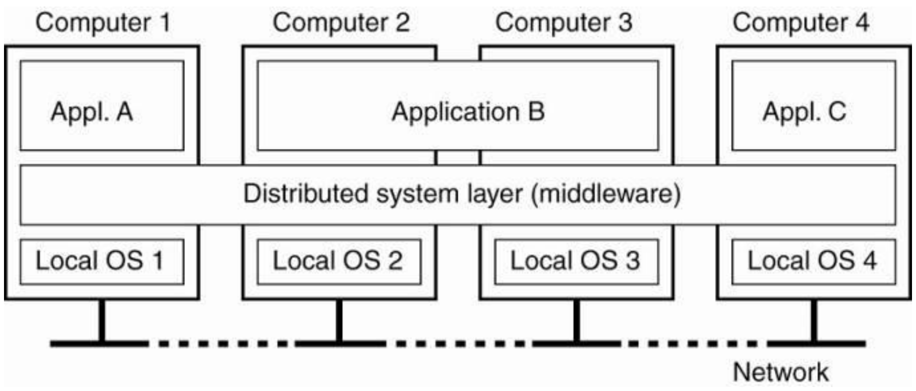
CHAPTER 01

1. DEMONSTRATE DISTRIBUTED COMPUTING SYSTEM

A distributed computing system is a **collection of independent computers that are connected by a network and work together to solve a common problem**. Each computer in the system has its own **local memory and processing capabilities,** and they **communicate with each other by sending and receiving messages.**

Distributed computing systems are often **used to solve problems that are too large or complex for a single computer to handle.** For example, distributed computing systems are used to power **large search engines, social media platforms, and scientific computing applications.**

Simple Architecture of DCS:



The system consists of four computers, 1, 2, 3 and 4. Each node has its **own local memory and processing capabilities.** The nodes are connected to each other via network, which **allows them to communicate with each other by sending and receiving messages.**

Let us consider above system is designed to solve a problem that requires a lot of processing power, such as **rendering a complex 3D image.** The problem is divided into smaller sub-problems, which are then assigned to the different nodes in the system. Each node works on its assigned sub-problem independently. Once a node has finished working on its sub-problem, it sends the results to the other nodes in the system.

The other nodes then combine the results from all of the sub-problems to produce the final solution. The final solution is then sent back to the user.

Advantages of Distributed Computing Systems:

1. **Scalability:** If a task is too big for one computer, we can connect more computers to distribute the work.
2. **Fault Tolerance:** If one computer stops working, the others keep going, making sure the system doesn't break.
3. **Efficiency:** Solving problems by breaking them into smaller parts and working on them simultaneously is indeed more efficient.

Distributed computing systems are a powerful tool for solving a wide range of problems. They are used in many different industries, including **science, engineering, business, and entertainment.**

Here are some examples of distributed computing systems:

1. **The World Wide Web:** The World Wide Web is a distributed system of **web servers and clients.** When you want visit a website, your web browser sends a request to the web server that hosts the website. The web server then sends the requested web page back to your browser.
2. **Cloud computing platforms:** Cloud computing platforms, such as **Amazon Web Services (AWS) and Microsoft Azure,** are distributed systems that provide a variety of computing services, such as **storage, processing, and networking.**

Distributed computing is a rapidly growing field, and new applications for distributed systems are being developed all the time.

2. CLASIFY THE TYPES OF TRANSPERENCY IN DISTRIBUTED SYSTEM.

Transparency in a distributed system refers to the **ability of the system to hide its complexities and present a unified interface to users and applications.** It **aims to make the distributed system appear as a single, cohesive system** rather than a collection of interconnected components.

Types of Transparency in Distributed Systems:

1. **Access Transparency:** Users and applications can access distributed resources without needing to know **how to locate or interact** with them. It **provides a consistent interface for accessing resources** regardless of their physical location.
2. **Location Transparency:** The **physical location of resources is hidden** from users and applications. They can access resources using a **logical name or identifier** without needing to know where the resource is located.
3. **Migration Transparency:** Resources can be **moved or migrated** between different nodes in the distributed system **without affecting the availability or accessibility** of those resources. Users and applications are unaware of the migration process.
4. **Relocation Transparency:** Similar to migration transparency, this type of transparency allows resources to be moved to different locations in the distributed system. However, **users and applications may experience a brief interruption in accessing the resource during the relocation process.**
5. **Replication Transparency**: Multiple copies of a resource can be maintained across different nodes in the distributed system. Users and applications can access the resource without being aware of its replication.
6. **Concurrency Transparency:** Concurrent access to shared resources is managed transparently by the distributed system. Users and applications can access shared resources without needing to handle **issues related to concurrent access.**
7. **Failure Transparency:** The distributed system can handle **failures of individual components or nodes without impacting the overall availability or functionality** of

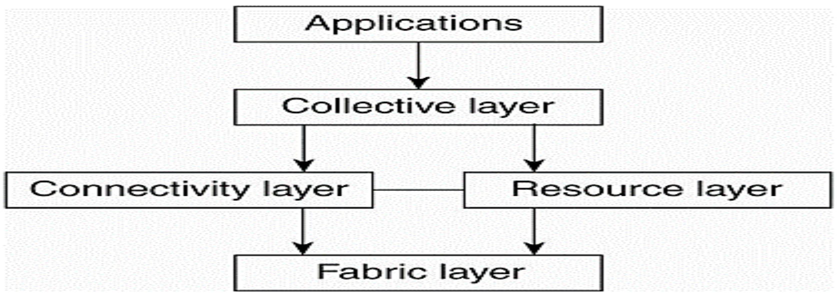
the system. Users and applications are shielded from the details of failures and the system continues to operate seamlessly.

These types of transparency collectively **aim to simplify the interaction with a distributed system,** making it easier for users and applications to utilize the resources and services provided by the system.

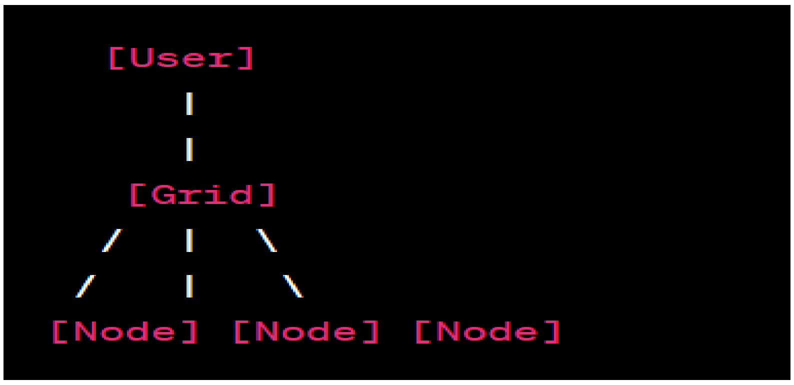
3. EXPLAIN GRID COMPUTING SYSTEM.

Grid computing is a distributed computing system where multiple computers, often geographically dispersed, are connected to work together as if they were a single, unified resource. It's designed to solve large-scale computational problems that may involve significant processing power or data storage.

Here's a basic architecture of grid computing system:



Here's a basic diagram to illustrate how it works:



Components:

1. **User:** This represents the **person or an application** who needs to perform some complex computation tasks.
2. **Grid:** The **grid is the main infrastructure that connects various computing resources.** It acts as a middle layer that **coordinates tasks across multiple nodes.**
3. **Nodes:** Nodes are **individual computers or servers** that are part of the grid. They **contribute their processing power, storage, and other resources** to the grid. Nodes can be located at different geographical locations.

How Grid Computing Works:

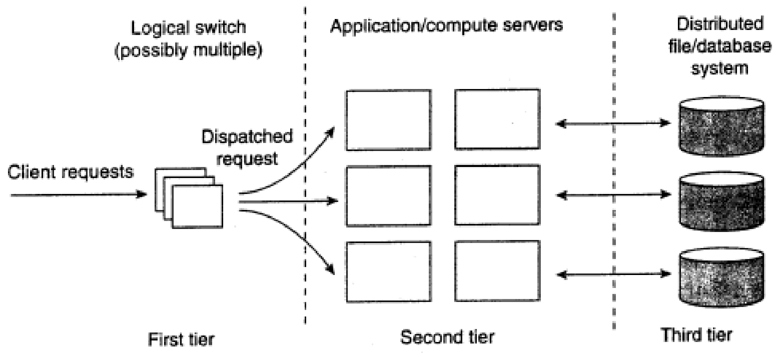
1. The user **submits a computational task to the grid.**
2. The grid infrastructure, which **includes specialized software and middleware,** takes care of breaking down the task into smaller sub-tasks.
3. These **sub-tasks are then distributed to different nodes in the grid.**
4. Each node **independently processes its assigned sub-task** using its own computing resources.
5. **Results from the sub-tasks are collected and combined** by the grid infrastructure.
6. The final result is sent back to the user.

Advantages of GRID computing system:

- Grid computing is **highly scalable because** it can add or remove nodes as needed to handle varying workloads.
- Frequently utilized in **scientific research, data analysis, and resource-intensive simulations.**
- Grid computing can **make efficient use of resources** by harnessing **idle computing power** from various sources.
- Grid computing **involve security and access control mechanisms** to protect data and resources.

4. EXPLAIN GENERAL ORGANIZATION OF A THREE - TIERED SERVER CLUSTER.

The general organization of a three-tiered server cluster involves the **logical structuring of machines connected through a network, where each machine runs one or more servers.**



The three tiers are as follows:

1. **First Tier:**
    - This tier **includes a logical switch through which client requests are routed.**
    - The switch can vary in functionality, such as **transport-layer switches** that **handle incoming TCP requests.**
    - The switch serves as the **entry point for the server cluster.**
    - For **scalability and availability,** a server cluster must have **multiple access points.**

2. **Second Tier:**

- The second tier consists of **application servers dedicated to processing client requests.**
- These **servers run on high-performance hardware** to deliver compute power.
- Depending on the application, servers in this tier may need to handle specific tasks, such as **media processing** in the case of **media streaming applications.**

3. **Third Tier:**

- The third tier comprises **data-processing servers,** particularly **file and database servers.**
- These servers **handle storage-related tasks** and run-on specialized machines **configured for high-speed disk access**.

The first tier, represented by the switch, plays a crucial role in maintaining **access transparency.** It hides the fact that there are multiple servers, providing a **single access point** for client applications. **Load balancing implemented at this tier to distribute requests among the various servers** with policies like **round-robin.**

5. DISTRIBUTED INFORMATION SYSTEM

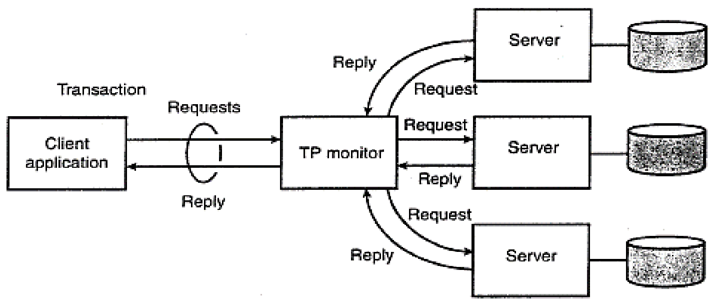
A distributed information system is a type of system **used by organizations to manage and share data across different locations and applications.** This approach is often **adopted when businesses struggle with seamless integration of networked applications.**

Two main levels of integration are commonly seen in these systems:

1. **Low-Level Integration:**
Low-Level integration allow grouping multiple requests into a larger one and executed as a single transaction. The key idea is that either all requests are executed, or none of them.

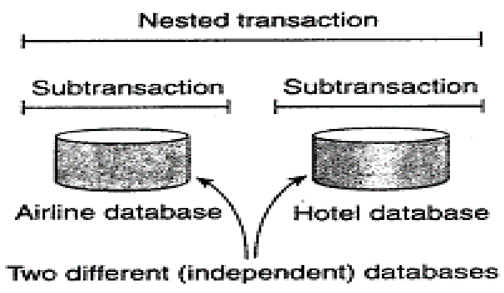
2. **High-Level Integration:**
In complex applications with separate parts, high-level integration (EAI) allows **direct communication between applications to share information.**

Transaction Processing Systems:



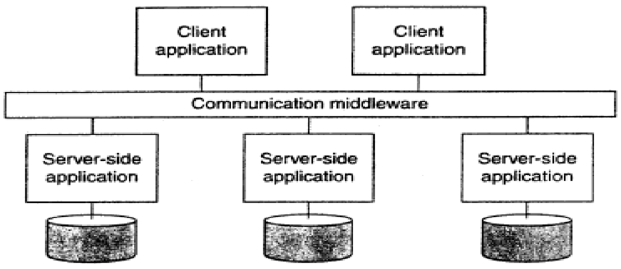
- Concentrating on database applications, transactions are essential. Transactions involve operations on a database and follow the ACID properties (Atomicity, Consistency, Isolation, Durability). These properties ensure that transactions happen completely or not at all, maintain system invariants, prevent interference between concurrent transactions, and make changes permanent after committing.

Nested Transactions:



- Nested transactions involve a **top-level transaction that may have parallel sub transactions,** potentially **running on different machines.** This **nesting allows for the distribution of a transaction across multiple machines.**

Enterprise Application Integration (EAI):



- **As applications become more independent from the databases** they use EAI. It focuses on **allowing different application components to communicate directly,** rather than **relying solely on transaction processing. Middleware** systems play a significant role in facilitating this communication, enabling applications to **exchange information.**

In summary, a distributed information system involves **managing data across different applications and locations.** It **addresses challenges in making networked applications work together efficiently,** from simple transaction processing systems to more advanced enterprise application integration.

6. EXPLAIN VARIOUS DECENTRALIZED ARCHITECTURES

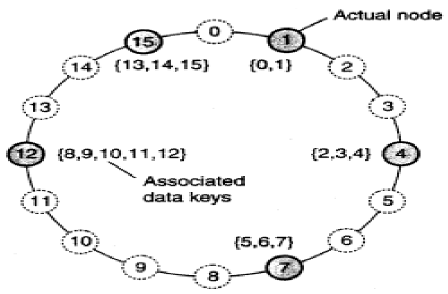
Decentralized architecture refers to a **system design** where **decision-making and control are distributed among multiple nodes** rather than being concentrated at central authority.

This approach promotes **resilience, fault tolerance, and scalability,** as **no single point of failure exists,** and system can **adapt to changes or disruptions more effectively.**

In the context of decentralized architectures, one prominent type is **Peer-to-Peer (P2P) systems.** P2P systems **allow nodes** (individual computers or devices) **to communicate and share resources directly with each other, without the need for a central server.**

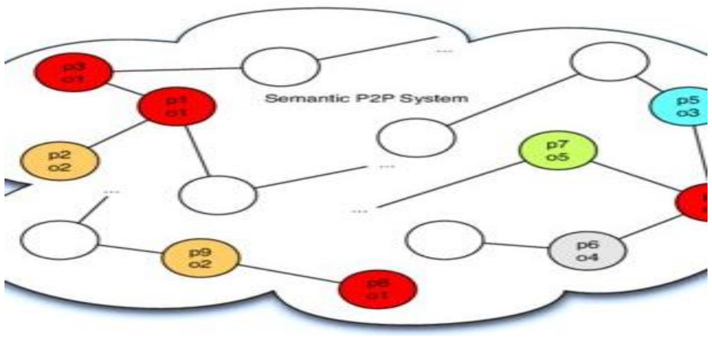
Here are various types of decentralized architectures within the realm of P2P systems:

1. **Structured P2P:**



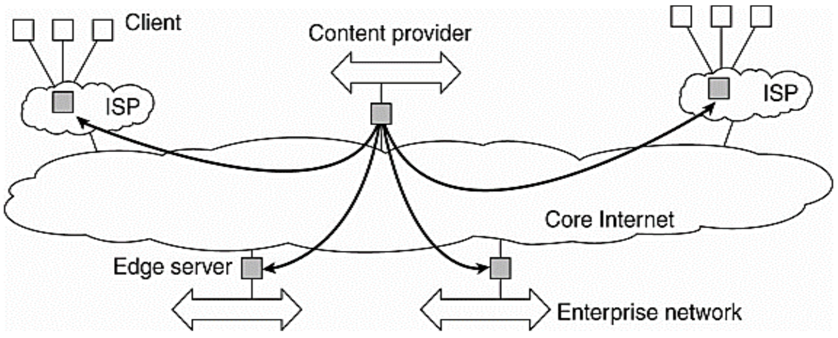
- Nodes are organized according to a specific **distributed data structure,** often called as **Distributed Hash Table (DHT).**
- Examples include **Chord, Kademlia, and Pastry.**

2. **Unstructured P2P:**



- Nodes **do not follow a specific organization or structured pattern.**
- **Nodes have randomly selected neighbours,** and the network may **lack a systematic arrangement.**
- Examples include early versions of **Napster and Gnutella.**

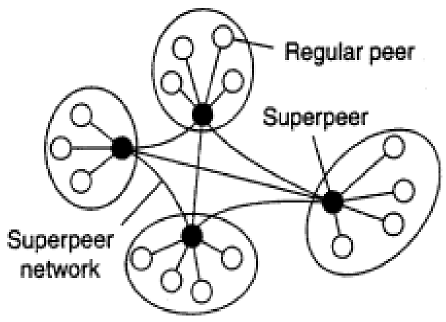
3. **Hybrid P2P:**



AFTAB-1405

- Combines elements of both structured and unstructured approaches.
- Some nodes are in an **organized manner,** while others **connect randomly.**
- Aims to **balance the efficiency of structured systems** with the **flexibility of unstructured systems.**

4. **Super Peers:**



- Super peers, also known as **ultrapeers,** are nodes within a P2P network that take on additional responsibilities.
- They often **act as intermediaries,** helping with tasks like **indexing and searching, enhancing the efficiency of the network.**
- Super peer architectures **can be a part of both structured and unstructured** P2P systems.

In summary, decentralized architectures, particularly within the realm of P2P systems, offer diverse approaches to **organizing and managing nodes** in a network.

7. EXPLAIN ALTERNATIVE CLIENT-SERVER ORGANIZATION.

In client-server architectures, where processes are divided into clients and servers, there are various ways to **distribute software components across machines.**
Here are some common alternatives:

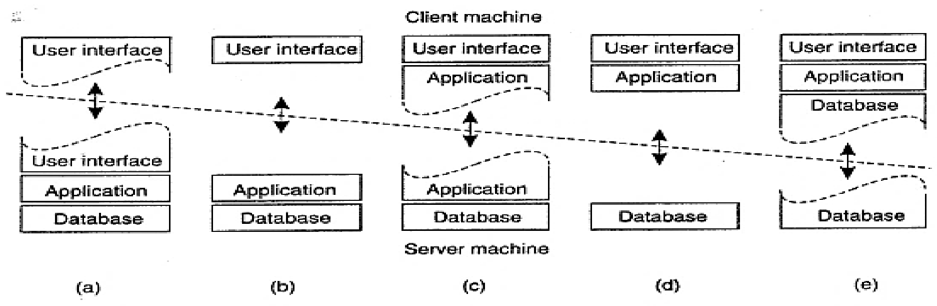


Figure 2-5. Alternative client-server organizations (a)-(e).

1. **Terminal-Dependent User Interface on Client:**
    - In this setup, **only the terminal-dependent part of the user interface** is on the client machine.
    - **Applications remotely control how data is presented.**
    - The **client acts as a display terminal,** and most processing occurs on the server.

2. **Full User Interface on Client:**
    - The **entire user interface software is on the client side.**
    - The **application is divided into a front end on the client** and the **remaining functionality on the server.**
    - Front end communicates with the rest of the application **through an application-specific protocol.**
3. **Partial Application on Client:**
    - **Part of the application is moved to the client,** which can be useful for tasks like **validation** before processing on the server.
4. **Client-Heavy Processing with Server Data Operations:**
    - Most of the application runs on the client, **but operations on files or database entries go to the server.**
    - Common in situations where the **client machine is connected to a distributed database system.**
5. **Client-Side Data Cache:**
    - Similar to the previous setup, but the **client's local disk contains part of the data.**
    - **Example:** Web browsing, where the **client builds a cache of recently visited web pages on the local disk.**

In summary, these alternative client-server organizations offer different trade-offs in terms of **processing distribution, user interface placement,** and **data management in distributed systems.**

CHAPTER 02
1. WHAT IS VIRTUALIZATION? EXPLAIN DIFFERENT ARCHITECTURES OF VIRTUAL MACHINES.[UNIT 2]

Virtualization is a technology that **allows multiple virtual machines (VMs) to run on a single physical server. Each VM has its own operating system** and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware.

Virtualization is achieved through the use of a software layer called a hypervisor. The hypervisor sits between the physical hardware and the VMs, and it is responsible for **managing the resources** of the physical server and **allocating them to the VMs.**

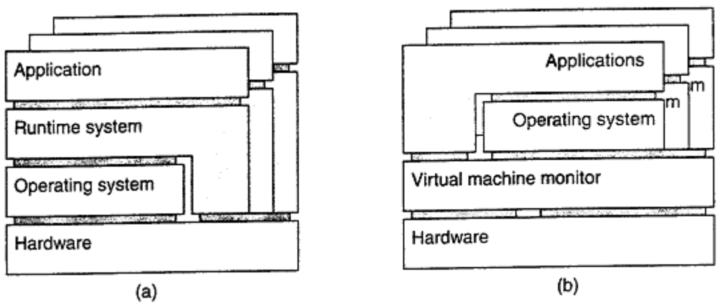


Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor. with multiple in-

There are two ways to virtualize:

1. **Process Virtual Machine:**

   - Creates a **runtime system with an abstract instruction set** for executing applications.
   - This runtime system mimics the behaviour of interfaces for a **single process.**
   - **Examples:** Java runtime environment or emulating Windows applications on UNIX.

2. **Virtual Machine Monitor (VMM):**

   - Implements a layer, **shielding original hardware,** offering the **complete instruction set as an interface.**
   - Allows **multiple, different operating systems to run independently** on the same platform.
   - Enhances **reliability and security** by isolating applications and environments.
   - **Examples: VMware and Xen.**

In essence, virtualization aims to mimic these interfaces to **provide flexibility and isolation in running applications and operating systems.**

Virtualization has a number of benefits, including:

- **Resource utilization:** Virtualization allows multiple VMs to run on a single physical server, which can significantly improve resource utilization.
- **scalability**: Virtual machines can be easily **scaled up or down** to meet changing demand.
- **flexibility: Virtual machines can be moved between physical servers without disruption.**
- **fault tolerance:** If a physical server fails, the virtual machines running on it can be **restarted on another server,** which can help to improve the fault tolerance of distributed systems.

2. WHAT IS CODE MIGRATION? EXPLAIN MODELS FOR CODE MIGRATION.[UNIT 2]

Code migration refers to the **process of moving programs or parts of programs between different machines in a distributed computing environment.** Instead of just passing data between systems, code migration involves **transferring executable code,** and in some cases, the **state of a running program.** The goal is often to optimize **system performance, improve flexibility, and manage resources more efficiently.**

Reasons for Code Migration:

- **Performance Improvement:** Moving processes from heavily-loaded to lightly-loaded machines to enhance overall system performance.
- **Minimizing Communication:** Reducing the need for data transfer over the network by executing code closer to where the data resides.
- **Flexibility: Allowing computer code to move around different computers** so that distributed systems can be easily set up and adjusted.

Models for Code Migration:

1. **Weak Mobility:** Involves transferring only the code segment and possibly some initialization data. The **program starts from predefined positions.**
2. **Strong Mobility:** Strong mobility **allows a program to pick up where it left off,** even when it moves to a new computer or device, ensuring a **smooth transition without loss of data or progress.**
3. **Sender-initiated Migration: Migration is initiated at the machine where the code resides or is being executed.** Common when uploading programs to a compute server.
4. **Receiver-initiated Migration:** The target machine takes the initiative for code migration. For example, **Java applets are downloaded by the client.**

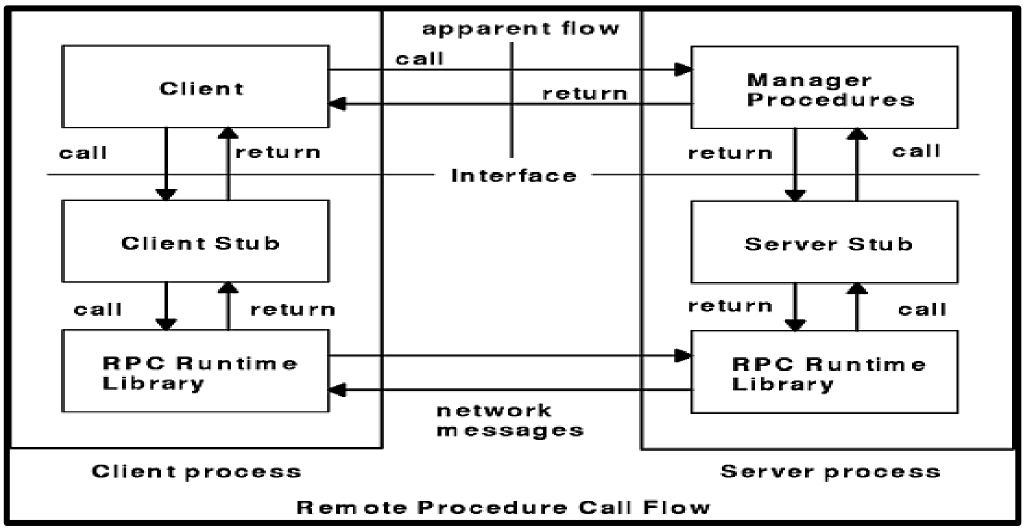
Dealing with Local Resources During Migration:

- **Binding by Identifier:** This is the **strongest connection.** A **process refers to a resource by its name.** It can involve **moving the resource, creating global references,** or sometimes **changing the connection.**

- **Binding by Value:** Weaker binding where the value of a resource is needed. Actions depend on whether the resource is fixed, fastened, or unattached.
- **Binding by Type:** Weakest binding, and the solution is to rebind the process to a locally available resource of the same type.

In summary, code migration is the movement of executable code between machines, driven by performance optimization, flexibility, and resource management. Different models and strategies are employed to address various aspects of the migration process.

3. ILLUSTRATE REMOTE COMPUTATION THROUGH RPC / DESCRIBE REMOTE PROCEDURE CALL IN DISTRIBUTED SYSTEM. [UNIT 2



Remote procedure call (RPC) is a programming technique that allows a program to call a subroutine or function that is located on another computer in a network without having to know the details of the underlying network communication. RPC is used to implement distributed systems, where multiple programs are running on different computers and need to communicate with each other.

How does RPC work?

1. RPC works by using a software layer called as RPC stub. The RPC stub is responsible for handling the details of the network communication.
2. When a program makes RPC call, it passes the parameters to the client stub. The client stub then marshals the parameters into a message and sends it to the server stub on the remote computer(Server Process).
3. The server stub on the remote computer un marshals the parameters and calls the remote procedure/manager procedure. The manager procedure executes and returns the results to the server stub.
4. The server stub on the remote computer then marshals the results into a message and sends it to the client stub on the client computer. The client stub on the client computer un marshals the results and returns them to the client program.

Why is RPC important for distributed information systems?

1. RPC is important for distributed information systems **because it allows programs to communicate with each other and access shared information** in a **transparent** and **efficient way.**
2. RPC makes it possible to develop distributed information systems **without having to worry about the details of the underlying network communication.** This makes it easier to develop and maintain distributed information systems.

Here is an example of how RPC is used in distributed systems:

1. Consider a scenario where a client program needs to access a database that is located on a different server computer. The client program can use RPC to make a remote call to the database server to perform the required operation, such as retrieving or updating data.

2. To do this, the client program would first send a message to the RPC server on the database server. The message would contain the name of the database operation to be performed and the parameters of the operation.

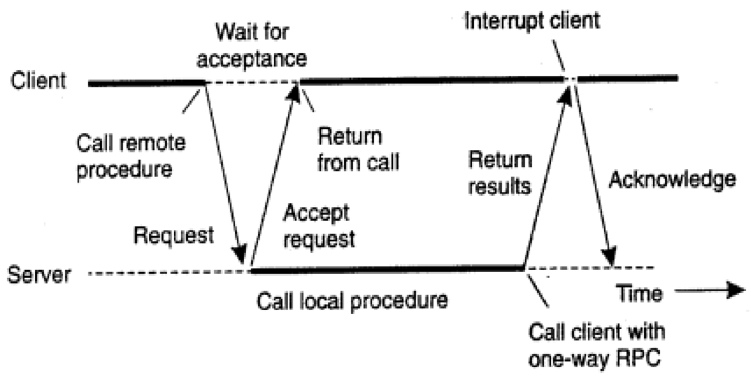
3. The RPC server on the database server would then execute the database operation and send the results back to the client program in a message.

4. The client program would then receive the message and return from the RPC call.

5. Through RPC, the client program is able to access the database on the remote server as if it were a local resource. The client program does not need to know any details about the underlying network communication.

Overall, RPC is a powerful tool that can be **used to implement a wide variety of distributed systems.**

4. EXPLAIN ASYNCHRONOUS RPC. [UNIT 2]



Asynchronous Remote Procedure Call (RPC) is a **communication mechanism that allows a client to issue a remote procedure call to a server without waiting for an immediate reply.** Unlike traditional RPC, where the client blocks until it receives a response from the server,

asynchronous RPC enables the client to continue its execution immediately after sending the request. This is particularly useful in scenarios where the client doesn't need to wait for a result or can perform other tasks during the processing of the remote procedure on the server.

Key points about asynchronous RPC:

1. **No Blocking for Results:**
   In asynchronous RPC, the **client does not wait for the server to process the remote procedure** and send back a response.
2. **Acknowledgment Mechanism:**
   The server immediately acknowledges the receipt of the RPC request by sending a reply to the client. This **acknowledgment signals to the client is that the server has received the request and will begin processing.**
3. **Deferred Synchronous RPC:**
   Asynchronous RPC can be organized as a pair of RPCs where the **client initiates the first RPC,** and upon acknowledgment from the server, continues with its tasks. The **second RPC is then initiated by the server** to provide any necessary results.
4. **Variants:**
   Asynchronous RPCs may have variants where the client does not wait for an acknowledgment. However, **in situations where reliability is essential, these variants may not be suitable.**

In summary, **asynchronous RPC allows for non-blocking communication** between clients and servers, enabling the client to continue its tasks without waiting for a response. This is particularly **beneficial in scenarios where immediate results are not needed,** improving the efficiency of distributed systems.

5. ILLUSTRATE MESSAGE-PASSING INTERFACE IN DISTRIBUTED SYSTEM.[UNIT 2]

In a distributed system, where multiple processes or processors work collaboratively to solve a problem, the Message Passing Interface (MPI) is a **standardized and portable communication protocol** that **facilitates the exchange of messages between these entities.** MPI **provides a set of functions and routines** that enable processes to **send and receive data, synchronize their actions, and coordinate their tasks effectively.**

MPI operates on the **principle of distributed memory,** where each process has its private memory and communicates with others by explicitly sending and receiving messages. This approach allows for efficient **parallel computation,** as processes can work on different parts of the problem independently while sharing data as needed.

Key Components of MPI:

1. **Processes:** Each process represents a **unit of execution** in the distributed system. Processes communicate with each other using MPI functions.
2. **Communicators:** Communicators **define groups of processes that can exchange messages.**

3. **Data Types:** MPI supports a variety of data types, including integers, floating-point numbers, strings, and custom data structures.
4. **Communication Primitives:** MPI provides functions for sending and receiving messages, including **point-to-point communication and collective communication.**
5. **Synchronization Mechanisms:** MPI offers synchronization mechanisms, **such as barriers and collectives, to ensure that processes execute in a coordinated manner.**

Benefits of MPI:

1. **Standardization:** MPI is a widely adopted standard, **ensuring portability across different hardware and software platforms.**
2. **Performance:** MPI are highly optimized for efficient **communication and synchronization.**
3. **Scalability:** MPI can **handle large-scale parallel applications** with thousands or even millions of processes.
4. **Flexibility:** MPI provides a **range of communication patterns and synchronization mechanisms** to suit various application requirements.

Applications of MPI:

1. **Scientific Computing**
2. **High-Performance Computing (HPC)**
3. **Distributed Algorithms**
4. **Real-time Systems**

Overall, the Message Passing Interface plays a crucial role in distributed computing, providing a **structured and efficient mechanism for communication and coordination among processes** in a distributed environment. Its standardization, portability, and performance make it a valuable tool for a wide range of applications, particularly in scientific computing and high-performance computing domains.

6. DESCRIBE SOFTWARE AGENT TECHNOLOGY IN DISTRIBUTED SYSTEMS.

A software agent is defined as an autonomous unit capable of performing a task in collaboration with other.

The field of software agents is considered immature, and there is still much disagreement about how to precisely define them. However, certain types of software agents can be identified, each serving specific purposes.

Types of Software Agents:

1. **Collaborative Agent:**
    - Also known as "multi-agent systems," these agents work together to achieve a common goal. For example, planning a meeting can be a collaborative task among multiple agents.
2. **Mobile Agent:**

- This type of agent is code that can relocate and continue executing on a remote machine. It has the ability to move its execution context from one environment to another.

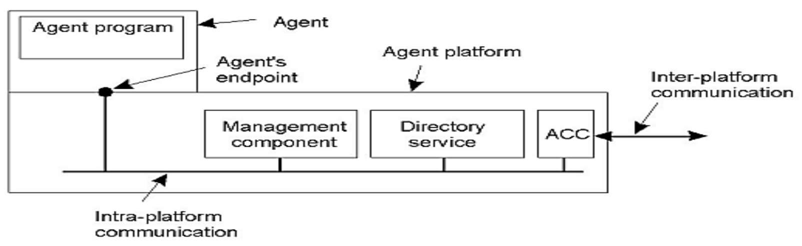
3. **Interface Agent:**
    - These agents possess "learning abilities" and are designed to interact with users, learning from interactions and adapting their behaviour accordingly.
4. **Information Agent:**
    - Agents designed to collect and process geographically dispersed data and information. They are specialized in handling and managing data from diverse sources.

Important Properties:

1. **Autonomous:** Agents can act on their own.
2. **Reactive:** They respond timely to changes in their environment.
3. **Proactive:** Agents can initiate actions that affect their environment.
4. **Communicative:** They can exchange information with users and other agents.
5. **Continuous:** Agents have a relatively long lifespan.
6. **Mobile:** Agents can migrate from one site to another.
7. **Adaptive:** Agents are capable of learning and adapting.

Agent Technology - Standards:
FIPA (Foundation for Intelligent Physical Agents):

- FIPA has standardized the general model of an agent platform.
- **Specifications include:**
    - Agent Management Component.
    - Agent Directory Service.
    - Agent Communication Channel.
    - Agent Communication Language.



The general model of an agent platform
(adapted from [FIPA 1998])

This overview provides a glimpse into the world of software agents, showcasing their diverse types, properties, and the standards that guide their development and interaction in distributed systems.

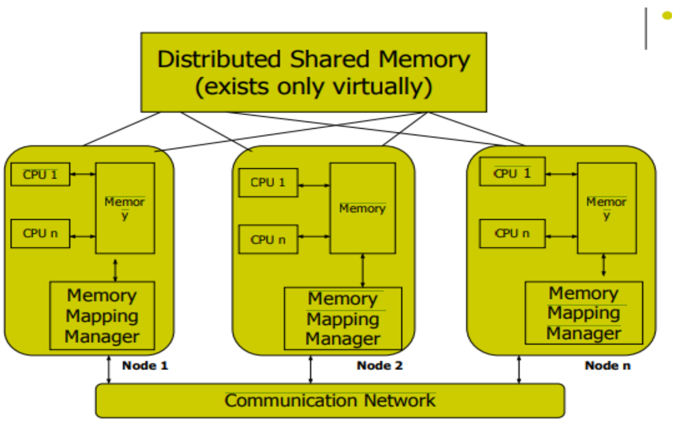
CHAPTER 03

1. EXPLAIN GENERAL ARCHITECTURE OF DISTRIBUTED SHARED MEMORY. [UNIT 3]

Distributed Shared Memory (DSM) is an **architectural model** that provides a **virtual memory address space** shared among nodes in a **distributed system environment.** Here are key points regarding DSM and its general architecture:

DSM Overview:

- DSM **integrates the local memory of different machines** in a network environment into a **single logical entity** shared by cooperating processes.
- The **shared memory exists only virtually,** leading to the term **Distributed Shared Virtual Memory (DSVM).**



General Architecture of DSM System:

- Each node in the system **consists of one or more CPUs and a memory unit,** connected by a **high-speed communication network.**
- The **DSM abstraction** presents a **large shared memory** to the processors of all nodes.
- A software **memory mapping manager** in each node **maps local memory onto the distributed shared memory** to facilitate mapping operations.
- The distributed shared memory is **partitioned into blocks and is shared** by **processes running in a distributed system environment.**
- **Cache memory** of each node is used to **store frequently used pieces of the shared memory** to address **memory latency.**
- The **memory mapping manager views local memory as a big cache** of distributed shared memory.

Design and Implementation Issues of DSM:

1. **Granularity:**
    - Refers to the **block size of the DSM** system.
    - Proper block size selection is crucial for design, as it affects **parallelism and network traffic.**
2. **Structure of Shared Memory:**
    - Depends on the type of application.
    - Defines the **layout of shared memory.** it refers to how the shared memory is **organized or structured.**

3. **Memory Coherence and Access Synchronization:**
    - **"memory coherence"** and **"access synchronization"** are about making sure that all the different copies of shared data in the memories of multiple machines are **consistent and up-to-date.**
4. **Data Location and Access:**
    - "data location and access" refers to the **ability to find and retrieve data** that is needed by a user process. When a user process wants to access a specific piece of data, it needs to know **where that data is located** and **how to retrieve it.**
5. **Replacement Strategy:**
    - Necessary when the local memory is full, and a **cache miss occurs.**
    - Involves **replacing a data block in local memory with a new data block.**
6. **Thrashing:**
    - A situation where **data blocks migrate between nodes too frequently, hindering actual work.**
    - DSM systems must avoid thrashing.
7. **Heterogeneity:**
    - DSM systems for homogeneous systems may not address heterogeneity.
    - In a heterogeneous environment, DSM systems must be designed to handle different architectures.

In summary, DSM provides a **virtual memory address space** shared among processors, and its architecture involves nodes with CPUs and memory, a communication network, and a memory mapping manager. Design and implementation issues include granularity, shared memory space structure, memory coherence, data location, replacement strategy, thrashing avoidance, and handling system heterogeneity.

2. DESCRIBE HETEROGENEOUS DSM. [UNIT 3]

Heterogeneous Distributed Shared Memory (DSM):
Heterogeneous Distributed Shared Memory (DSM) characterizes a distributed shared memory model where **nodes within a system exhibit diverse characteristics in terms of memory,** including **distinct memory organizations, processing speeds, and communication capabilities.** In such a system, **nodes may differ in terms of architectures, operating systems, or hardware configurations.**

Objectives of Heterogeneous DSM:
The primary objective of Heterogeneous DSM is to present a **distributed shared memory abstraction** to applications in a distributed system. This abstraction allows **transparent data access and sharing across** the system, irrespective of the inherent heterogeneity.

Key Techniques:
Heterogeneous DSM systems commonly **employ techniques like data replication and coherence protocols.** Data replication involves creating multiple copies of shared data across the distributed system, ensuring each node possesses a local copy. Coherence protocols maintain consistency among these replicated copies, **addressing potential inconsistencies due to concurrent accesses and updates.**

Challenges and Solutions: Managing heterogeneity poses challenges in **memory access latencies, communication protocols, and data representation formats. Load balancing and data placement strategies** are essential for efficient resource utilization. Despite challenges, heterogeneous DSM systems offer advantages such as **improved scalability, fault tolerance.**

In summary, Heterogeneous DSM systems enable the utilization of diverse resources in a distributed environment while **abstracting complexities related to heterogeneity.** This **results in efficient and transparent memory access** across the entire system.

3. EXPLAIN THE FOLLOWING W.R.T. DISTRIBUTED SHARED MEMORY. I) STRICT CONSISTENCY MODEL II) CASUAL CONSISTENCY MODEL III) WEAK CONSISTENCY MODEL IV) RELEASE CONSISTENCY MODEL

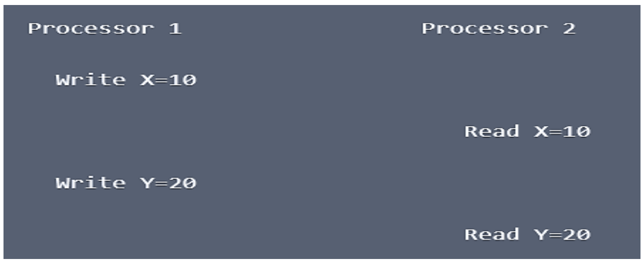
Distributed Shared Memory (DSM): In a DSM system, multiple nodes in a distributed system share a **global virtual memory address space,** allowing processes on different nodes to read and write shared memory. Consistency models **define the rules** governing **how the shared memory is updated and observed by different nodes.**

Here's a brief explanation of each consistency model with respect to DSM:

1. **Strict Consistency Model:**
    - The **"strict consistency model"** refers to a way of organizing a system where all the nodes in the system **see the same order of operations.** This means that **each operation becomes immediately visible to all the nodes.**

    - When using strict consistency in a DSM system, **if one process writes data to the shared memory, all other processes can immediately see and access that updated data.** Similarly, if a process reads data from the shared memory, it will always see **the most up-to-date version** of that data.

Here is an explanation of the strict consistency model with an example and a diagram:



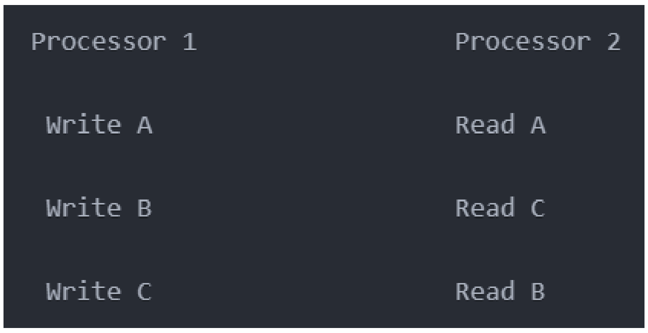
As an example:

- Processor 1 writes X=10 followed by Y=20 to shared memory
- Processor 2 reads the value of X and then Y from the same location
- With strict consistency, Processor 2 is guaranteed to see the writes by P1 in X=10 and then Y=20, exactly in the order they were performed

2. **Causal Consistency Model:**
    - Causal consistency allows for a more **relaxed order of operations but maintains the causal relationship between events.** If event A causes event B,

then all processes should agree on this order. **Concurrently occurring events** can be **observed in different orders** by different processes.

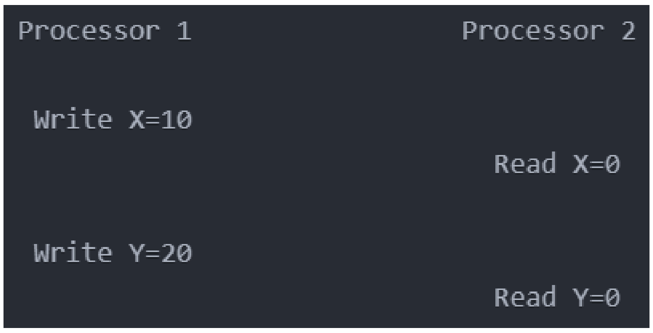
Here is a neat diagram for the causal consistency model in distributed shared memory systems:



```
Processor 1                 Processor 2

 Write A                    Read A

 Write B                    Read C

 Write C                    Read B
```

3. **Weak Consistency Model:**
    - In a weak consistency model, **order of operations are not maintained strictly.** Different nodes may have varying views of the shared data, so **weak consistency model does not ensure that all nodes will immediately see the latest value written to memory.** Writes will propagate at some point but the model makes no guarantees about when.

Here is an explanation of the weak consistency model with an example and a diagram:



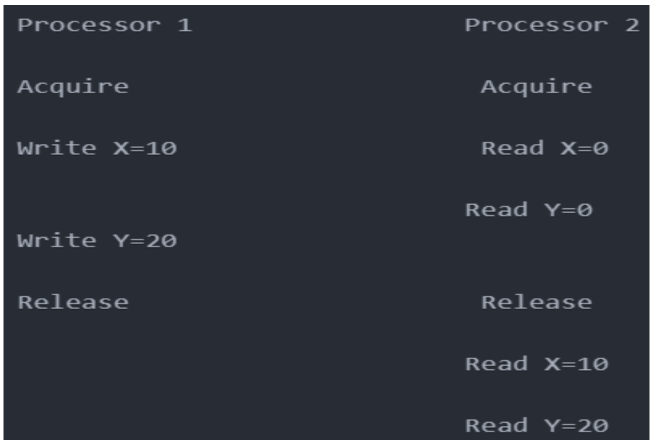
```
Processor 1                  Processor 2

 Write X=10
                               Read X=0

 Write Y=20
                               Read Y=0
```

As an example:

- Processor 1 writes X=10 and then writes Y=20 to shared memory
- Processor 2 reads X and Y from the same memory location
- With weak consistency, Processor 2 can read the initial value of X and Y (0) even after Processor 1 has written new values
- The new values 10 and 20 may be read **after an arbitrary amount of time**

4. **Release Consistency Model:**
    - In this model, **memory access is divided into phases** marked by **"acquire"** and **"release"** points. Within a phase, nodes can perform operations and writes. The key idea is that **writes done in a phase by one processor** will be visible on other processors once a release completes. However, there are **no ordering of operations guarantees across phases.**

Here is an explanation of the release consistency model with an example and diagram:



```
Processor 1                      Processor 2

Acquire                            Acquire

Write X=10                         Read X=0

                                 Read Y=0
Write Y=20

Release                            Release

                                 Read X=10

                                 Read Y=20
```

In the example:

- Processor 1 does writes in an acquire-release phase
- The writes X=10 and Y=20 are not visible to P2 when reading the variables before Processor 1's release completes
- After Processor 1's release and P2's acquire, it reads the updated values of X and Y

4. ILLUSTRATE SEQUENTIAL CONSISTENCIES USING NON-REPLICATED AND MIGRATING BLOCKS. [UNIT 3]

Sequential consistency is a DSM consistency model that **defines the order in which operations on shared memory must be observed by different nodes in a distributed system.**

Non-Replicated Migrating Blocks refer to a specific approach in Distributed Shared Memory (DSM) systems.

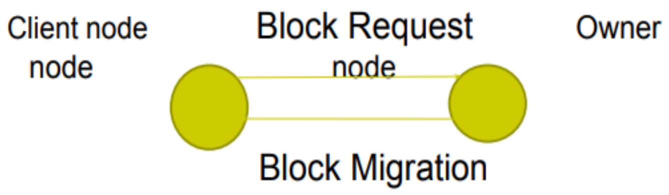
1. **Single Copy:** In the case of Non-Replicating Migrating Blocks, **each block of shared memory exists as a single copy** throughout the entire system. Unlike replicated blocks, **there are no duplicate copies of the block on different nodes.**

2. **Block Migration:** When a read or write operation is required on a specific block, the block is migrated or moved from its current node to the node where the access is needed. This migration process involves **transferring the ownership of the block from the current node to the destination node.**

3. **Fixed Owner Node:** **Each block has a designated owner node** responsible for **maintaining and managing that block.** The owner node is initially assigned when the block is created and remains fixed until the block is migrated.

4. **Owner Node Change:** When a block is migrated to a new node, the ownership of the block is transferred to the destination node.

Nonreplicated, migrating blocks (NRMBs)



Non-replicated Migrating Blocks:

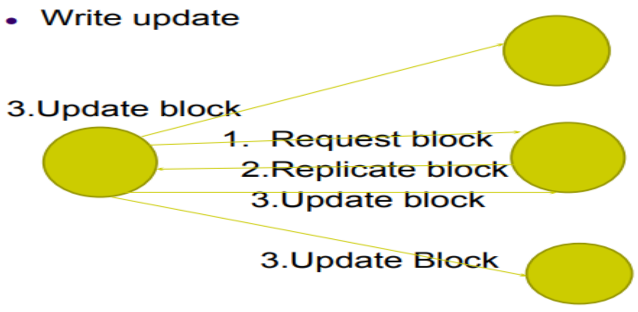
- In non-replicated migrating blocks, a **sequential consistency violation can occur when multiple nodes access and modify shared memory concurrently without synchronization.**
- Let's consider a simple example with **two processors, P1 and P2,** accessing a shared variable x:
    - Initially, x = 0.
    - P1 executes x = x + 1, incrementing the value of x to 1.
    - P2 executes x = x + 1, incrementing the value of x to 2.
- In a **sequentially consistent system,** the **result should reflect the order of the operations,** ensuring that **P1's update is observed before P2's update.** However, **without synchronization, the order of execution may vary,** leading to different results. For example, P2's update may be observed before P1's update, violating sequential consistency.

In **non-replicated migrating blocks** violation in sequential consistency can occurs **if proper synchronization of the order of operations is not maintained.** To ensure sequential consistency synchronization mechanisms such as **Mutex, Switches, locks, and barriers** are used.

5. EXPLAIN SEQUENTIAL CONSISTENCIES USING REPLICATED, NON-MIGRATING BLOCKS. [UNIT 3]

Replicated, non-migrating blocks refer to a specific approach where **multiple copies of data are stored on different nodes,** and **these copies do not move or migrate to different locations** during execution.

Replicated, nonmigrating blocks (RNMBs)



AFTAB-1405

Replicated and non-migrating blocks represent a different approach in distributed shared memory (DSM) systems compared to non-replicated and migrating blocks.

Sequential Consistency with Replicated and Non-Migrating Blocks:

1. **Multiple Copies with Consistency:**
    - In the case of replicated and non-migrating blocks, **each shared memory block is replicated across different nodes** in the system. This means every node maintains a copy of the data, and **updates to any copy need to be propagated to maintain consistency across all replicas.**

2. **No Block Migration:**
    - Unlike migrating blocks where the data physically moves to the node that requires it, **non-migrating blocks stay in place. Instead of moving, updates are transmitted across the system to update replicas on different nodes.**

3. **Synchronized Updates:**
    - In a sequentially consistent system, **updates to any of the replicated blocks are synchronized such that all nodes observe these updates in the same order.**

4. **Static Ownership with Replication:**
    - **Ownership of blocks does not change because blocks do not migrate.**

Example Demonstrating Sequential Consistency:
Imagine we have two processors, P1 and P2, but this time in a system using replicated and non-migrating blocks. Both processors access a shared variable x, which is initially 0 and is replicated across both their local caches.

- P1 executes x = x + 1, incrementing the value of x to 1.
- Almost simultaneously, P2 also executes x = x + 1, aiming to increment the value of x.

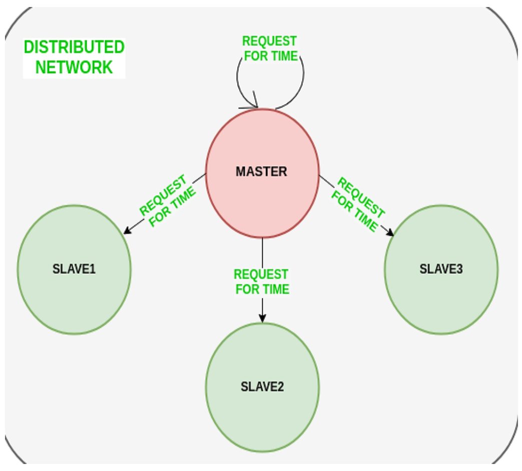
In the replicated and non-migrating block system:
- The updates by both P1 and P2 are sent to a **coordination mechanism that ensures updates are ordered consistently across all replicas of x.**
- This **system guarantees that all nodes will see the operations in the same sequence;** for example, P1's increment might be processed first, followed by P2's increment, resulting in a final, consistent value of x = 2.
- Even though both operations took place at nearly the same time, both nodes will observe the changes in the same order, hence maintaining sequential consistency.

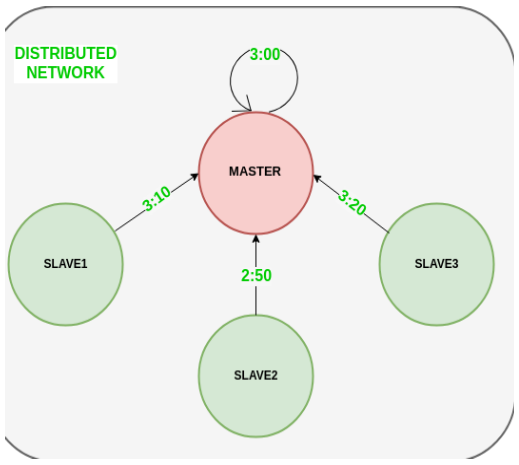
6. DEMONSTRATE BERKELEY ALGORITHM FOR CLOCK SYNCHRONIZATION. [UNIT 3]

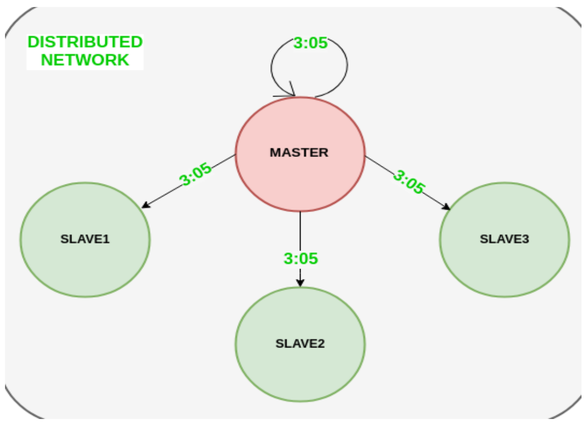
Clock synchronization in distributed computing systems refers to the **process of ensuring that the clocks across multiple devices or nodes in the system are set to the same time or are closely aligned.** This synchronization is crucial for **coordinating activities and events** in a distributed system.

The Berkeley Algorithm is a **clock synchronization protocol** used in distributed computing systems to **sync all machine clocks to the time of a "master" machine.**

Here's an explanation of how it works:







1. **Election of the Master Node:** To start with, a master node is elected out of all the nodes in the distributed system. This **master node is the one that performs the core task of performing clock synchronization.**

2. **Polling and Collection of Times:** Once elected, the **master node polls each of the slave nodes** in the distributed system to **report their current time.**

3. **Calculation of Average Time:** After the master node receives all the times, it **calculates the average time.** The master node also takes its **own local time** into consideration when calculating this average.

4. **Adjust Slave Clocks:** The master node proceeds to send out the **time difference (offset)** to each of the slave nodes. This offset indicates how much each slave node should adjust its clock. A **positive offset means** the slave node should **move its clock forward,** and a **negative offset means** it should **move its clock backward.**

5. **Adjust Master Clock:** If necessary, the master node will also adjust its own clock based on the calculated average time.

AFTAB-1405

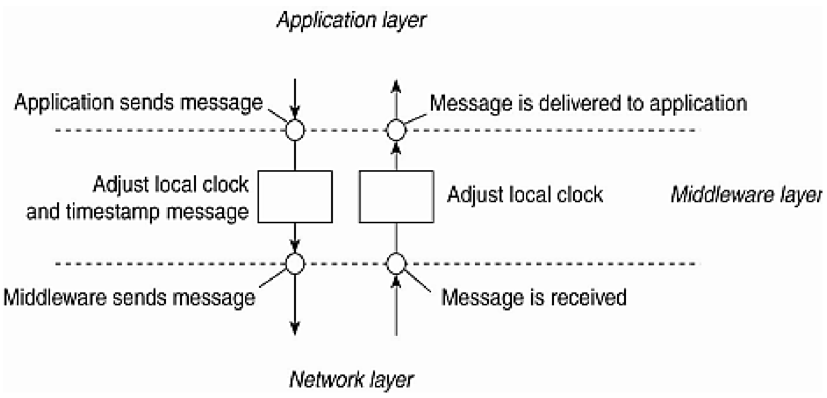
This process of **steps 2-5 keeps repeating periodically to maintain synchronization** as **system clocks inevitably drift over time.** It's important to note that the algorithm mitigates the impact of **delay in message transmission** by considering the **round-trip time of messages.**

The key to the effectiveness of the Berkeley Algorithm is the use of the average time and the updating of all nodes based on this time, but, it is based on the assumption that **skew (rate of drift)** in the clocks is relatively **small** and **stable** over short periods.

7. EXPLAIN THE IMPLEMENTATION OF LOGICAL CLOCKS BY USING COUNTERS/LAMPORT'S ALGORITHM.

Lamport's Algorithm, proposed by **Leslie Lamport,** is a distributed algorithm **used to establish a partial ordering of events** in a distributed system. It is based on the concept of **logical clocks,** which are used to **order events** without requiring access to **physical clocks or a global time**.

In Lamport's Algorithm, each **event** in the distributed system **maintains a logical clock,** represented by a **local counter.** The **logical clock is used to assign a timestamp to each event,** allowing for the **establishment of a partial ordering of events.**



The positioning of Lamport's logical clocks in distributed systems

The implementation process involves the following steps:

1. **Before a process executes an event** (such as sending a message or executing an internal event), the **process increments its local counter by 1.** This incrementation serves as the **'timestamp'** for the event.

2. If the process sends a message to another process, it sets the **message's timestamp equal to its current local counter value** (after incrementing the counter in the previous step). This **stamped value indicates when the message was sent,** in logical clock terms.

3. When a process receives a message, **it adjusts its local counter.** The local counter value becomes the **maximum of its current value** and the **received message's**

timestamp value. Then, the **process increments this new counter value by 1** and delivers the message to the application.

This mechanism ensures that all **events occurring in the distributed system have unique timestamps and can be ordered logically**. However, it's important to note that these time values **do not reflect the actual time** of the events.

Using this implementation of **logical clocks with counters,** distributed systems can **achieve a partial ordering of events,** even **without relying on physical clocks or a global time reference.** This allows for the **coordination and synchronization of activities** within the distributed system.

8. EXPLAIN CENTRALIZED APPROACH FOR MUTUAL EXCLUSION.

Definition: Mutual Exclusion is a fundamental concept in distributed systems, referring to the need for ensuring that **multiple processes or nodes do not access a shared resource simultaneously.** Mutual exclusion is essential to **maintain data integrity.** This is crucial to **prevent corruption or inconsistency** of the shared resource due to **concurrent access.**

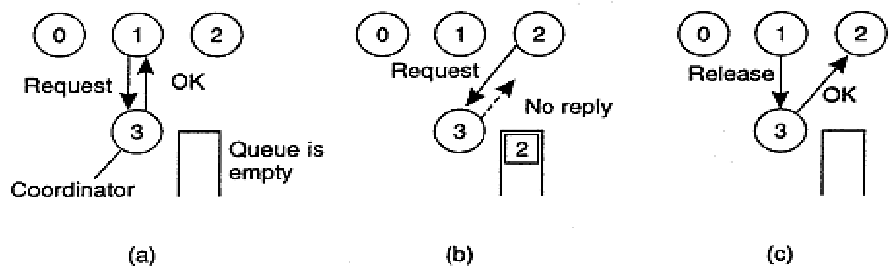


Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

Centralized Approach to Mutual Exclusion:

In a centralized approach, there is a **single central authority,** often referred to as the **"coordinator".** In a centralized approach to mutual exclusion, a **coordinator is designated to manage access of shared resources.** The **coordinator** maintains a data structure to track which process is currently holding resource.

The process involves the following steps:

1. **Requesting Access:**
   When a process or node wants to access the shared resource, it **sends a request message to the coordinator**, specifying the **resource** and seeking permission.
2. **Permission Grant:**
   If no other process is currently accessing the resource, the **coordinator grants permission by sending a reply to the requesting process.**

3. **Denying Permission:**
   If another process or node is already accessing the resource, the coordinator denies permission. The exact method of denial can vary, such as sending an explicit **"permission denied"** message.

4. **Release and Grant:**
   When a process or node finishes work with the resource, it notifies the coordinator. The coordinator then **grants permission to the next process in the queue** of deferred requests.

Advantages:

- **Guarantees mutual exclusion.**
- Requests are granted in the order that they received.
- No process waits indefinitely **(no starvation).**

Disadvantages:

- **Single Point of Failure:** The coordinator poses a risk, as its failure can bring down the entire system.
- **Performance Bottleneck:** In large systems, a single coordinator may become a **performance bottleneck.**

9. DISTRIBUTED ALGORITHM FOR MUTUAL EXCLUSION.

Distributed Algorithm for Mutual Exclusion

In distributed systems, a **mutual exclusion algorithm ensures that only one process can access a shared resource at a time,** even when multiple processes are running on different computers. This **prevents conflicts and maintains data integrity.** One such algorithm is the **Ricart and Agrawala algorithm,** which combines **Lamport's clock synchronization** and a **total ordering of events** in the system.

In this explanation, we will explore the underlying principles, key steps, and relevant concepts of this distributed algorithm for mutual exclusion.

Underlying Principles:

The Ricart and Agrawala algorithm relies on the following principles:

1. **Logical Clocks:** Processes in a distributed system maintain logical clocks that provide a partial ordering of events. Logical clocks **capture the ordering of events,** even if they occur concurrently on different processes.

2. **Message Passing:** Communication between processes is achieved through message passing. Processes exchange request and reply messages to coordinate access to the shared resource.

The Ricart and Agrawala algorithm involves the following steps:

1. **Request Message:**
    - When a process wants to access a shared resource, it **constructs a request message** containing the **resource's name, its process number,** and the **current logical time.**
    - This request message is broadcasted to all other processes, including the sender.

2. **Handling Request:**
    - Upon receiving a request message, the recipient process takes action based on its state regarding the named resource:
        - If the recipient is **not accessing or not interested in the resource**, it replies with an **OK message** to the sender, granting permission to access the resource.
        - If the recipient already has access to the resource, it **queues the request without replying.**
        - If the recipient wants to access the resource but hasn't yet, it **compares the timestamps in the received request and its own request.**
            - The request with the lowest timestamp wins. If the incoming request has a lower timestamp, the recipient replies with an OK message, granting permission; otherwise, it queues the incoming request.

3. **Permission Collection:**
    - After sending requests, a process **waits until it receives an OK message** from all other processes.
    - Once all permissions are granted, the process proceeds with accessing the shared resource.

4. **Release and Communication:**
    - After completing its use of the resource, the process sends OK messages to all processes in its queue, indicating that they can now proceed.
    - The process then clears its queue, allowing other processes to access the resource.

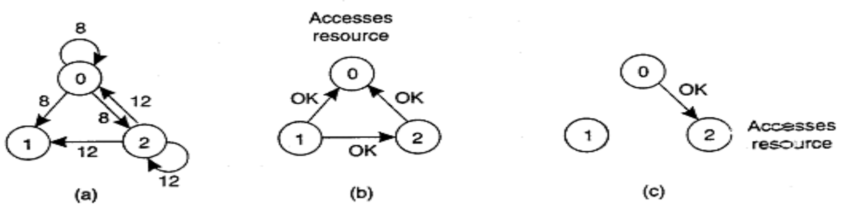


Figure 6-15. (a) Two processes want to access a shared resource at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now go ahead.

Examples and Use Cases:

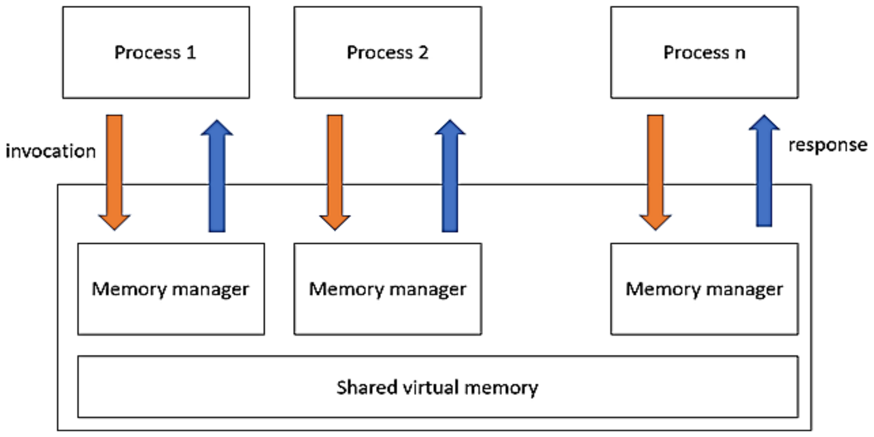
Let's consider an example where multiple processes need to access a printer in a distributed system.

1. Process A sends a request message to access the printer, with a timestamp of 5.
2. Process B, which already has access to the printer, queues the request from Process A.
3. Process C, which also wants to access the printer but hasn't yet, compares the timestamps. If its timestamp is lower than Process A's timestamp, it replies with an OK message. Otherwise, it queues Process A's request.
4. Once Process A receives OK messages from all other processes, it can proceed to access the printer exclusively.
5. After completing its print job, Process A sends OK messages to clear the queues of other processes, allowing them to access the printer.

The Ricart and Agrawala algorithm ensures that only one process can access the printer at a time, preventing conflicts and maintaining data integrity.

10. EXPLAIN THRASHING IN DISTRIBUTED SHARED MEMORY.

Thrashing in distributed shared memory refers to a situation where the system's **performance degrades significantly** due to **excessive communication and synchronization overhead** among distributed processes or nodes. It occurs **when the system spends a significant amount of time and resources on managing distributed shared memory operations rather than executing useful computations.**



To understand thrashing in distributed shared memory, let's consider a scenario where multiple processes or nodes in a distributed system are trying to access shared memory simultaneously. **When the number of requests for accessing the shared memory surpasses, thrashing can occur.** This **excessive demand** for **shared memory access** leads to increased communication and synchronization overhead, resulting in poor performance and limited progress in actual computations.

Thrashing in distributed shared memory typically arises due to the following reasons:

1. **Increased Inter-Process Communication**: As more processes contend for access to shared memory, the frequency of communication between processes increases. This communication overhead can **saturate the network and degrade overall system performance.**

2. **Excessive Synchronization: To maintain consistency and order in accessing shared memory, synchronization mechanisms such as locks or barriers are employed.** However, when numerous processes contend for access, the system spends a significant amount of time on synchronization, leading to thrashing.
3. **Insufficient Memory Bandwidth:** If the available memory bandwidth cannot handle the high demand for shared memory access, the system becomes overwhelmed, resulting in thrashing.

4. **Inefficient Data Placement: Improper assignment of data** to distributed memory nodes can lead to increased remote memory access and contention, exacerbating thrashing.

The impact of thrashing includes a significant **decrease in system throughput, increased response times,** and a higher likelihood of **deadlock or live lock situations.**
To mitigate thrashing in distributed shared memory systems, various techniques can be employed, such as:

1. **Load Balancing:** Distributing the workload evenly across nodes to **avoid overloading specific nodes** and reducing contention for shared memory.

2. **Caching and Prefetching:** Utilizing local caches and prefetching mechanisms to reduce remote memory access and contention.

3. **Optimized Synchronization:** Employing efficient synchronization mechanisms, such as **fine-grained locking or lock-free algorithms,** to minimize the impact of synchronization overhead.

4. **Data Placement Strategies:** Utilizing intelligent data placement strategies to **reduce remote memory access and improve locality.**

11. EXPLAIN BULLY ALGORITHM.

The Bully Algorithm, **devised by Garcia-Molina in 1982,** is an **election algorithm** used in distributed systems. Its **primary purpose is to elect a coordinator** among a group of processes when the **current coordinator fails to respond.**

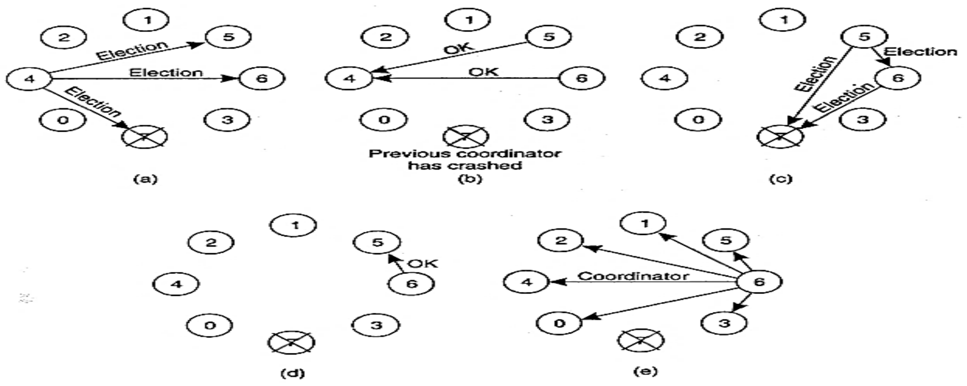


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

How It Works:

1. **Initiation of Election:**
   - When a process (let's call it P) notices that the coordinator is not responding, it initiates an election.

2. **Election Message:**
   - P sends an **ELECTION** message to all processes with higher numbers than itself.

3. **No Response:**
   - If no higher-numbered process responds, P wins the election and becomes the new coordinator.

4. **Higher-Numbered Response:**
   - If one of the higher-numbered processes responds, that process takes over as the new coordinator, and P's election process is terminated.
5. **OK Messages:**
   - Any process receiving an ELECTION message responds with an OK message, indicating that it is alive and ready to take over if needed.
6. **Cascade Elections:**
   - If a process receives an OK message, indicating the presence of a higher-numbered process, it starts its own election process to compete for the coordinator role. This cascade effect continues until the process with the highest number wins and becomes the coordinator.
7. **Winner Announcement:**
   - Eventually, all processes, except the winner, give up on the election process. The winning process announces its victory by sending a **COORDINATOR** message to all processes.

Example Scenario:

- Suppose there are eight processes (0 to 7) in a distributed system.
- Process 7 was the coordinator but has crashed.
- Process 4 initiates an election by sending ELECTION messages to processes 5, 6, and 7.
- Processes 5 and 6 respond with OK, indicating they are alive.
- Process 4 stops its election process as it knows one of the higher-numbered processes will take over.
- Now, processes 5 and 6 each start their own elections, leading to a cascade of elections.
- Process 6 wins and informs everyone by sending a COORDINATOR message.
- The system can now resume normal operations with process 6 as the new coordinator.

The Bully Algorithm **ensures that the process with the highest number becomes the coordinator,** hence the name "bully." It handles coordinator failures and enables the system to seamlessly transition to a new coordinator.

12. EXPLAIN RING ALGORITHM.

The Ring Algorithm is an **election algorithm** used in distributed systems to elect a new coordinator when the existing coordinator is no longer functional. In this algorithm, processes are organized in a ring, and **each process knows its successor in the ring.**

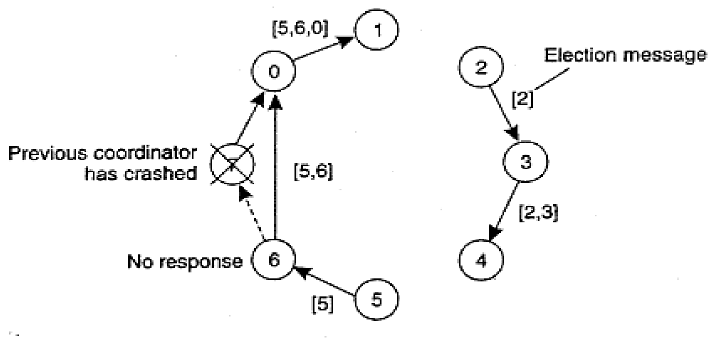


Figure 6-21. Election algorithm using a ring.

How It Works:

1. **Initiation of Election:**
   - When a process detects that the coordinator is not functioning, it initiates an election by building an **ELECTION message** containing its own **process number.**

2. **Message Circulation:**
   - The process **sends the ELECTION message to its successor** in the ring. **If the successor is down, the sender skips over it** and continues to the next member along the ring.

3. **Candidate Addition:**
   - At each step, the **sender adds its own process number to the list** in the message.

4. **Message Completion:**
   - Eventually, the **ELECTION message circulates back to the process that initiated it.** The process recognizes this when it receives the **incoming message containing its own process number.**

5. **Coordinator Announcement:**
   - The **message type is then changed to COORDINATOR,** and it circulates once again to inform everyone else who the new coordinator is and who the members of the new ring are.

6. **Message Removal:**
   - **Once the COORDINATOR message has circulated once, it is removed**, and normal operations resume.

Key Points:

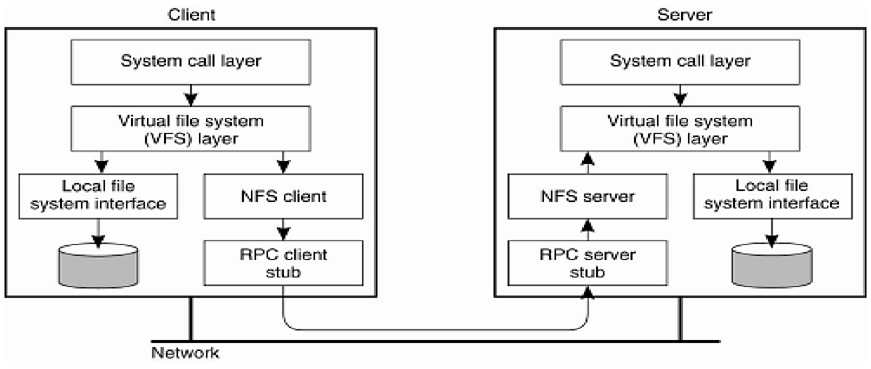
- The Ring Algorithm is based on the **physical or logical ordering of processes** in a ring.
- Each process initiates an independent election, and **multiple elections can occur simultaneously without interference.**
- The **process with the highest number becomes the new coordinator.**

This algorithm provides a **decentralized approach to coordinator election,** leveraging the ring structure to determine the new coordinator in the event of a coordinator failure.

CHAPTER 04:

1. EXPLAIN THE BASIC NFS ARCHITECTURE FOR UNIX SYSTEM.

The basic NFS (Network File System) architecture for UNIX systems follows a **client-server model.** Here's a simplified explanation:


Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

The basic NFS architecture for UNIX systems

Client-Server Model:

- **Client Side:** The **client accesses the file system using its local operating system's system calls.** However, instead of interacting directly with the local file system, **it communicates with a Virtual File System (VFS).**
- **VFS Interface:** The VFS **acts as an interface to different file systems, including NFS.** It **abstracts the differences between various file systems,** providing a consistent interface.
- **NFS Client:** The NFS client is a component that **handles access to files stored on a remote server.** It **converts operations on the VFS interface into Remote Procedure Calls (RPCs)** to the server.
- **Server Side:** The **NFS server is responsible for managing incoming client requests.** It **converts RPCs from clients into local file operations** that are then executed on the server's file system.

Communication:

- All **communication between the NFS client and server is done through Remote Procedure Calls (RPCs).** These RPCs represent **file system operations.**

Independence from Local File Systems:

- NFS is designed to be independent of the specifics of local file systems. It doesn't matter if the client and server are running different operating systems or file systems, as long as they **comply with the NFS file system model.**

File System Model:

- **File Representation:** Files are treated as **sequences of bytes, organized hierarchically into directories and files.**
- **Naming:** Files are **named and accessed through a UNIX-like file handle.** Clients look up a file's name in a naming service to obtain the associated file handle.

- **Attributes:** Files have attributes (e.g., type, length, modification time) that can be read and modified using specific operations.

File Operations:

- NFS supports various file operations, including create, link, symlink, mkdir, mknod, rename, remove, open, close, lookup, readdir, readlink, getattr, setattr, read, and write.

NFS Versions:

- Versions discussed include **NFSv3 and NFSv4,** each with its own characteristics and improvements.

In summary, **NFS provides a standardized way for clients to access files from a remote server, abstracting the underlying file system details** and enabling **transparent file sharing** across different machines in a client-server architecture.

2. EXPLAIN RPCS IN NFS.

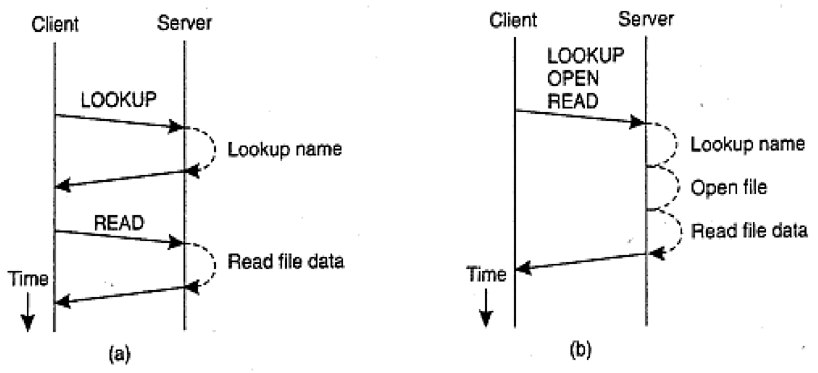


Figure 11-7. (a) Reading data from a file in NFS version 3. (b) Reading data using a compound procedure in version 4.

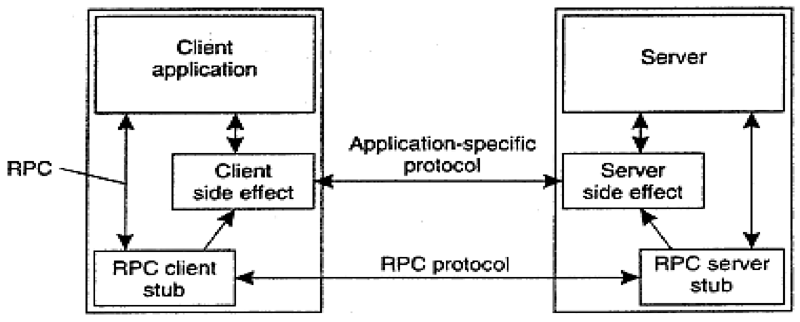


Figure 11-8. Side effects in Coda's RPC2 system.

Remote Procedure Calls (RPCs) in NFS:

- **Protocol Used:** NFS uses the **Open Network Computing RPC (ONC RPC) protocol** for **communication.** This protocol is formally defined, ensuring a standardized way for the client and server to communicate.

- **Single RPCs in NFSv3:** In earlier versions (up to NFSv3), **each NFS operation, such as reading data from a file, was typically implemented as a separate RPC.** For example, to read data, a **client first need to perform a lookup operation** to find the **file handle** and then issue a **separate read request.**

- **Compound Procedures in NFSv4:** In NFSv4, there's a more efficient approach called **compound procedures.** Instead of making separate RPCs for each operation, **multiple operations can be grouped into a single RPC.** This **reduces the number of RPCs between the client and server.**

- **Transactional Semantics:** Compound procedures in NFS do not have transactional semantics. **Operations are executed in the order they are requested,** and **if one operation fails, subsequent operations in the compound procedure are not executed.**

In summary, RPCs in NFS facilitate communication between the client and server. While **earlier versions relied on separate RPCs for each operation, NFSv4 introduced compound procedures** to optimize performance by grouping multiple operations into a single RPC.

3. EXPLAIN NAMING IN NFS.

Naming in NFS (Network File System) used to **uniquely identify file systems** over the network.
Two main components of NFS:
1. NFS Client – Trying to access file system.
2. NFS Server – Server stores shared files.

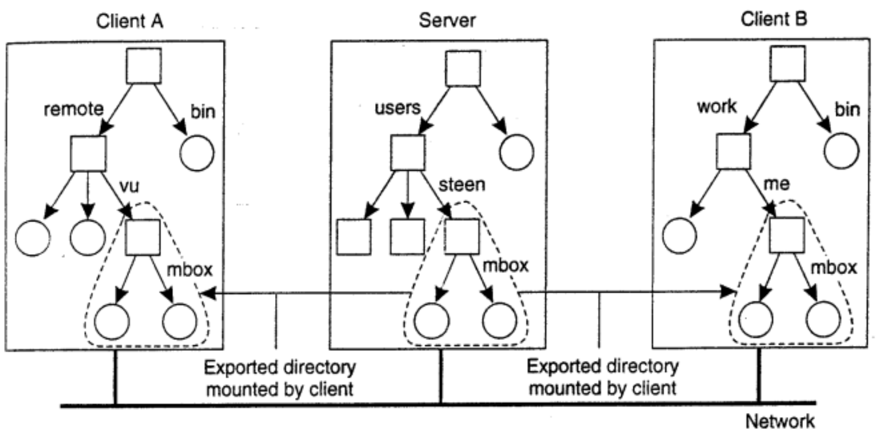

Client A
Server
Client B
remote
bin
vu
mbox
users
steen
mbox
work
bin
me
mbox
Exported directory mounted by client
Exported directory mounted by client
Network

Figure 11-11. Mounting (part of) a remote file system in NFS.

Here are some key points related to naming in NFS:

Hierarchical Structure
- NFS utilizes directories and sub-directories, similar to folders on your local computer, to organize files in a tree structure. This allows **logical grouping and easier navigation when accessing files.**

Mounting Remote Directories

- Clients can mount remote directories shared by the NFS server onto their own local directory hierarchy. This **"mounting" process integrates the remote directory into the client's file system,** making it transparently accessible like any local resource.

Shared Namespace Challenges

- When mounted by different clients, the **same remote directory may be integrated at different locations in each client's namespace.** This can make **shared naming conventions** difficult.

File Handles

- NFS uses file handles, which **uniquely identify files and directories on the NFS server.** Clients **cache these handles,** avoiding repetitive lookups and enhancing performance when accessing files repeatedly.

NFSv4 Enhancements

- NFSv4 brought **recursiveness and junction traversal capabilities.** This **allows clients to directly look up full pathnames** on the server rather than iterative lookups. **Junctions are traversed automatically.**

In essence, NFS naming aims to provide **location transparency** and performance while also accommodating shared namespace challenges in distributed environments.

4. EXPLAIN AUTOMOUNTING IN NFS.

The concept of auto mounting in NFS is used to **automatically mount files and directories when they are needed or accessed, without requiring the user to manually run mount commands.**

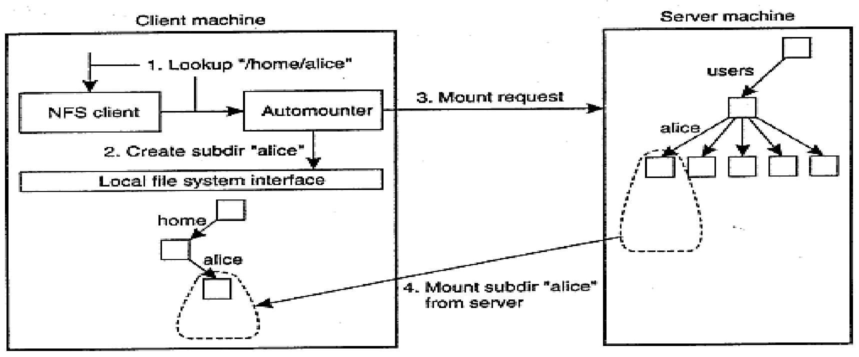


Figure 11-13. A simple automounter for NFS.

The key benefits of auto mounting are:

1. **Convenience for the User:** The **user does not have to remember and explicitly run any mount commands to access remote filesystems.** It happens automatically behind the scenes.
2. **On-Demand Mounting:** The **remote directories get mounted only when the user accesses or require them.**

3. **Automatic Unmounting:** Once the user moves out of the mounted directory, it gets automatically unmounted in the background.

Here is how auto mounting works:

1. The **administrator pre-configures mappings of remote directories to local mount point directories.** For example, remote server1***:/export/users*** should map to local ***/home/remote*** mount point.
2. When the user tries to access the local mount point directory, the **NFS client automatically and transparently mounts the mapped remote filesystem there** to fulfil the user's request.
3. The user access continues seamlessly seeing just the local /home/remote without realizing the behind-the-scenes mount.
4. Finally, when the user exits the **local mount point directory,** automatic unmount happens.

In this manner, auto mounting in NFS provides **on-demand, transparent mounting and unmounting** of remote file systems for ease of access by users.

5. FILE SHARING IN CODA.

Coda is a distributed file system developed at **Carnegie Mellon University** to provide **highly available and fault-tolerant file access** for mobile or disconnected users. Coda **involves a special allocation scheme to facilitate concurrent access of files by multiple clients.** Let's break down how file sharing works in Coda.

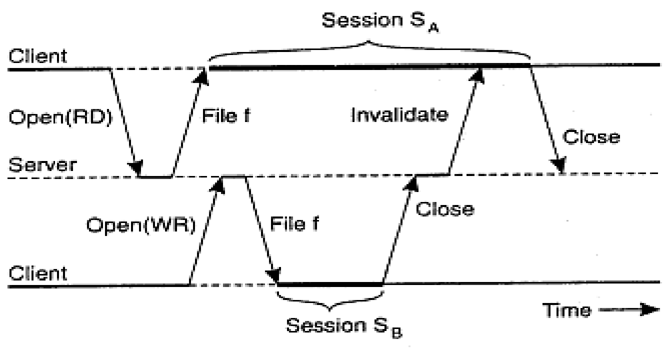


Figure 11-20. The transactional behavior in sharing files in Coda.

1. **Initial File Open:**
   - When a client successfully opens a file 'f' in Coda, a complete copy of the file is transferred to the client's machine.
   - The server notes that the client now has a copy of 'f,' .
2. **Read and Write Access:**
   - If a client, let's say **client A, opens 'f' for writing, any attempt by another client (e.g., client B) to open 'f' will fail.** This is because the server knows that client A might be modifying 'f.'
   - However, **if client A had opened 'f' for reading, client B could successfully obtain a copy of 'f' for reading.** Additionally, **if client B also wanted to open 'f' for writing, it would succeed.**

3. **Transaction-Like Behaviour:**
    - **Coda treats interaction with a file as a transaction.**
    - Imagine two processes, A and B. A has a read session (SA) for 'f,' while B has a write session (SB) for 'f'.
    - When B completes its write session (SB) and closes the file, the **updated version of 'f' is sent back to the server.** The server then notifies A (**through an invalidation message**) that its local copy is outdated.
    - Despite having outdated local copies, other **clients can continue to read from them until they attempt to modify the file.**

In essence, Coda's file-sharing mechanism ensures that **multiple clients can read from a file concurrently, but modifications are serialized to maintain consistency.** The **transactional approach** allows to **control concurrent access of files/directories.**

6. EXPLAIN CLIENT-SIDE CACHING IN CODA.

Caching is a technique used in **network file systems** to improve performance by **storing frequently accessed data closer to the user.** Instead of fetching data from the server every time it's needed, a **local cache retains a copy of recently accessed files.** This helps **reduce latency and enhances overall system efficiency.**

Coda is a distributed file system that relies on **client-side caching.** It's designed to **enhance scalability and fault tolerance** of distributed file system. If client open a file then copy of that file transferred to the client's local machine. **Coda maintain cache coherence by using call-backs.** When a client updates a locally cached file, it notifies the server, which, in turn, sends invalidation messages to other clients. This ensures that all clients have consistent data.

Client-side caching in Coda is a crucial aspect of the system's operation for two main reasons: **scalability and fault tolerance.**

Here's a simplified explanation:

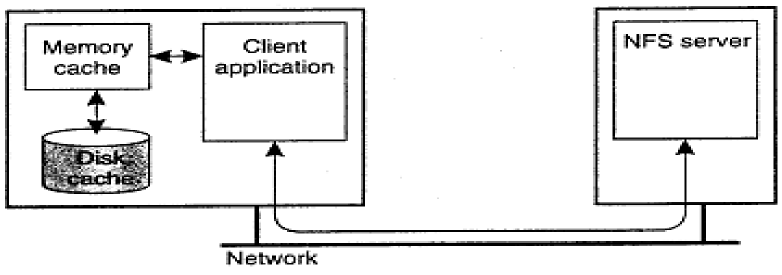


Figure 11-21. Client-side caching in NFS.

1. **Scalability:**
    - Caching is employed to achieve scalability in Coda, **allowing the system to handle a larger number of users and requests efficiently.**
    - When a file is opened for reading or writing, the entire file is transferred to the client and cached locally. This means that the client has a full copy of the file stored on its machine.

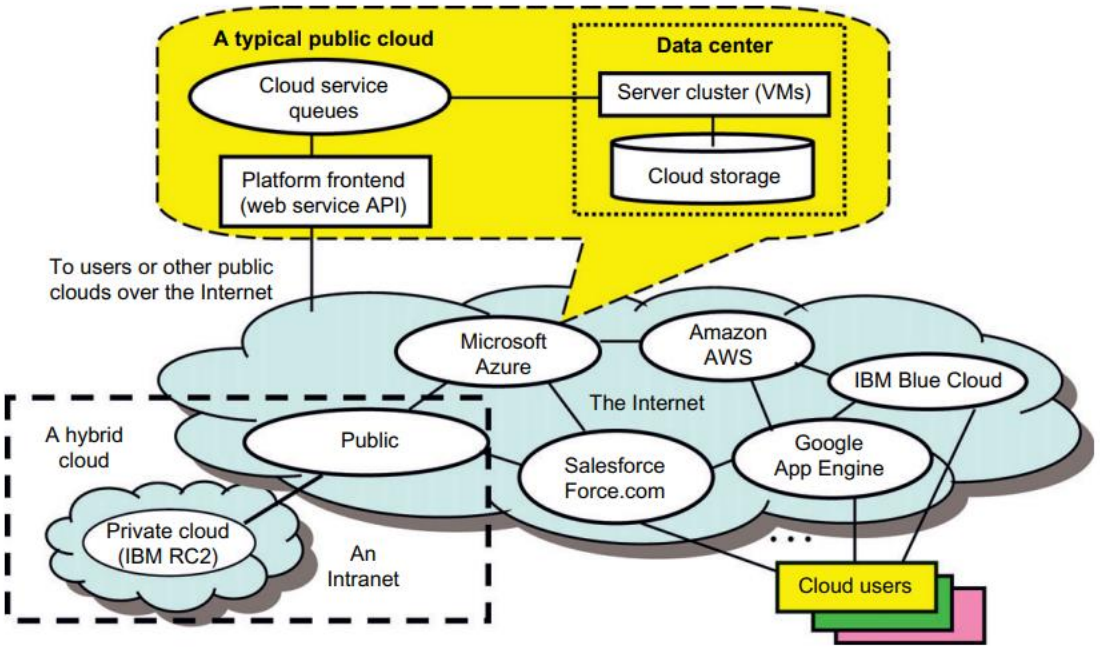
2. **Fault Tolerance:**
    - Client-side caching enhances fault tolerance by **reducing dependency on the server's availability.** If a client has a local copy of a file, it can access and work with that file **even if the server is temporarily unavailable.**
    - The cache coherence in Coda is maintained through a mechanism called **call-backs.**

3. **Call-back Mechanism:**
    - When a client updates its local copy of the file, it notifies the server, and the **server sends an invalidation message** (call-back break) to other clients. This means the **server discards the call-back promise it had for those clients.**

In summary, client-side caching in Coda involves storing entire files locally on clients for improved **scalability and fault tolerance.** The **call-back mechanism helps maintain cache coherence** and ensures that **clients can safely use their local copies of files while staying synchronized with the server.**

CHAPTER 05

1. DIFFERENTIATING PUBLIC CLOUD, PRIVATE CLOUD, AND HYBRID CLOUD



Public Cloud:

- Public Cloud is a cloud computing environment where **cloud resources and services are owned and operated by a third-party cloud service provider.**
- These **resources are made available to the public and multiple organizations, usually over the internet.**
- **Users can access and utilize services on a pay-as-you-go basis,** and the **provider is responsible for infrastructure management and maintenance.**

Private Cloud:

- Private Cloud is a cloud environment that is **exclusively used by a single organization.**
- It can be hosted on-premises or by a third-party provider, but the key feature is the **dedicated and isolated nature of the cloud resources.**
- Private Cloud offers more **control, security, and customization** to the organization.

Hybrid Cloud:

- Hybrid Cloud is a **combination of both public and private cloud environments.**
- It allows **data and applications to be shared between them,** offering more **flexibility and scalability.**
- **Organizations can choose where to run workloads based on specific needs.**

Comparison Table:

Aspect	**Public Cloud**	**Private Cloud**	**Hybrid Cloud**
Ownership	Owned and operated by third-party provider	Owned and operated exclusively by a single organization	Combination of both public and private environments
Accessibility	Accessible over the internet by multiple organizations	Accessed over a secure network, often restricted to a single organization	Combines internet accessibility and private network access
Cost Model	Pay-as-you-go, variable costs	Upfront capital expenditure, fixed costs	Variable costs for public cloud resources, fixed costs for private resources
Control	Limited control by the organization	Full control and customization	Varied control, organization retains control over private resources
Security	Security managed by the provider	Organization has greater control and responsibility	Security varies, organization can secure its private resources
Use Cases	Suited for scalable, less-sensitive workloads	Ideal for highly sensitive, critical workloads	Optimal for scenarios requiring flexibility and security
Examples	AWS, Azure, Google Cloud, IBM Cloud, etc.	Private data centres, on-premises cloud solutions	Combining AWS with on-premises data centres

2. DEMONSTRATE INFRASTRUCTURE-AS-A-SERVICE (IAAS) WITH EXAMPLE:

Definition:

Infrastructure-as-a-Service (IaaS) is a cloud computing service model that **provides virtualized computing resources over the internet.** In an IaaS model, **clients can rent virtualized hardware resources, including servers, storage, and networking,** on a pay-as-you-go basis. It offers flexibility, scalability, and **eliminates the need for organizations to invest in and maintain physical hardware.**

Demonstration of IaaS with Example:

Scenario:

Consider a small business, **"Tech Solutions,"** that requires computing infrastructure to host its website.

Steps in IaaS Implementation:

1. **Select an IaaS Provider:**
    - Tech Solutions chooses an IaaS provider like Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP).
2. **Register and Access the Console:**
    - Tech Solutions registers an account with the chosen IaaS provider and **gains access to the provider's management console.**
3. **Specify Infrastructure Requirements:**
    - In the console, Tech Solutions specifies its infrastructure requirements:
        - Two virtual machines (VMs) with 2GB RAM and 50GB storage each.
        - Load balancer for distributing incoming web traffic.
4. **Provider Allocates Resources:**
    - The IaaS provider **allocates the requested virtual resources using its underlying hypervisor.**
    - Virtual machines and the load balancer are provisioned.
5. **Access and Configuration:**
    - **Tech Solutions receives access credentials to the VMs and configures them:**
        - Installs a Linux operating system.
        - **Configures Apache web server on both VMs.**
6. **Deploy Web Application:**
    - Tech Solutions deploys its website on the configured VMs.
    - The **load balancer ensures even distribution of traffic between the VMs.**
7. **Monitor and Scale (Optional):**
    - Using the IaaS provider's monitoring tools, Tech Solutions tracks the website's performance.
    - If the website gains popularity, Tech Solutions can dynamically scale by adding more VMs for increased capacity.

Benefits of IaaS in this Example:

- **Flexibility:** Tech Solutions can configure VMs according to its specific needs.
- **Scalability:** Additional resources can be provisioned easily to handle increased demand.

- **Cost-Effective:** Tech Solutions pays only for the resources it uses, without the need for upfront hardware investments.

In summary, IaaS allows Tech Solutions to focus on its web application without worrying about the underlying infrastructure, promoting agility and cost-effectiveness.

3. DEMONSTRATE PLATFORM-AS-A-SERVICE (PAAS) WITH A SMALL AND SIMPLE EXAMPLE.

Platform-as-a-Service (PaaS):

Definition: Platform-as-a-Service (PaaS) is a cloud computing service model that provides a platform **allowing developers to develop, run, and manage applications** without dealing with the complexities of infrastructure. PaaS typically **includes tools and services for application development, such as databases, middleware, and frameworks.**

Demonstration of PaaS with Example:

Google App Engine (GAE) Overview: Google App Engine is an example of PaaS that allows users to build and deploy web applications without managing the underlying infrastructure. **Here's how it works:**

1. **Development Environment:**
    - Developers write and test their applications using a fully-featured local development environment provided by Google. This environment simulates the GAE infrastructure on the developer's computer.
2. **Supported Languages:**
    - GAE supports multiple programming languages, including Python and Java. Developers can choose their preferred language for application development.
3. **Coding and Debugging:**
    - All coding and debugging stages can be performed locally, similar to traditional software development. Developers implement functions and application logic on their local machines.
4. **Automatic Scaling and Load Balancing:**
    - Applications running on GAE benefit from features like automatic scaling and load balancing. This is convenient for building web applications that may experience varying levels of demand.
5. **Distributed Scheduler Mechanism:**
    - GAE includes a distributed scheduler mechanism that schedules tasks for triggering events at specified times and regular intervals. This mechanism enhances the operational model for GAE.
6. **Deployment to Google's Infrastructure:**

- Once development is complete, developers use the GAE SDK (Software Development Kit) to upload their applications to Google's infrastructure. This is where the applications are actually deployed for public access.

7. **Additional Third-Party Capabilities:**
   - GAE encourages third parties to provide software management, integration, and service monitoring solutions. Users can leverage these additional capabilities to enhance their applications.

Google App Engine exemplifies PaaS by providing developers with a platform to build and deploy web applications effortlessly, abstracting the complexities of infrastructure management.

4. DEMONSTRATE SOFTWARE-AS-A-SERVICE (SAAS) WITH A SMALL AND SIMPLE EXAMPLE.

Software-as-a-Service (SaaS):

Definition: Software-as-a-Service (SaaS) is a cloud computing service model that **delivers software applications over the internet on a subscription basis.** Instead of users **installing, maintaining, and updating software** on their local devices, they access the software through a web browser.

Demonstration of SaaS with Example:

Scenario: Consider a business, "TechDocs Inc.," in need of a **document collaboration and editing tool for its employees.** Instead of installing software on each computer, TechDocs opts for a SaaS solution.

Steps in SaaS Implementation:

1. **Select a SaaS Provider:**
   - TechDocs chooses a SaaS provider such as **Google Workspace, Microsoft 365, or Dropbox Paper.**
2. **Sign Up and Access the SaaS Platform:**
   - TechDocs signs up for an account with the chosen SaaS provider.
   - The employees gain access to the document collaboration platform through a web browser.
3. **Use the SaaS Application:**
   - Employees can create, edit, and collaborate on documents directly within the SaaS application.
4. **Cloud Storage and Accessibility:**
   - Documents are stored in the cloud, eliminating the need for local storage.

    - Employees can access their documents from any device with an internet connection.

5. **Real-time Collaboration:**
    - Multiple employees can simultaneously collaborate on a document in real-time.
    - Changes made by one user are instantly reflected for others.

Benefits of SaaS in this Example:

- **No Software Installation:** TechDocs doesn't need to install software on individual computers.
- **Automatic Updates:** The SaaS provider handles software updates and ensures all users have the latest features.
- **Scalability:** As TechDocs grows, it can easily scale the number of user subscriptions with the SaaS provider.
- **Remote Access:** Employees can work on documents from anywhere, promoting flexibility.

In summary, SaaS provides TechDocs Inc. with a **convenient, accessible, and collaborative document editing solution** without the burden of managing software installations and updates locally.

5. ILLUSTRATE CLOUD ARCHITECTURAL DESIGN CHALLENGES.

Cloud Architectural Design Challenges

Cloud architecture development faces several challenges that need to be addressed for the effective design and implementation of cloud computing platforms. Here are six key challenges along with potential solutions:

1. **Challenge 1 — Service Availability and Data Lock-in Problem:**
    - **Solution:** Use **multiple cloud providers for high availability (HA)** and **standardize APIs** to enable **data and service deployment** across providers. This **mitigates the risk of data lock-in.**
2. **Challenge 2 — Data Privacy and Security Concerns:**
    - **Solution:** Implement technologies such as **encrypted storage, virtual LANs, and network middleboxes.**
3. **Challenge 3 — Unpredictable Performance and Bottlenecks:**
    - **Solution:** **Improve I/O architectures and operating systems** for efficient virtualization of interrupts and I/O channels.

4. **Challenge 4 — Distributed Storage and Widespread Software Bugs:**
    - **Solution: Develop scalable, durable, and highly available storage systems. Use VMs for debugging in cloud computing.**
5. **Challenge 5 — Cloud Scalability, Interoperability, and Standardization:**
    - **Solution: Implement scalable pay-as-you-go models,** explore virtual appliance standards like Open Virtualization Format (OVF), and address challenges in cross-platform live migration.
6. **Challenge 6 — Software Licensing and Reputation Sharing:**
    - **Solution: Explore pay-for-use and bulk-use licensing schemes,** encourage **open-source adoption,** and establish reputation-guarding services to protect against malicious activities.

Addressing these challenges is crucial for the **continued growth and success of cloud computing, ensuring robust, secure,** and **efficient cloud architectures.**

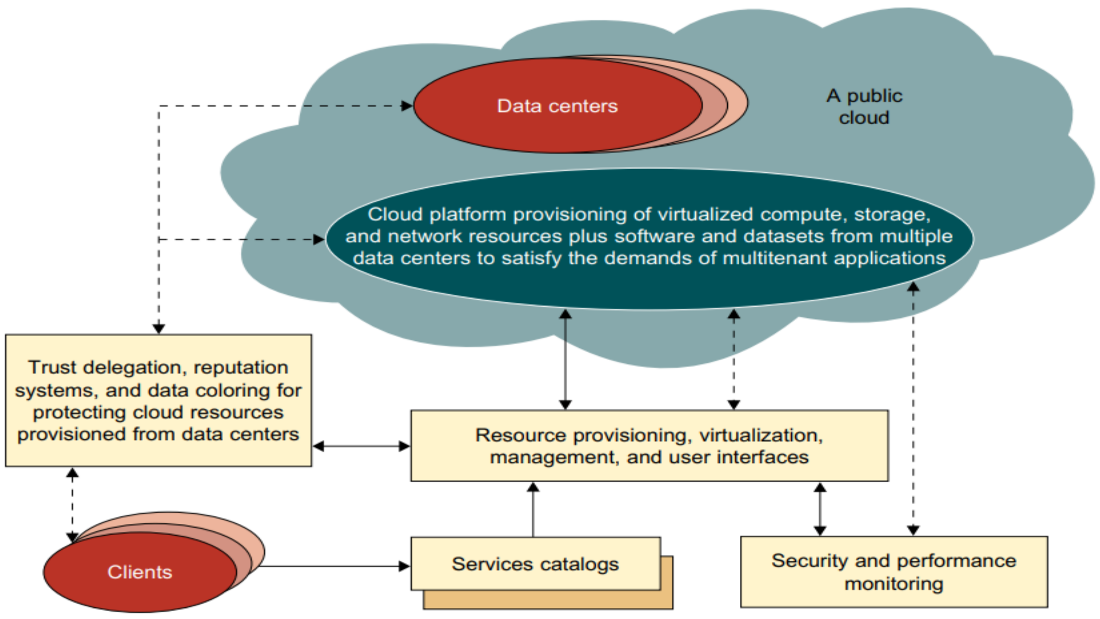
6. DEMONSTRATE SECURITY-AWARE CLOUD ARCHITECTURE.

Security-Aware Cloud Architecture:

The security-aware cloud architecture, as illustrated in Figure 4.14, is designed to address the challenges associated with the provisioning, management, and utilization of cloud resources. Here are the key elements and features of this architecture:

1. **Envisioning the Internet Cloud:**
    - The Internet cloud is envisioned as a massive cluster of servers.
    - These servers are provisioned on demand to collectively deliver web services or distributed applications using data-centre resources.
2. **Dynamic Cloud Platform Formation:**
    - The cloud platform is formed dynamically through the provisioning or deprovisioning of servers, software, and database resources.
    - Both physical machines and virtual machines (VMs) can serve as servers in the cloud.
3. **User Interfaces and Provisioning Tool:**
    - User interfaces are employed to request services from the cloud.
    - A provisioning tool is utilized to carve out the cloud system dynamically to deliver the requested services.
4. **Integration with Data Centres:**
    - Cloud computing resources are integrated into data centres, which are typically owned and operated by third-party providers.

- Consumers are abstracted from underlying technologies, focusing on software as a service.



Data centers
A public cloud
Cloud platform provisioning of virtualized compute, storage, and network resources plus software and datasets from multiple data centers to satisfy the demands of multitenant applications
Trust delegation, reputation systems, and data coloring for protecting cloud resources provisioned from data centers
Resource provisioning, virtualization, management, and user interfaces
Clients
Services catalogs
Security and performance monitoring

FIGURE 4.14

5. **High Trust and Massive Data Handling:**
    - The cloud demands a high degree of trust for handling massive amounts of data retrieved from large data centres.
    - A framework is needed to process large-scale data stored in the distributed storage system over the database.
6. **Incorporation of Various Cloud Resources:**
    - Distributed storage, storage area networks (SANs), database systems, firewalls, and security devices are integrated into the cloud platform.
    - Web service providers offer special APIs to enable developers to exploit Internet clouds.
7. **Cloud-Enabling Technologies:**
    - Cloud-enabling technologies in hardware, software, and networking are employed for fast platform deployment, virtual clusters on demand, multitenant techniques, massive data processing, web-scale communication, distributed storage, and licensing/billing services.
8. **Monitoring and Metering:**
    - Monitoring and metering units are employed to track the usage and performance of provisioned resources.

9. **Automated Resource Management:**
    - The software infrastructure of the cloud platform handles resource management and maintenance automatically.
    - It detects the status of each server node joining or leaving and performs relevant tasks accordingly.
10. **Considerations in Cloud Physical Platform Building:**
    - Cloud computing providers prioritize the performance/price ratio and reliability over sheer speed performance.
    - Data centres are often strategically located to reduce power and cooling costs, with considerations like hydroelectric power.
11. **Private, Public, and Hybrid Cloud Trends:**
    - Private clouds are highlighted as easier to manage, while public clouds are easier to access.
    - The trend in cloud development is towards more hybrid clouds due to the necessity of cloud applications extending beyond intranet boundaries.
12. **Security as a Critical Concern:**
    - Security is emphasized as a critical issue in safeguarding the operation of all cloud types.
    - Specific attention is directed towards studying cloud security and privacy issues at the conclusion of the chapter.

In summary, the security-aware cloud architecture outlined above underscores the importance of dynamic resource provisioning, data security, and the integration of various technologies to ensure efficient and secure cloud computing.

Here are five major cloud platforms and their key service offerings:

1. **Amazon Web Services (AWS)**
    - Infrastructure as a service (EC2, VPCs, storage)
    - Platform as a service (Elastic Beanstalk)
    - Serverless computing (Lambda, DynamoDB)
    - Wide range of computing, database, analytics, networking, mobile, and devops services
2. **Microsoft Azure**
    - Public, private and hybrid cloud solutions
    - Infrastructure as a service (VMs, storage, networking)

- Platform as a service (.NET framework, Azure SQL Databases)
- Comprehensive set of cloud services including AI, analytics, databases, DevOps tools

3. **Google Cloud Platform (GCP)**

- Infrastructure (compute engine, networking), data storage
- Platform as a service (App Engine, Kubernetes Engine)
- Artificial intelligence and machine learning
- BigQuery for data analytics and data warehousing

4. **IBM Cloud**

- Public, private and hybrid cloud models
- IaaS (bare metal servers, VMs, storage, VPN)
- PaaS (Cloud Foundry, OpenShift Container Platform)
- Artificial intelligence (Watson) and IoT solutions

5. **Oracle Cloud**

- IaaS (compute, storage, networking, data canters)
- PaaS for databases, app development and big data
- SaaS (ERP, HCM, marketing automation)
- Maximum security for enterprise apps

In summary, major cloud providers offer infrastructure, platform, and application services with global scale, pay-as-you-go pricing, and enterprise-grade security and reliability.