shopping site**, thousands of users can browse, order, and make payments **simultaneously**. Each request is processed in isolation using message passing.

4. Fault Tolerance (Inspired by Erlang)

- If one process fails, **only that process is restarted**.
- Other processes remain unaffected → prevents expensive system-wide restarts.
- Achieved using Erlang-like **"let it crash" philosophy**.

✅ **Real-time example**:

In **Netflix**, if one user's video stream crashes, only that stream is restarted, not the entire Netflix service.

5. Cost Model for Communication

- Cloud Haskell introduces a **cost model** to evaluate the **communication overhead** between distributed processes.
- Helps optimize distributed execution.

✅ **Real-time example**:

If two servers (one in **India** and another in **US**) need to communicate, Cloud Haskell considers **latency + data transfer cost** before deciding how to execute tasks.

6. Serialization & Remote Execution

- Cloud Haskell allows running distributed code on **remote machines**.
- Serialization is handled **automatically** (the programmer doesn't need to worry about converting data into network-friendly format).

✅ **Real-time example**:

If you deploy a **cloud-based banking application**, some functions run in one datacenter (e.g., Mumbai), and others in another datacenter (e.g., Singapore). Cloud Haskell manages communication & serialization behind the scenes.

🔹 **Key Features of Cloud Haskell**

✓ **Fault tolerant** – one failure doesn't crash the system.

✓ **Domain-specific language** – built specifically for distributed cloud apps.

✓ **Concurrent programming** – handles thousands of user requests safely.

✓ **No shared memory** – avoids concurrency bugs.

✓ **Idempotent** – functions can restart safely.

✓ **Purely functional** – predictable, immutable state.

◇ **MultiMLton: Functional Programming**

1. What is MultiMLton?

- **MLton**: an open-source, whole-program, optimizing compiler for **Standard ML** (a functional programming language).
- **MultiMLton**: an **extension of MLton**, designed for **multiprocessor environments** (multi-core CPUs, distributed systems).
- Focus: expressing and implementing **fine-grained parallelism** (lots of small tasks executed in parallel).

✅ **Real-time example**:

Think of MLton like a **single chef** who cooks dishes one by one.

MultiMLton is like a **team of chefs in a large kitchen**, working on many small tasks (cutting, frying, mixing) in parallel to finish the meal faster.

2. Lightweight Threads

- MultiMLton can handle a **large number of lightweight threads** efficiently.
- Lightweight threads = tiny, independent units of execution (much cheaper than OS threads).

✅ **Real-time example**:

Like **WhatsApp** handling millions of chat messages at once → each chat is a lightweight thread instead of using heavy full processes.

3. Message-Passing Communication

- Threads communicate via **message passing** (not shared memory).
- Supports both **synchronous** (waiting for response) and **asynchronous** (fire-and-forget) communication.

✅ **Real-time example**:

- **Synchronous**: Sending a **bank transfer request** → you wait for confirmation.
- **Asynchronous**: Sending an **email** → you don't wait for the reply immediately.

4. Parasites (Mini Threads)

- MultiMLton introduces **parasites**:
  - Mini threads that depend on a **main/master thread**.
  - Parasites can communicate with each other through message passing.

✅ **Real-time example**:

In a **food delivery app**:

- Main thread = order management.
- Parasites = tracking delivery, updating payment, sending notifications. Each parasite depends on the main order process but does its own small task.

🔹 **Key Features of MultiMLton**

✓ Can efficiently handle **many lightweight threads**.

✓ Designed for **multiprocessor/multi-core environments**.

✓ **Deterministic concurrency** → predictable results.

✓ **Message-passing** based thread communication.

✓ Supports **synchronous & asynchronous events**.

✓ Uses **parasites (mini threads)** for modular execution.

◇ Erlang – Functional Programming

🔹 1. Introduction

- Erlang = functional + concurrent programming language.
- Developed by **Joe Armstrong** at Ericsson.
- Used in **real-time, fault-tolerant, distributed systems**.

Example: WhatsApp backend is built with Erlang → handles billions of real-time messages.

🔹 2. Concurrency Model

- Based on **Actor Model** → processes communicate via **asynchronous message passing**.
- Processes are **lightweight** (cheaper than OS threads).
- No shared memory → avoids race conditions.

Example: In an online payment system →

- One process validates payments,
- Another logs transactions,
- Another sends notifications.

  All run **independently in parallel**.

🔹 3. Fault Tolerance

- Follows **"Let it crash" philosophy**.
- If one process fails, others continue.
- Supervisors restart failed processes automatically.

Example: In WhatsApp →

- If a chat process crashes, it restarts without affecting other chats.

🔹 4. Hot Code Swapping

- Erlang allows **updating code while the system is running**.
- No downtime required.

Example: A telecom operator updates call-routing algorithms **without interrupting live calls**.

🔹 5. Memory Management

- Uses **automatic garbage collection** (runtime).
- Prevents memory leaks & manual errors.

Example: Like an automatic dishwasher cleaning memory after every process.

🔹 6. Functional & Immutable

- Erlang is **functional & declarative**.
- Variables are immutable (once assigned, never changed).
- Functions = pure (output depends only on input).

Example: In a flight booking system →

- Input = request + flight details.
- Output = confirmation or rejection.
- Immutable data prevents overwriting bookings.

◆ **7. Key Features (Quick Recall)**

✓ Fault tolerant (self-healing systems)

✓ Highly **concurrent** (millions of processes)

✓ **No shared memory** (message passing only)

✓ **Hot code swapping** (update during runtime)

✓ Automatic **garbage collection**

✓ Functional + immutable (predictable behavior)

1 CloudI

◆ **What is CloudI?**

CloudI is an **open-source cloud computing framework** developed by *Michael Truog*.

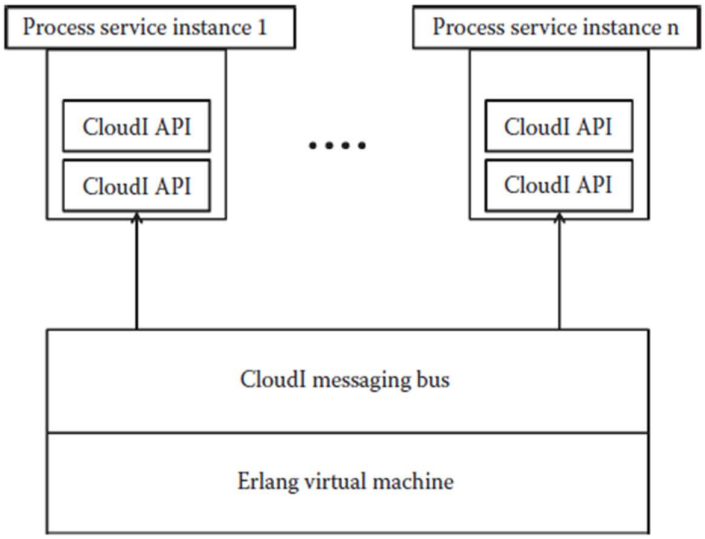
It is **different from traditional clouds** because:

- It allows building a **private cloud without virtualization**.
- It focuses on **backend server processing tasks** such as soft real-time transaction handling.
- It supports **multiple programming languages (polyglot environment)** like C, C++, Python, Java, Erlang, Ruby, etc.

At its core, CloudI is built on **Erlang**, which provides:

- **Fault tolerance**
- **Concurrency**
- **High scalability**

The **architecture of CloudI communication**.



1. **Erlang Virtual Machine (VM)**
    - This is the foundation.
    - Erlang VM ensures **fault tolerance, lightweight process handling, concurrency, and scalability**.
    - All services run on top of this VM.
2. **CloudI Messaging Bus**
    - Sits above the Erlang VM.
    - Works like a **middleware** that connects all services.
    - Provides **publish/supply** and **request/reply** communication between different service instances.
    - Example: One service publishes data, and multiple services can consume it.
3. **Process Service Instances (1 ... n)**
    - These are the **individual services running in CloudI**.
    - Each service has one or more **CloudI APIs** for communication.
    - They can be written in **different languages** (thanks to polyglot support).
    - Each service is like a **mini-cloud module**, independent but still fault-tolerant.
4. **CloudI API**
    - Provides communication between services.
    - Developers don't need to know Erlang deeply to use CloudI.
    - API ensures **asynchronous message passing** (no shared memory, only messaging).

🔹 **Key Features of CloudI**

✅ **Functional programming model** (inherited from Erlang).

✅ **Fault tolerant** – if one service crashes, others continue without failure.

✅ **Polyglot support** – services can be written in C, C++, Python, Java, Erlang, Ruby, etc.

✅ **Scalable** – can handle many services concurrently.

✅ **Asynchronous communication** – efficient for distributed systems.

🔹 **Real-Time Example**

Imagine an **online banking system**:

- **Service Instance 1**: Handles user authentication (login/logout).
- **Service Instance 2**: Handles balance checking.
- **Service Instance 3**: Handles money transfer.
- **Service Instance 4**: Handles fraud detection.

All services run independently but **communicate via CloudI APIs** over the **CloudI Messaging Bus**, on top of the **Erlang VM**.

👉 If the fraud detection service fails, the other services **still continue running without downtime**, thanks to fault tolerance.

🔑 SORCER: Object Oriented Programming

1. What is SORCER?

- **Full form**: *Service-ORiented Computing EnviRonment*.
- Developed by **Michael Sobolewski**.
- It is an **object-oriented cloud platform** for **transdisciplinary service abstractions**.
- Works on the principle of **federated Service-to-Service (S2S) metacomputing**:
    - Each **service provider** (software or tool) is a **peer** in a federation.
    - They work together dynamically for a task and then dissolve after completion.

👉 Think of it like a *temporary project team*: services join → solve problem → disband → search for another federation.

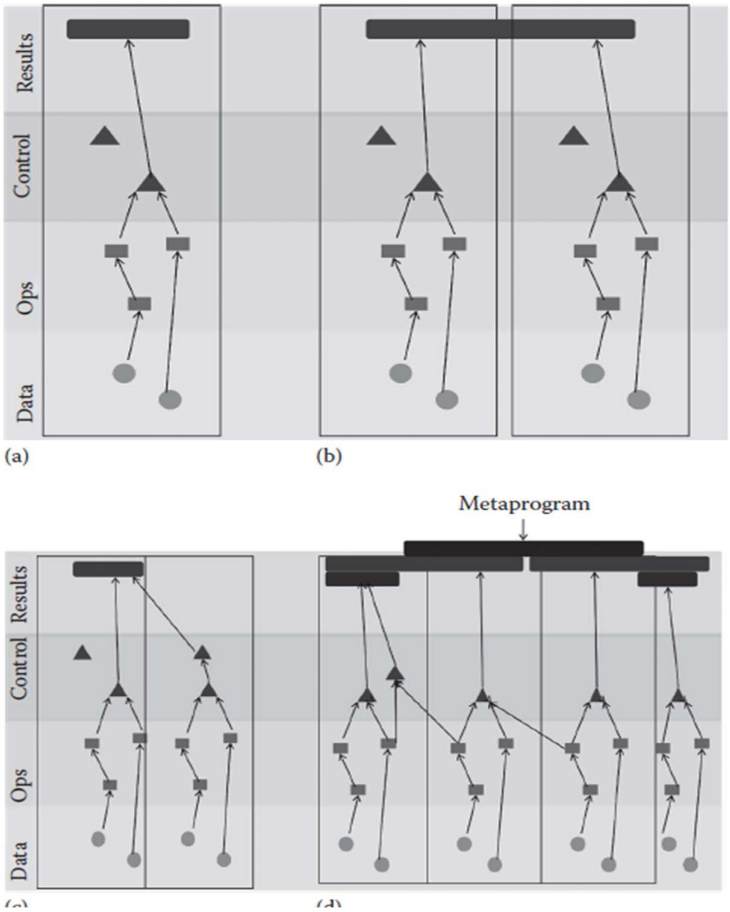
2. How Does It Work?

- Each **service** = independent **remote service provider** (self-contained).
- Services interact only via the **network** (network-centric).
- To manage them, a **metacompute OS** is used:
    - Executes **metaprograms** (instructions for collaboration).
    - Handles **time, place, order** of execution.
    - Enables **orchestration** of services.
- **Metaprograms**:
    - Special programs that **remotely control other programs**.
    - Define how services should collaborate.
- **Metainstructions**:
    - Machine-readable instructions generated from metaprograms.

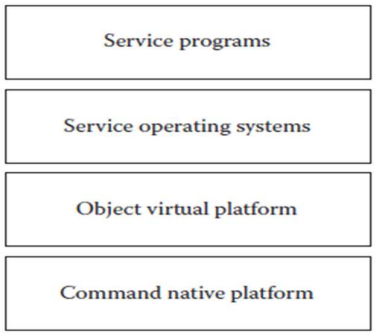
👉 Example: Instead of sending an executable simulation file, SORCER sends *metainstructions* to service providers (physics solver, CAD model, optimization tool) to run collaboratively.

4. Types of Computing in SORCER (Fig. 8.6)

- **Discipline** → Single field (e.g., Physics simulation).
- **Multidisciplinary** → Many fields, no interaction (Civil + Mechanical engineers work separately).
- **Interdisciplinary** → Fields communicate (Civil ↔ Mechanical exchange data).
- **Transdisciplinary** → Fields + external contributions + **metaprogram orchestration** (Smart City involving engineers, economists, climate scientists, sociologists).



5. Layers in SORCER



- **Platform Cloud**
  - *Command virtual platform*
  - *Object virtual platform*
- **Service Cloud**
  - *Service nodes*
  - *Domain-specific service providers*
- **Control** → Handled by *metaprocessor* (translates metaprograms into execution).
- **Service OS (SOOS)** → Manages orchestration, service federation, and user requests.

👉 User submits request → Metaprogram prepared → SOOS decides execution → Services collaborate → Results returned.

5. Deployment on FIPER

- **FIPER** = *Federated Intelligent Product EnviRonment*.
- Provides a **federation of distributed service objects**.
- Used in **engineering applications** (e.g., aircraft design).

⚡ **Real-time Example: Electric Car Design**

- **Service Providers**: CAD software, Battery simulation, Structural analysis, Cost estimator, Environmental model.
- **Federation**: Services come together for one design task.
- **Metaprogram**: Defines workflow — e.g., chassis design → battery placement → stress analysis → cost optimization → emission impact.
- **Execution**: Services run in parallel or sequence depending on the metaprogram.
- **Result**: Integrated electric car design solution.
- After work → federation dissolves → services free for another task.

📌 **Key Features of SORCER**

- ✅ Object-oriented cloud environment
- ✅ Service-to-service collaboration (S2S)
- ✅ Transdisciplinary computing model
- ✅ Fault-tolerant, flexible federations
- ✅ Uses **metaprograms** for orchestration
- ✅ Supports **high-performance computing**
- ✅ Autonomic resource management

Programming Models in Aneka

Aneka is a PaaS developed by Manjrasoft, Inc. There are three programming models defined and used by Aneka, which are discussed in the following.

1 Task Execution Model

In this model, the application is divided into several tasks. Each task is executed independently. As defined by the developers of Aneka, tasks are work units that are executed by the scheduler in any order. As it involves a large number of tasks, it is very much suitable for distributed applications, specifically applications where several independent results are involved, and these independent results are then combined to give a final result. For these kinds of problems, the independent results can be considered as output from the tasks, and later these results can be combined by the user. It also supports dynamic creation of tasks .

Key features

- Suitable for distributed application
- Independent tasks

2 Thread Execution Model

In thread model, the applications are executed using processes. Each process consists of one or more threads. As defined by the developers of Aneka, threads are a sequence of instructions that can be executed in parallel with other instructions. Thus, in thread execution model, the execution is taken care by the system [10].

Key features

Supports heavy parallelization.

Multiple thread programming is available.

3 Map Reduce Model

This model is similar to the actual MapReduce model described in the first section. The Map Reduce model is implemented in the Aneka platform.

New Programming Models Proposed for Cloud

1. Why New Programming Models are Needed?

- Traditional programming models (like thread-based or object-oriented models) were not created with cloud in mind.
- These old models work well for single machines or small networks but face challenges in **cloud environments**, which are:
  - **Massive in scale** (millions of users).
  - **Highly distributed** (many servers across the globe).
  - **Shared by multiple tenants/users** (multitenancy).
- Limitation of old models:
  - They preserve certain basic structures and cannot be drastically changed.
  - Only small modifications (like adding parameters or minor tweaks) are possible.
- Because of these drawbacks, **completely new models** had to be designed for cloud computing.

2. Features of Cloud-Aware Programming Models

These models are built from scratch and keep cloud characteristics at the core:

1. **Scalability**
   - Applications should handle growth automatically (more users, more data).
   - Example: A cloud app serving 1,000 users should scale to serve 1 million users.

2. **Distributed Nature**
    - Algorithms must work across multiple machines and data centers.
    - Data and tasks are divided and processed in parallel.
3. **Multitenancy**
    - Many users and organizations share the same cloud infrastructure.
    - Programming models ensure security, isolation, and fairness in resource use.
4. **Elasticity & Fault Tolerance** (often included)
    - Resources scale up or down as needed.
    - If one machine fails, the system continues to work smoothly.

3. Advantages of New Programming Models

- **No restrictions from old language rules** → designers have full freedom.
- Models can be optimized directly for distributed computing.
- Allow developers to **focus on algorithms** instead of worrying about hardware setup.
- More efficient in handling **big data** and **large-scale applications**.

4. Examples of Cloud Programming Models

Some well-known models and approaches include:

- **MapReduce (Google)**
    - Splits big data into smaller tasks → processes them in parallel → combines results.
    - Best for large-scale data processing (e.g., search engines, analytics).
- **Dryad (Microsoft)**
    - A graph-based model for distributed computation.
    - Tasks are represented as nodes in a graph, making workflow management easier.
- **Cloud Haskell**
    - Extends Haskell language to support distributed cloud systems.
    - Focus on functional programming in the cloud.
- **Serverless Programming Models (e.g., AWS Lambda, Azure Functions)**
    - Developers write functions, and the cloud automatically runs them when needed.
    - No need to manage servers manually.

1. Orleans Framework

Orleans is a framework developed by **Microsoft** for building **cloud-based applications**. It aims to simplify the development of scalable, reliable, and concurrent distributed systems by providing a **virtual actor model** abstraction. Orleans facilitates development on both the **client** and **server** sides of cloud applications, supporting a **concurrent programming paradigm**.

Key Features

Orleans has several characteristics designed specifically for cloud environments, including:

- **Concurrency:** Enables parallel processing and message-driven execution.
- **Scalability:** Automatically distributes workloads across servers and scales out easily.
- **Reduced Errors:** Simplifies concurrent programming, minimizing common multi-threading errors.
- **Security:** Integrates with secure cloud identity and access management mechanisms.

Core Concept — Grains

The foundation of Orleans is based on the concept of **grains**, which are **logical computational units** or **actors**. Grains act as the **building blocks** of the framework:

- Each grain is **independent and isolated**.
- All computation within Orleans is executed through grains.
- Grains can have multiple **activations**—these are instances of grains running on physical servers.

Activations allow multiple processes to execute **concurrently**, thereby increasing system **throughput** and **reducing latency**. However, each grain handles **one request at a time**, ensuring internal thread safety and avoiding race conditions.

State Management and Consistency

Each grain maintains a **state**, representing its current data or context. When multiple activations of the same grain exist, maintaining consistent state becomes challenging. Orleans addresses this through built-in **state management and synchronization mechanisms**, ensuring data consistency across all activations.

Communication

Grains communicate with each other **asynchronously via message passing**. This model simplifies concurrency, avoids shared-memory issues, and enhances the reliability of distributed systems.

Scalability and Fault Tolerance in Orleans

Scalability is a critical concern in the development of cloud-based applications. The Orleans framework places strong emphasis on scalability and provides several mechanisms to address it. The fundamental concepts of **grains** and **activations** inherently support scalability by allowing distributed and concurrent computation. In addition, Orleans enhances scalability through the use of a **sharded database architecture**.

Sharded Database Architecture

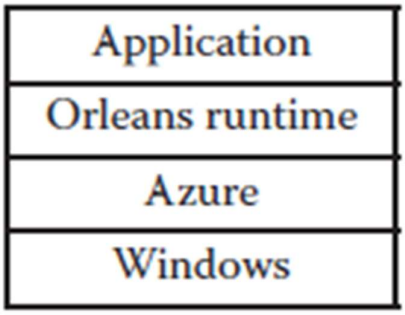
A **sharded database** is a **horizontally partitioned** database system where data is divided into multiple components known as **shards**. Each shard contains a subset of the total dataset (rows) and

operates independently. Orleans uses this sharded architecture in its backend to support grain operations efficiently.

This design enables each grain to function independently within its own **grain domain**, allowing parallel and isolated computation. By distributing both computation and data across multiple shards, Orleans avoids the **bottleneck** associated with a single centralized database. In contrast to monolithic systems where a single database failure can bring down the entire application, Orleans' distributed architecture localizes failures—affecting only a portion of the system while the rest continues to operate normally.

This results in a system that is **fault-tolerant**, **highly available**, and **scalable**. Because each grain and activation is independent, there is no restriction on the number of grains or activations that can be created. As the number of requests or processes increases, the system can dynamically scale by creating more grains and activations, without interference between them.

Thus, the **concept of grains and activations itself forms the foundation for Orleans' scalability solution.**



The framework is implemented as a **C# library built on top of the .NET framework** .

Transactions and Data Consistency

In addition to scalability, Orleans introduces **transactional mechanisms** to maintain data **consistency** and **integrity** across grains. Transactions are essential for several reasons:

1. **State Management:** Since activations can modify the state of grains, a mechanism is required to ensure that all changes remain consistent across concurrent operations.
2. **Data Replication:** Data replication is a common requirement in cloud systems for performance and availability. However, improper replication can lead to inconsistencies. Orleans manages this through controlled transactions that maintain synchronized and reliable data across replicas.

Restricted Concurrency

Although Orleans supports concurrent execution at the system level, it introduces **restricted concurrency** within grains. This means that each grain processes only one request at a time, preventing race conditions and ensuring thread safety.

Key Features

- **Fault tolerance**
- **Sharded database support**
- **Efficient transaction handling**
- **Restricted concurrency**
- **Security**

2 BOOM and Bloom

Berkeley Orders of Magnitude (BOOM) is a research project developed at the **University of California, Berkeley**, with the goal of creating a **programming framework specifically designed for cloud computing**. The project aims to enable developers to build **large-scale distributed systems** more easily and efficiently.

To support this framework, UC Berkeley researchers developed a new programming language called **Bloom**. Bloom provides a **data-centric and distributed programming model**, suitable for highly parallel and scalable cloud systems.

Nature and Objectives of BOOM

- **BOOM** stands for **Berkeley Orders of Magnitude**.
- It is **highly distributed** and **unstructured** in nature.
- The main goal is to help developers design **large, cloud-scale systems** with **less code** and **high scalability**.
- BOOM focuses on **data-driven and disorderly execution** rather than strict control-flow–based programming.
- It supports **fully distributed computation**, ensuring that applications can scale seamlessly across clusters of machines.

Example:

Suppose a developer wants to process huge amounts of sensor data collected from thousands of IoT devices. Using BOOM, they can design a distributed data-processing system that automatically scales across multiple servers, minimizing the need to manually manage communication or coordination.

BOOM File System (BOOM FS)

BOOM uses a custom distributed file system called **BOOM FS**, which is built upon the **Hadoop Distributed File System (HDFS)**.

- BOOM FS is designed to handle **large-scale data storage and retrieval** more efficiently.
- According to UC Berkeley's analysis, **BOOM FS performs about 20% better than standard HDFS** in data throughput and reliability.

Bloom Programming Language

Bloom is the programming language developed for the BOOM framework. It is designed to make writing **distributed and parallel applications** easier.

Design Goals of Bloom

1. **Familiar Syntax:**
   Bloom uses a syntax similar to common imperative languages (like Python or Java), making it easy for developers to learn.
2. **Integration with Imperative Languages:**
   Bloom can be embedded into other programming environments, allowing hybrid systems that use both imperative and declarative styles.
3. **Modularity, Encapsulation, and Composition:**
   Supports modular code design so that large systems can be built from smaller, reusable components.

Underlying Principles and Technologies

Bloom is based on two key concepts:

1. **CALM Principle (Consistency as Logical Monotonicity):**
   - This principle is used to **analyze and reason about coordination and consistency** in distributed systems.
   - According to CALM, if a program is *monotonic* (i.e., adding more data doesn't change earlier results), then it can execute without coordination and still produce consistent results.
   - This helps automate the detection of which parts of a program need synchronization.

Example:

If a program only adds data (like counting events) and never removes it, its results remain consistent regardless of message order — hence, no coordination is needed.

2. **Dedalus Language:**
   - Bloom is **built using Dedalus**, a **temporal logic-based language**.
   - Dedalus focuses on **time-oriented** data representation rather than space-oriented data.
   - It treats computations as events that happen over time, which fits well with the asynchronous nature of distributed systems.
   - Because Dedalus is a **pure temporal logic language**, programmers do not need to understand the low-level interpreter or compiler behavior.

Key Features of BOOM and Bloom

- **Modularity, encapsulation, and composition** — supports structured program organization.

- **Extensively distributable** — designed for fully distributed execution across many nodes.
- **Unstructured framework** — flexible and adaptable for large-scale systems.
- **Data-centric and time-oriented** — focuses on data flow and temporal relationships rather than control flow.
- **Improved performance** — BOOM FS provides about **20% better throughput** compared to HDFS.

3. GridBatch

GridBatch is a **parallel programming framework** developed by **Liu and Orban** that helps programmers **convert high-level application designs into parallel implementations** suitable for **cloud, grid, and cluster computing environments**.

Unlike systems that automatically parallelize programs, **GridBatch requires the programmer** to understand the **application logic** and decide **how to divide tasks** for parallel execution. This gives developers **more control** over the distribution of work and resource management.

The main goal of GridBatch is to make it **easier to design parallel applications** without needing to write complex low-level code for parallelization.

It provides **libraries and operators** that simplify:

- Breaking a large application into smaller parallel tasks
- Distributing these tasks across multiple computing nodes
- Collecting and merging results efficiently

Working Mechanism

1. **Task Partitioning:**
    - The application is divided into smaller, **independent tasks** that can be executed simultaneously.
    - For example, in a data analysis program, each task might analyze a different subset of data.
2. **Library Support:**
    - GridBatch provides **predefined libraries and operators** that help the programmer manage these tasks easily.
    - The libraries handle task scheduling, synchronization, and data communication between nodes.
3. **Execution:**
    - These tasks are then executed across a distributed cloud environment (or grid/cluster).
    - Results from all nodes are collected and combined to form the final output.

4. **Performance Optimization:**
    - Since programmers control the partitioning, they can design the system to achieve **maximum efficiency and scalability**, especially when handling **large datasets**.

Example Scenario

Suppose you are running a **large statistical analysis** on customer behavior data with 10 million records.

In **GridBatch**, you can:

- Divide the dataset into 10 smaller chunks (each 1 million records).
- Assign each chunk to a separate worker node (parallel task).
- Each worker processes its data independently (e.g., calculates average spending).
- The results are merged to produce the final overall statistics.

Advantages

- Increases **developer productivity** by simplifying parallel programming.
- Provides **high scalability** for large analytical workloads.
- Offers **manual control** over task division for **optimized performance**

4.Simple API for Grid Applications (SAGA)

SAGA (Simple API for Grid Applications) is a **high-level programming interface** designed to make it easier to develop and run **distributed and grid-based applications**.

It was developed by **Goodale et al.**, and its main goal is to provide **a simple, standard, and uniform interface** to access distributed computing resources — **without needing to understand the low-level complexities** of grid systems.

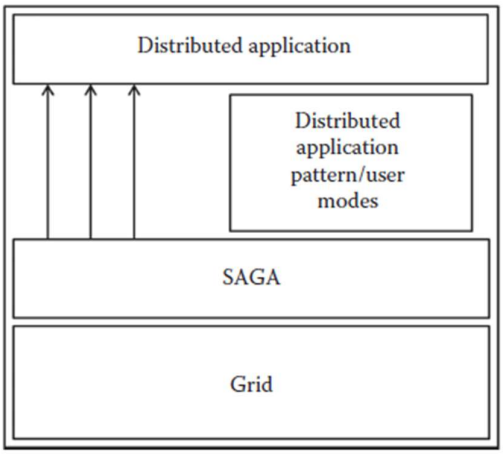
When programmers build distributed systems (like cloud or grid applications), they often need to:

- Manage resources across multiple systems
- Handle communication between nodes
- Control concurrent processes

SAGA simplifies this by offering **a standard API** that can work across different grid toolkits like **Condor** and **Globus**.

Architecture Overview

SAGA sits **above existing grid toolkits** and **below user applications**:



✅ **SAGA is not middleware** —

Unlike middleware (like OGSA), SAGA doesn't manage the grid itself; it simply **provides APIs** to use grid services easily.

Programming Model

SAGA is built using **C++**, with support for **C** and **Java** languages. It provides:

- A **Task Model** → for concurrent execution of tasks.
- An **Asynchronous Notification Mechanism** → for managing parallel tasks and receiving status updates.

These allow developers to easily write **concurrent and distributed programs**.

Concurrency in SAGA

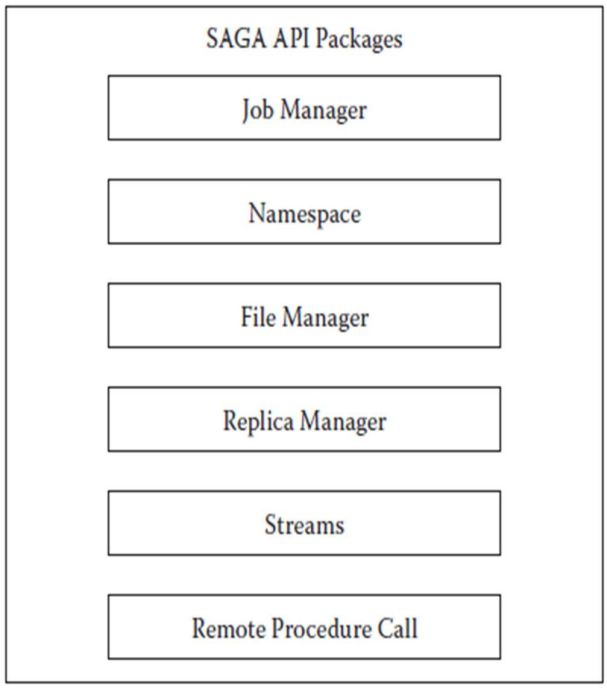
SAGA allows **concurrent units** (tasks) to **share object states**. However:

- It **does not enforce concurrency control** automatically.
- Programmers must handle synchronization manually.

👉 This design choice avoids unnecessary complexity and overhead, giving developers flexibility to choose how to manage concurrency.

SAGA (Simple API for Grid Applications) provides several **core packages** that help manage distributed and grid applications efficiently.

These packages simplify job submission, file management, and communication across multiple grid nodes.



1. Job Management

Used for **submission, control, and monitoring** of jobs running on the grid.

Functions:

- Submit jobs to grid resources
- Monitor running and completed jobs
- Control job execution (start, stop, resume)

Modes of submission:

- **Batch Mode:** Job runs automatically after submission
- **Interactive Mode:** User can interact with running job

Example:

A researcher uses SAGA to submit 100 simulations at once to grid servers and monitor their status using the job management package.

2. Namespace Management

Handles the organization and identification of files or objects in the grid system (like folders in a filesystem).

Works with: File Management package.

Example:

If a dataset is stored across multiple grid nodes, namespace management ensures it can be accessed using a **single logical name**,

3. File Management

Manages reading, writing, and accessing files in distributed storage.

Applications can access files **without knowing their actual physical location**.

Example:

A program can open and read data.txt from SAGA's file system — whether it's on a local machine or remote grid node.

4. Replica Management

Maintains **copies (replicas)** of important files to improve performance and reliability.

Functions:

- Create and maintain replicas
- Search logical files based on metadata

Example:

If a file result.csv is stored in multiple grid locations, replica management ensures consistency across all copies and retrieves the nearest or available one when needed.

5. Streams Management

Provides **secure, authenticated socket connections** for communication between distributed components.

Example:

A monitoring application streams live log data from remote grid nodes using authenticated connections for safety.

6. Remote Procedure Call (RPC)

Allows one grid node to **execute a function** or **method** on another remote node.

Example:

A node can request another node to execute computeResult() remotely and return the output.