1. What is the classical definition of a distributed system?
Ans:

- A collection of independent computers that appears to its users as a single coherent system.
- A collection of independent computers that appears to its users as a single coherent system.

2. What is the definition of a centralized system? Give an example of a centralized system.
Ans:

DEF: a centralized system typically refers to a setup where there's a single point of control or coordination for a distributed network. This central point may handle tasks such as resource allocation, data management, or decision-making for the entire system.

Example: small office network where all computers are connected to a central server

- File storage and sharing: All files are stored on the central server, and users access and share files through it.
- User authentication and access control: The server manages user accounts and permissions, allowing or denying access to resources based on user credentials.
- Internet access: The server could also act as a gateway to the internet, controlling and monitoring internet traffic for all connected devices.
- Email services: The server hosts email accounts for users within the network, handling incoming and outgoing email communications.
- Backup and data recovery: The server could be responsible for regular backups of data stored on the network and facilitating data recovery in case of loss.

3**. On the topic of distributed vs decentralized system, what is the definition of a distributed system? Give an example of a distributed system.**
Ans:

- **DEF**: A collection of independent computers that appears to its users as a single coherent system."
- A collection of independent computers that appears to its users as a single coherent system.

Example: Email service, because the processes and resources in such a system are spread across multiple computers, due to the increase or decrease of users using the service.

4. On the topic of distributed vs decentralized system, what is the definition of a decentralized system? Give an example of a decentralized system.
Ans:
Def: A decentralized system is a networked computer system in which processes and resources are necessarily spread across multiple computers.

example: A distributed ledger or a blockchain, because the processes and resources in such a system are spread across multiple computers, due to lack of trust.

5. What are the design goals for building distributed systems?
Ans:

1. Resource Sharing
2. Distribution Transparency
3. Openness
4. Dependability
5. Security
6. Scalability

Resource Sharing: Enabling multiple users or processes to access and utilize shared resources such as files, hardware, or software concurrently.

Distribution Transparency: Concealing the complexities of a distributed system from users and applications, providing a seamless interface regardless of the underlying network structure.

Openness: Allowing interoperability and accessibility by adhering to open standards and protocols, fostering integration and collaboration among different systems.

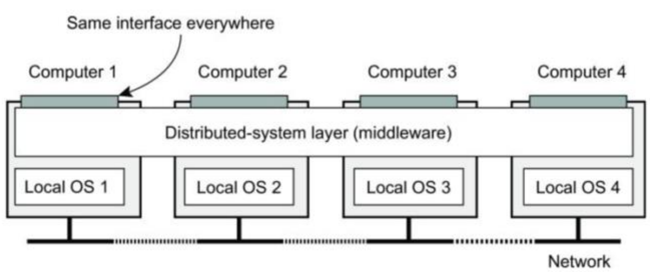
Dependability: Ensuring the reliability, availability, and fault tolerance of a system to deliver consistent and predictable performance under varying conditions.

Security: Safeguarding data, resources, and communications from unauthorized access, modification, or disruption, maintaining confidentiality, integrity, and availability.

Scalability: Ability to accommodate increasing demands by efficiently expanding resources or adapting the system to handle larger workloads without compromising performance or functionality.

6. What are the **types of distribution transparency** found in distributed systems? Name and define them all.

Distribution transparency



What is transparency?

The phenomenon by which a distributed system attempts to hide the fact that its processes and resources are physically distributed across multiple computers, possibly separated by large distances.

Observation

Distribution transparancy is handled through many different techniques in a layer between applications and operating systems: a middleware layer

Distribution transparency

Types

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

7. What are the **characteristics of openness** in distributed systems? Name them and define them all.

Ans:

- **Interoperability**: Different systems can seamlessly communicate and work together.
- **Accessibility**: Resources and services are easily reachable and usable by users or entities.
- **Transparency**: Operations, behaviors, and policies of the system are clearly visible and understandable.
- **Extensibility**: The system can easily accommodate new functionalities or technologies.
- **Standards Compliance:** Adherence to established protocols ensures compatibility and consistency across implementations.

V2:

- **Interoperability**: This characteristic ensures that diverse systems, regardless of their underlying technologies or platforms, can exchange data and services seamlessly. It enables integration and cooperation among different components within the distributed system, promoting flexibility and scalability.
- **Accessibility**: In an open distributed system, accessibility ensures that resources, services, and information are readily available and usable by authorized users or entities. It emphasizes removing barriers to entry and ensuring equitable access to system functionalities, regardless of factors such as location or device type.
- **Transparency**: Transparency entails providing visibility into the inner workings of the distributed system, including its processes, algorithms, and decision-making mechanisms. It fosters trust and accountability among stakeholders by allowing them to understand how the system operates, how data is managed, and what rules govern its behavior.
- **Extensibility**: Extensibility refers to the ease with which the distributed system can be expanded or modified to accommodate new requirements, functionalities, or technologies. It involves designing the system architecture in a modular and flexible manner, allowing for the seamless integration of additional features without disrupting existing services.

- **Standards Compliance**: This characteristic involves adhering to industry-standard protocols, specifications, and conventions within the distributed system. By following established standards, the system ensures compatibility, interoperability, and consistency across different implementations. It promotes consistency in communication protocols, data formats, and security measures, facilitating collaboration and integration among disparate components or subsystems.

8. What are the useful **requirements covered by dependability** in distributed systems? Name them and define them all.

Ans:

- **Availability**: The likelihood that a system is operational and ready to be used immediately at any given moment, indicating its readiness to perform functions without downtime.

- **Reliability**: The capability of a system to operate continuously without failure over a period of time, highlighting its ability to sustain functionality without interruption.

- **Safety**: Ensuring that even when a system temporarily malfunctions, it does not lead to catastrophic consequences. This is particularly crucial in critical systems such as those used in nuclear power plants or space missions.

- **Maintainability**: The ease with which a failed system can be repaired or restored to operation. A highly maintainable system facilitates quick detection and repair of failures, potentially contributing to its overall availability.

9. In distributed systems, what is the **difference between a failure, an error and a fault**? Give the definition and an example for each.

Ans: A system is said to fail when it cannot meet its promises. In particular, if a distributed system is designed to provide its users with several services, the system has failed when one or more of those services cannot be (completely) provided.
For eg: a crashed program is clearly a failure, which may have happened because the program entered a branch of code containing a programming bug.

Error: An error is a part of a system's state that may lead to a failure. For example, when transmitting packets across a network, it is to be expected that some packets have been damaged when they arrive at the receiver. Damaged in this context means that the receiver may incorrectly sense a bit value (e.g., reading a 1 instead of a 0), or may even be unable to detect that something has arrived.
Fault: The cause of an error is called a fault. Clearly, finding out what caused an error is important. For example, a wrong or bad transmission medium may easily cause packets to be damaged. In this case, it is relatively easy to remove the fault. However, transmission errors may also be caused by bad weather conditions, such as in wireless networks. Changing the weather to reduce or prevent errors is a bit trickier.

10. On the topic of security in distributed systems what is the **difference between confidentiality and integrity**? Give the definition of each.

Ans:

Aspect	Confidentiality	Integrity
Definition	Information is disclosed only to authorized parties	Alterations to assets can only be made in authorized ways
Focus	Protecting the secrecy of information	Ensuring the accuracy and trustworthiness of data
Concern	Unauthorized access or disclosure of information	Unauthorized modification or tampering with data
Goal	Preventing data breaches and unauthorized access	Preventing unauthorized alterations or corruption
Examples	Encryption, access control, data classification	Data validation, checksums, digital signatures

In summary, confidentiality focuses on protecting the secrecy of information, while integrity focuses on ensuring the accuracy and trustworthiness of data. Confidentiality prevents unauthorized access or disclosure of information, while integrity prevents unauthorized modification or tampering with data. Examples of measures to achieve confidentiality include encryption and access control, while measures for integrity include data validation and digital signatures.

11. On the topic of security in distributed systems **what is the difference between authentication, authorization and trust?** Give the definition of each.

Ans:

- **Authentication**:
  Definition: Verifying the claimed identity of an entity.
  Importance: Ensures only legitimate users access the system.

  Authorization:
  Definition: Determining if an authenticated entity has permission to perform actions or access resources.
  Importance: Controls what authenticated users can do within the system.

- **Trust**:
  Definition: Confidence in the integrity and reliability of entities or processes.
  Importance: Establishes reliability in interactions and communications, influencing decisions regarding access and collaboration.

In essence, authentication confirms identity, authorization grants access rights, and trust ensures reliability in distributed systems.

12. Give example of **two mechanisms** that can be used to **secure distributed systems**.

Ans:

Cryptography:
Essential for securing distributed systems by encrypting and decrypting data using keys.
In symmetric cryptography, the same key is used for both encryption and decryption.
In asymmetric cryptography, different keys are used: a public key for encryption and a private key for decryption.

Ensures data confidentiality and integrity by preventing unauthorized access and tampering.

Digital Signatures:
Used to verify the authenticity and integrity of digital data.
Implemented using asymmetric cryptography, where a signer uses their private key to sign data.
Others can verify the signature using the signer's public key.
Provides assurance that the data originated from the claimed sender and has not been altered.
Often combined with hash functions to efficiently generate and verify signatures.
Crucial for establishing trust and security in distributed systems by enabling parties to authenticate each other and validate the integrity of transmitted data.

13. What is the **difference between symmetric and asymmetric** key encryption?

Ans:

Symmetric Key Encryption	Asymmetric Key Encryption
It only requires a single key for both encryption and decryption.	It requires two keys, a public key and a private key, one to encrypt and the other one to decrypt.
The size of cipher text is the same or smaller than the original plain text.	The size of cipher text is the same or larger than the original plain text.
The encryption process is very fast.	The encryption process is slow.
It is used when a large amount of data is required to transfer.	It is used to transfer small amounts of data.
It only provides confidentiality.	It provides confidentiality, authenticity, and non-repudiation.
The length of key used is 128 or 256 bits	The length of key used is 2048 or higher
In symmetric key encryption, resource utilization is low as compared to asymmetric key encryption.	In asymmetric key encryption, resource utilization is high.
It is efficient as it is used for handling large amount of data.	It is comparatively less efficient as it can handle a small amount of data.
Security is less as only one key is used for both encryption and decryption purpose.	It is more secure as two keys are used here- one for encryption and the other for decryption.
The Mathematical Representation is as follows- P = D (K, E(K, P)) where K –> encryption and decryption key P –> plain text D –> Decryption E(K, P) –> Encryption of plain text using K	The Mathematical Representation is as follows- P = D(Kd, E (Ke,P)) where Ke –> encryption key Kd –> decryption key D –> Decryption E(Ke, P) –> Encryption of plain text using encryption key Ke. P –> plain text
Examples: 3DES, AES, DES and RC4	Examples: Diffie-Hellman, ECC, El Gamal, DSA and RSA

14. What **command** can you use on a Linux system to **generate an asymmetric key pair**?

Ans:

On a Linux system, you can use the ssh-keygen command to generate an asymmetric key pair, specifically for SSH (Secure Shell) protocol. This command generates both the public and private keys. Here's how you can use it:

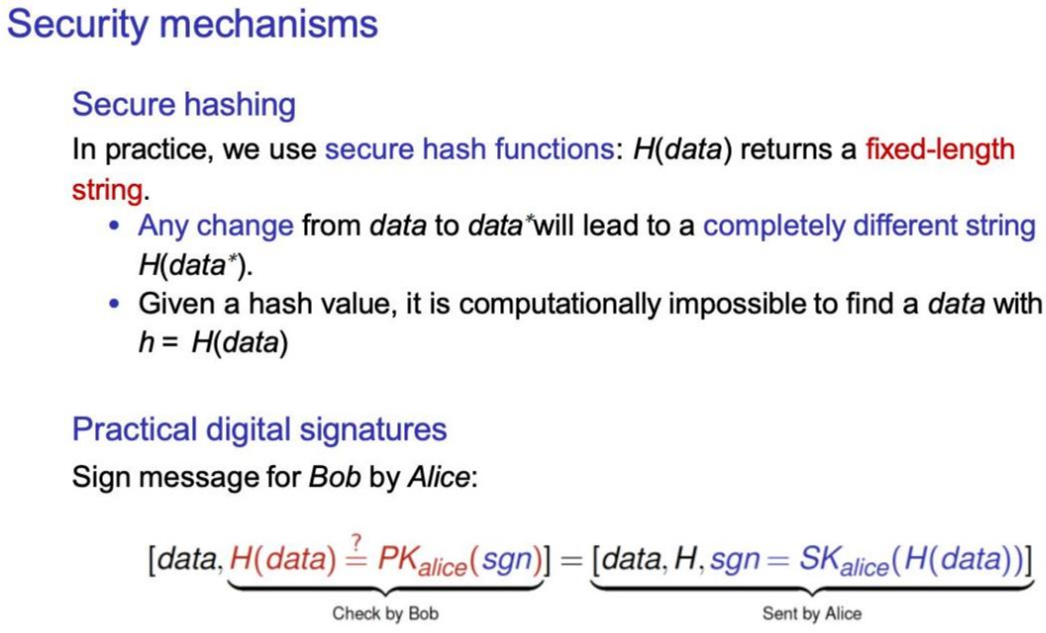
bash
ssh-keygen -t rsa -b 2048

Explanation of options:

-t rsa: Specifies the type of key to create, in this case, RSA.
-b 2048: Specifies the number of bits in the key. 2048 bits is commonly used for RSA keys.
After running this command, you'll be prompted to specify the location to save the keys and to optionally provide a passphrase for additional security. The generated keys will be saved in the

default location (~/.ssh/id_rsa for the private key and ~/.ssh/id_rsa.pub for the public key) unless you specify a different path.

15. What is the property that makes **secure hash functions secure**?



16. What are **the types of scalabilities found in distributed systems**? Give the definition and an example of each.

Ans:

Size Scalability:

Definition: A system can be scalable regarding its size, meaning that we can easily add more users and resources to the system without any noticeable loss of performance.

Example: Adding more servers to a web application to accommodate increasing user traffic without slowing down response times.

Geographical Scalability:

Definition: A geographically scalable system is one in which the users and resources may lie far apart, but the fact that communication delays may be significant is hardly noticed.

Example: A video conferencing platform that maintains smooth communication between users in different countries, despite varying network latencies.

Administrative Scalability:

Definition: An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organizations.

Example: A cloud infrastructure that allows multiple organizations to manage their resources independently while still adhering to overall governance and security policies.

17: Give example of **two techniques for scaling a distributed system**.

Ans:

1] Communication Latencies:

This technique aims to mitigate delays in communication, particularly in geographically distributed systems. Instead of waiting for responses from remote-service requests, applications are designed to perform other useful tasks locally while waiting for replies. This is achieved through asynchronous communication, where tasks are scheduled for execution independently, allowing the application to continue functioning while waiting for communication to complete. Alternatively, in cases where asynchronous communication isn't feasible, computation tasks can be shifted to the client side to reduce overall communication. This approach is particularly effective in interactive applications where users typically wait for responses. example: In an online chat application, messages are sent asynchronously, allowing users to continue typing new messages while waiting for responses, thereby reducing perceived latency.

2]Partitioning and Distribution:
Partitioning and distribution involve breaking down components or resources into smaller parts and dispersing them across the system. A prime example is the Internet Domain Name System (DNS), which organizes domain names into a hierarchical structure of non-overlapping zones. Each zone is managed by a single name server, facilitating efficient resolution of domain names to network addresses. This technique enables effective management and resolution of domain names in large-scale distributed systems.

18. What is the **difference between a shared memory and a distributed memory** parallel computing system?

Ans:
The main difference between a shared memory and a distributed memory parallel computing system lies in how they handle data access and communication between processing units (e.g., CPUs or cores):

Shared Memory System:

In a shared memory system, multiple processing units (e.g., CPUs or cores) have access to a single, global memory space.
All processing units can directly read from and write to this shared memory, enabling easy data sharing and communication between threads.
Synchronization mechanisms such as semaphores are used to control access to shared data and ensure data consistency.

Example: Multi-core processors, symmetric multiprocessing (SMP) systems.

Distributed Memory System:
In a distributed memory system, each processing unit (e.g., CPU or node) has its own local memory.
Processing units communicate and share data by explicitly sending messages to each other over a network.
Each processing unit operates independently and cannot directly access the memory of other units.
Synchronization and data sharing require explicit message passing between processing units.

Example: Cluster computing systems, supercomputers with multiple nodes.
In summary, shared memory systems provide a single, globally accessible memory space for all processing units, facilitating easy data sharing and communication through shared memory. In contrast, distributed memory systems have separate memory spaces for each processing unit, requiring explicit message passing for communication and data sharing between units.

19. What is one difference between a **computer cluster and a grid**?
Ans : One difference between a computer cluster and a grid is their level of heterogeneity:

Computer Cluster:

Typically consists of homogeneous systems with the same operating system (OS) and near-identical hardware.
Systems are tightly coupled and interconnected, often through a LAN (Local Area Network).
Managed by a single or a few tightly coupled managing nodes.

Grid Computing:

Involves heterogeneous resources, including systems with different operating systems, hardware configurations, and capabilities.
Nodes are dispersed across multiple organizations and locations, often spanning a wide-area network (WAN).
The grid computing infrastructure is designed to accommodate diverse resources and collaborate across organizational boundaries.

20. What does it mean for an **application to be CPU-intensive, memory-intensive and IO-intensive**. Give an example of each.

Ans: When we describe an application as CPU-intensive, memory-intensive, or I/O-intensive, we are referring to the primary resource it consumes the most during its execution. Here's what each term means:

CPU-Intensive:

An application is CPU-intensive if it primarily relies on the central processing unit (CPU) for its operations.
These applications typically involve complex computations, mathematical calculations, or algorithms that require significant CPU processing power.
Examples:
Cryptocurrency mining software that performs complex mathematical calculations to mine new coins.
Video rendering software that processes and renders high-definition video files, applying various effects and transformations.

Memory-Intensive:

An application is memory-intensive if it consumes a large amount of memory (RAM) during its execution.
These applications often manipulate large datasets, perform data processing, or store a significant amount of information in memory.
Examples:
Database management systems (DBMS) that store and manage large volumes of data in memory for fast access and retrieval.
Scientific simulations or modeling software that load extensive datasets into memory for analysis and processing.

I/O-Intensive:

An application is I/O-intensive if it relies heavily on input/output (I/O) operations to read from or write to storage devices, networks, or other external resources.
These applications frequently perform tasks such as reading/writing files, accessing databases, or communicating over networks.

Examples:
File compression or decompression utilities that read large files from disk, process them, and write the results back to disk.
Web servers that handle numerous incoming HTTP requests, read data from databases, and send responses over the network to clients.

In summary, CPU-intensive applications heavily utilize the CPU for computation, memory-intensive applications require significant memory resources for data processing, and I/O-intensive applications heavily rely on input/output operations for data access and communication with external resources.

21. What are the **common false** (and often hidden) **assumptions** when **developing** distributed systems?
Ans:

Developing distributed systems: Pitfalls

Observation

Many distributed systems are needlessly complex, caused by mistakes that required patching later on. Many false assumptions are often made.

False (and often hidden) assumptions

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

22. What is an **IP address** and what is it used for? What is the **TCP port used by SSH**?

Ans: IP (Internet Protocol) address: Every device connected to a computer network that communicates via the Internet Protocol is given an IP address, which is a numerical label. It performs two primary tasks: location addressing and host or network interface identification. IP addresses are necessary for data packet routing on a network.

UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) ports are used for network communication. By comparison, ports function as openings that allow information to enter and exit a computer. On a device, they aid in distinguishing between various services or apps. IP addresses and port numbers work together to route network traffic to the relevant application. Port numbers are used to identify ports.

Default SSH port: 22 is the port that SSH (Secure Shell) uses by default. The SSH service uses this port to listen for incoming connections.
To find the IP address of your virtual machine, ip a is command used by me.

23. Describe **how to enable SSH password-less authentication** on Linux machines.
Ans:
To enable SSH password-less authentication on Linux:

- Generate an SSH key pair: **ssh-keygen -t rsa**
- Copy the public key to the remote server**: ssh-copy-id username@remote_host**
- Verify password-less authentication: **ssh username@remote_host**
- Optionally configure SSH config file for automatic key usage.

24. What is the difference between the **CPU time and the wall time?**
Ans: **CPU Time:** Refers to the amount of time the CPU spends actively executing a program's instructions. It excludes time spent waiting for input/output operations or other system tasks.

Wall Time: Represents the total elapsed time from the start to the end of a program's execution. It includes all time, including CPU processing time, I/O wait time, and any other system delays or activities. It's essentially the time observed on a clock hanging on the wall.

27. Describe the **organization of the layered architectural** style found in distributed systems.
Ans: The basic idea for the layered style is simple: components are organized in a layered fashion where a component at layer Lj can make a downcall to a component at a lower-level layer Li (with i < j) and generally expects a response. Only in exceptional cases will an upcall be made to a higher-level component. The three common cases are shown in Figure 2.1.

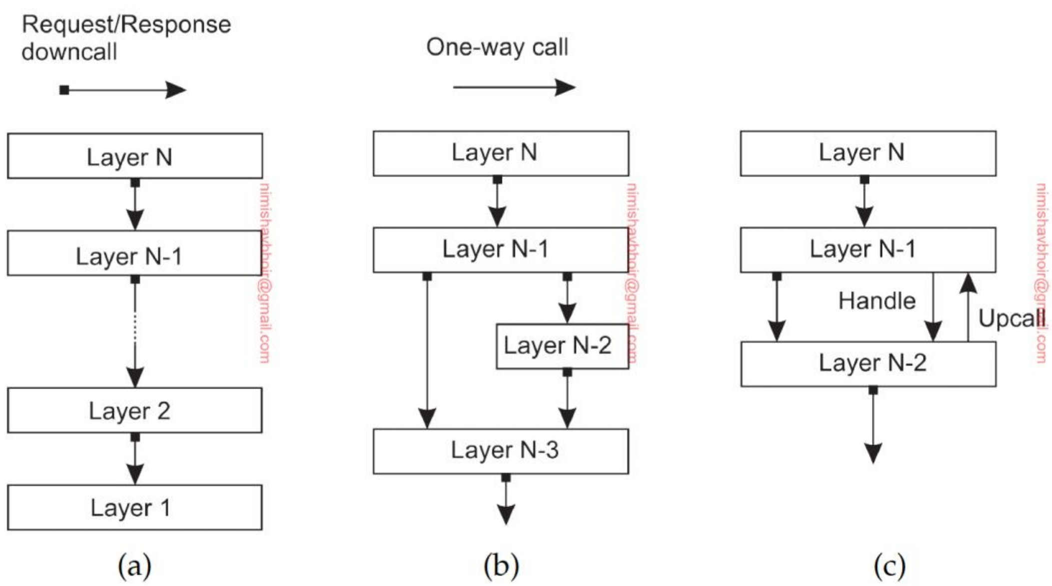


Figure 2.1: (a) Pure layered organization. (b) Mixed layered organization. (c) Layered organization with upcalls (adopted from [Krakowiak, 2009]).

Figure 2.1(a) shows a standard organization in which only downcalls to the next lower layer are made. This organization is commonly deployed in the case of network communication.

In many situations, we also encounter the organization shown in Figure 2.1(b). Consider, for example, an application A that makes use of a library LOS to interface to an operating system. At the same time, the application uses a specialized mathematical library Lmath that has been implemented by also making use of LOS. In this case, referring to Figure 2.1(b), A is implemented at layer N − 1, Lmath at layer N − 2, and LOS which is common to both of them, at layer N − 3.

Finally, a special situation is shown in Figure 2.1(c). In some cases, it is convenient to have a lower layer do an upcall to its next higher layer. A typical example is when an operating system signals the occurrence of an event, to which end it calls a user-defined operation for which an application had previously passed a reference (typically referred to as a handle).

28: Describe the **organization of the service-oriented architectural** style found in distributed systems.
Ans:
Service-oriented architectures
Although the layered architectural style is popular, one of its major drawbacks is the often strong dependency between different layers. Good examples where these potential dependencies have been carefully considered are found in designing communication protocol stacks. Bad examples include applications that have essentially been designed and developed as compositions of existing components without much concern for the stability of interfaces or the components themselves, let alone the overlap of functionality between different components. (A compelling example is given by Kucharski [2020], who describes the dependency on a simple component that pads a given string with zeroes or spaces. The author withdrew the component from the NPM library, leaving thousands of programs affected.)
Such direct dependencies to specific components have led to an architec- tural style reflecting a more loose organization into a collection of separate,independent entities. Each entity encapsulates a service. Whether they are called services, objects, or microservices, they have in common that the ser- vice is executed as a separate process (or thread). Of course, running separate entities does not necessarily lower dependencies in comparison to a layered architectural style.

29. Describe the organization of the **publish-subscribe architectural** style found in distributed systems.
Ans:
The described architectural approach emphasizes loose coupling between processes in distributed systems, separating processing from coordination. Coordination involves communication and cooperation between processes, serving as the glue that integrates their activities. Cabri et al. (2000) offer a taxonomy of coordination models categorized along temporal and referential dimensions.

In direct coordination, processes are both temporally and referentially coupled, requiring them to be active simultaneously for communication to occur. This type of coordination involves explicit referencing in communication, where processes must know each other's identifiers.

Mailbox coordination involves temporally decoupled but referentially coupled processes. Communication occurs by placing messages in shared mailboxes, eliminating the need for simultaneous activity. Processes explicitly address the mailbox for message exchange.

Event-based coordination applies to referentially decoupled but temporally coupled systems. Processes publish notifications about events, and other processes subscribe to specific notifications of interest. Ideally, notifications are delivered only to relevant subscribers, but the subscriber must be active when the notification is published.

Shared data space coordination involves referentially and temporally decoupled processes, utilizing a shared space for communication. Processes communicate through tuples, resembling structured data records. Retrieving tuples involves specifying search patterns, and any matching tuple is returned, implementing an associative search mechanism.

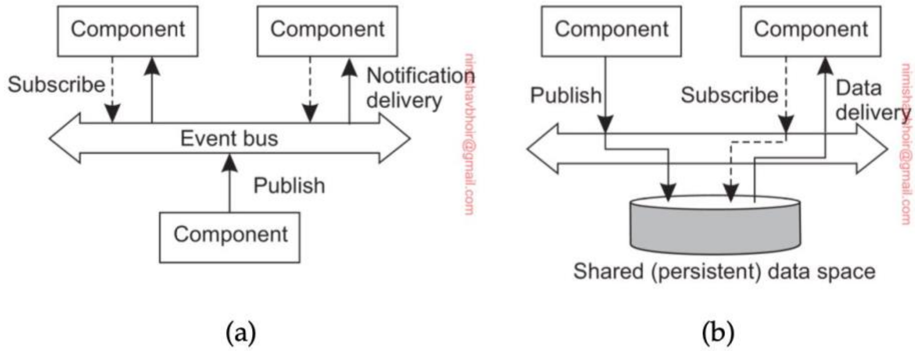


Figure 2.10: The (a) event-based and (b) shared data-space architectural style.

30. Describe **the layers found in a traditional three-layered** application. Name and define each layer.
Ans:
In the traditional three-layered view:

1. **Application-interface layer:** Serves as the interface between users or external systems and the distributed system, handling input and output.
2. **Processing layer:** Houses the core application logic, focusing on how tasks are executed without specific data, promoting modularity and reusability.
3. **Data layer:** Stores and manages the actual data manipulated by the system, ensuring persistence and integrity.

This architecture fosters modular design, maintenance ease, and scalability, making it widely adopted, particularly in systems using traditional database technology.

31. Name, define and give examples of each of the four coordination models found in publish-subscribe systems

Ans:

1. **Direct Model:**

Definition: In the direct model, processes communicate by directly invoking functions or methods on each other. This establishes a point-to-point connection between the communicating parties.

Example: A web server directly calls a function on a database server to retrieve data.

2. **Mailbox Model:**
Definition: The mailbox model utilizes mailboxes as intermediate communication channels. Processes send messages to mailboxes, and receiving processes periodically check these mailboxes for new messages. This model offers asynchronous communication, meaning the sender and receiver don't need to be active simultaneously.

Example: An email system uses mailboxes to store messages until recipients access them.

3. **Event-Based Model:**
Definition: In the event-based model, processes don't communicate directly. Instead, they publish events (signals indicating something has happened) to an event broker. Other processes can subscribe to these events and react accordingly. This enables loose coupling, where processes don't need to know about each other's details.

Example: A user interface might publish an "item_added" event when a user adds an item to a shopping cart. Different parts of the application (e.g., checkout system) can subscribe to this event and take necessary actions.

4. **Shared Data Space:**
Definition: The shared data space model involves a shared memory region accessible by multiple processes. Processes can directly read and write data to this shared space, enabling efficient communication for frequently accessed information. However, careful synchronization mechanisms are crucial to avoid data corruption.

Example: A multi-threaded application might use a shared data space to store application state or configuration data that needs to be accessed by multiple threads concurrently.

Temporal and referential coupling

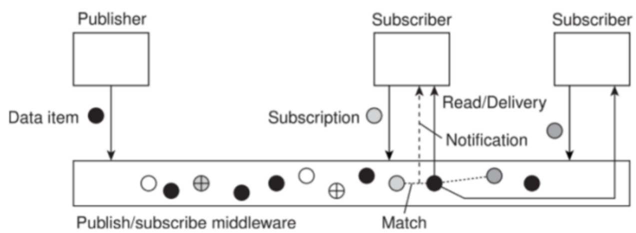
	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

32. What is the **main challenge** when **building publish-subscribe systems**?
Ans:

Publish and subscribe

Issue: how to match events?

- Assume events are described by (attribute,value) pairs
- topic-based subscription: specify a "attribute = value" series
- content-based subscription: specify a "attribute ∈ range" series



Observation
Content-based subscriptions may easily have serious scalability problems (why?)

The main challenge when building publish-subscribe systems lies in matching events, especially with content-based subscriptions. These systems can easily face serious scalability problems due to the complexity of event matching. Let's explore this further:

Publish-Subscribe Pattern:

- A publish-subscribe system allows different services in a system to communicate in a decoupled manner.
- The fundamental semantic feature of the publish-subscribe model lies in how messages flow from publishers to subscribers.

Challenges:

- **Content-Based Subscriptions:**

- Subscribers express interest in specific types of events based on their content (e.g., keywords, attributes).
- Matching events to subscribers' interests can be complex and resource-intensive.
- Scalability becomes a challenge when dealing with a substantial concurrent user base, frequent data publication, and varying user interests.

Solution Considerations:

Efficient event matching algorithms are crucial.

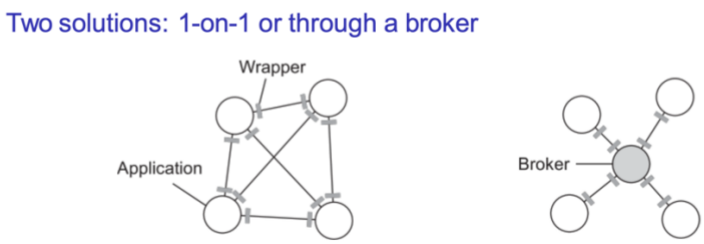
Systems must handle a large number of subscriptions and efficiently route events to relevant subscribers.

Remember, achieving efficient event matching while maintaining scalability is essential for robust publish-subscribe systems! 🕰️ 🌐

33.: Describe **two techniques** that can be used in the **<u>middleware</u>** to solve the problem of **<u>incompatible interfaces</u>** of legacy components in distributed systems.
Ans:

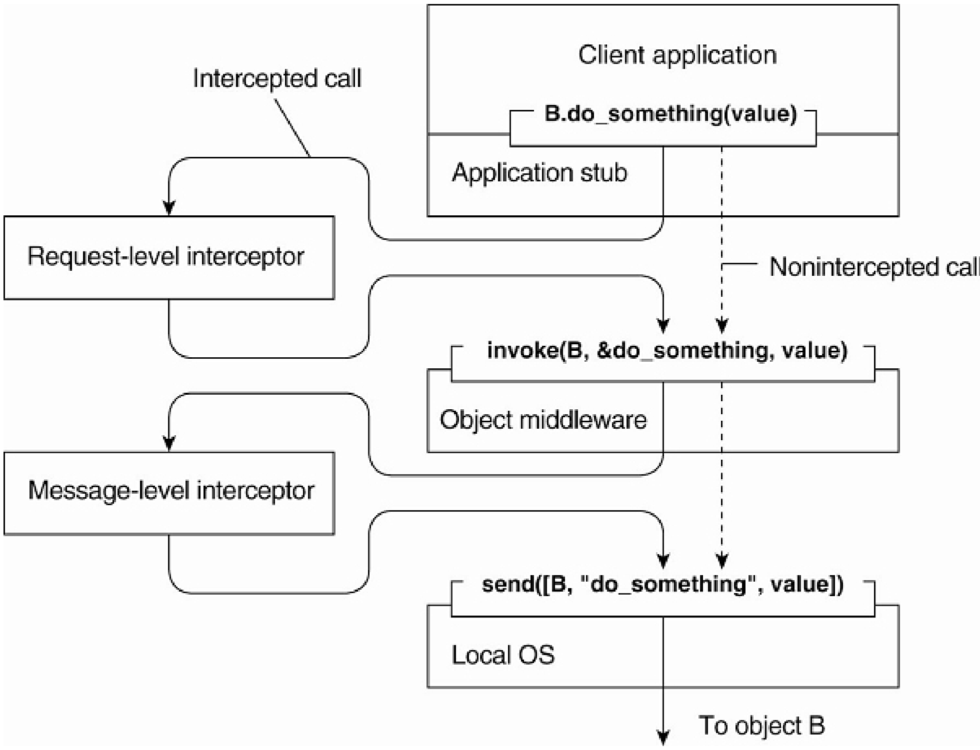
Wrappers / Adapters:

Use: Wrappers or adapters provide a standardized interface for legacy components, allowing them to seamlessly integrate with modern middleware systems. By translating messages between the standardized interface of the middleware and the specific interface of legacy components, they enable interoperability without requiring modifications to the legacy codebase.
Benefits: They shield the middleware from the intricacies of legacy systems, making integration easier and more manageable. Wrappers/adapters facilitate the reuse of legacy components within distributed systems by abstracting away interface incompatibilities.



Inceptors:

Use: Inceptors dynamically adapt communication between the middleware and legacy components by intercepting and modifying messages as they pass through the middleware layer. They enable real-time transformations, data format conversions, or protocol translations, ensuring seamless interaction between modern middleware and legacy systems.
Benefits: Inceptors allow for flexible and dynamic integration with legacy components, accommodating their interface requirements without modifying the legacy codebase. They facilitate interoperability and communication between distributed system components, even when legacy interfaces are incompatible with modern standards.



34. What are the characteristics of client-server layered-system architecture?
Ans:
Layered Structure: The architecture is organized into layers, with each layer handling specific tasks.
Client-Server Model: Clients request services or resources, and servers fulfill these requests over a network.
Client Responsibilities: Clients initiate requests and process responses, often providing user interfaces.
Server Responsibilities: Servers receive requests, process them, and return responses, providing services or resources.
Intermediary Layers: Additional layers may exist between clients and servers for communication or services.
Communication Protocols: Standardized protocols like HTTP define how clients and servers exchange data.
Scalability and Flexibility: The architecture allows for easy addition of clients or servers and supports diverse platforms.
Security Measures: Authentication, encryption, and access control are implemented to protect data.
Centralized Control: While distributed, there's often centralized control for managing resources and access.

35. What is the main challenge when building structured peer-to-peer distributed systems?
Ans: The main challenge in building structured peer-to-peer distributed systems lies in efficiently routing lookup requests to the node responsible for storing data associated with a given key. This routing process is crucial for enabling efficient data retrieval and relies heavily on the topology of the overlay network, such as a ring, binary tree, or hypercube. Additionally, ensuring that the system can handle dynamic changes in network topology, node failures, and load balancing while maintaining scalability and fault tolerance poses significant challenges.

36. Describe <u>two search techniques</u> that can be used in <u>unstructured peer-to-peer</u> distributed systems

Ans:

Two search techniques used in unstructured peer-to-peer distributed systems are Flooding and Random Walks:

Flooding:

Description: In flooding, an issuing node forwards a request for a data item to all its neighbors. Each receiving node searches locally for the requested data. If found, it responds directly or forwards it further. If not found, the request is forwarded to all of its own neighbors.

Use: Flooding is simple and ensures that the request eventually reaches a node with the data. However, it can be expensive due to high communication costs, especially with large networks.

Optimization: To mitigate costs, requests may have a time-to-live (TTL) value, limiting the number of hops allowed. Alternatively, starting with a low TTL and incrementally increasing it if needed can help balance search efficiency and cost.

Random Walks:
Description: In random walks, an issuing node selects a random neighbor and forwards the request. Each receiving node then randomly selects another neighbor to forward the request to, continuing until the data is found. Multiple random walks can be initiated simultaneously to decrease waiting time.

Use: Random walks reduce network traffic compared to flooding but may take longer to find the data. Initiating multiple walks concurrently can speed up the search process.

Termination: Random walks need to be stopped to prevent indefinite searching. This can be achieved using a TTL or by checking with the issuer if further forwarding is necessary upon receiving a request.

These search techniques offer trade-offs between communication cost, search efficiency, and network traffic, allowing for flexible adaptation to different system requirements and constraints in unstructured peer-to-peer distributed systems.

37. Give two examples of **<u>hybrid distributed systems architectures</u>**
Ans:

Two examples of hybrid distributed systems architectures are:

1. **Cloud Computing:**

Description: Cloud computing provides easily accessible pools of virtualized resources, allowing customers to dynamically configure and use resources as needed. It is characterized by scalability, pay-per-use models, and customizable service-level agreements.
Layers: Clouds are typically organized into four layers: hardware, infrastructure, platform, and application. These layers provide virtualized storage, computing resources, application development platforms, and software applications, respectively.
Services: Cloud computing offers Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) through various interfaces, enabling customers to outsource computing

infrastructures.

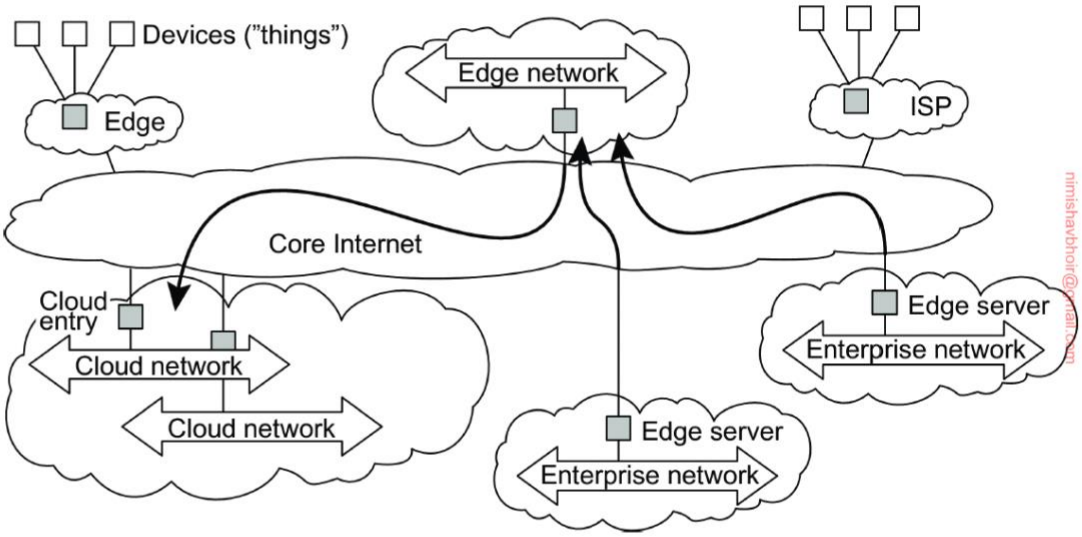


Figure 2.28: A collection of infrastructures involving edge devices, edge infrastructures and cloud infrastructures, and a possible setup between two enterprise edge infrastructures, an intermediate edge infrastructure, and a cloud infrastructure.

2. **Blockchain Systems:**

Description: Blockchain systems enable the registration and validation of transactions in a distributed ledger, eliminating the need for a central authority. Transactions are recorded in immutable blocks, forming a chain of validated transactions.
Trust Model: Unlike traditional transaction systems relying on trusted third parties like banks, blockchain systems assume a lack of trust among participants. Transactions are validated and stored by a decentralized network of participants, ensuring transparency and accountability.
Consensus Mechanisms: Blockchain systems require distributed consensus mechanisms to determine which participant can append a block of validated transactions to the existing chain. Consensus ensures global agreement on transaction validity and prevents double spending.

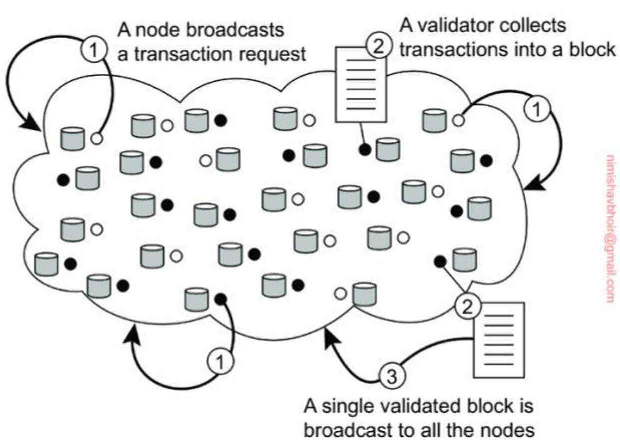


Figure 2.29: The principle operation of a blockchain.

38. What is the **difference between a process and a thread**? Give the definition for both. When is a thread more useful than a process? When is a process more useful than a thread?
Ans:
The basic difference between a process and a thread is that a **process is an independent unit of execution** with its own memory space, resources, and state, while a **thread is a lightweight unit of execution** within a process that shares the same memory space and resources as its parent process. In other words, processes are independent entities that run independently of each other, while threads are units of execution within a process that can run concurrently with other threads within the same process.

S.NO	Process	Thread
1.	Process means any program is in execution.	Thread means a segment of a process.
2.	The process takes more time to terminate.	The thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
6.	Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
7.	The process is isolated.	Threads share memory.
8.	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
9.	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
10.	If one process is blocked then it will not affect the execution of other processes	If a user-level thread is blocked, then all other user-level threads are blocked.
11.	The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.

Introduction to threads

Basic idea

We build virtual processors in software, on top of physical processors:

Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.

Thread: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.

Process: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

When is a thread more useful than a process for context switching?

Threads are more useful when context switching needs to be lightweight and efficient, as they share the same memory space within a process, resulting in faster context switches.
When is a process more useful than a thread for context switching?

Processes are more useful when isolation between tasks is crucial, as context switching between processes involves separate memory spaces and resources, ensuring better isolation but with higher overhead.

39. Which is **more expensive**, **a process context switch** or a **thread context** switch, and why?

Ans:
In general, a **process context switch is more expensive** than a thread context switch. This is because a process context switch involves more overhead and system resources compared to a thread context switch.

Here's why:

Memory: When switching between processes, the operating system must switch memory mappings, which involves updating page tables and swapping memory pages in and out of physical memory. This operation is relatively costly in terms of time and resources.

Resource Management: Processes have their own resources, such as file descriptors and memory allocations, which need to be managed during context switches. This management requires additional overhead compared to threads, which share resources within the same process.

Privilege Level: Switching between processes often involves transitioning between different privilege levels (e.g., user mode and kernel mode), which requires additional processor instructions and overhead.

Process Initialization: When switching between processes, the operating system may need to perform additional initialization steps, such as loading executable code and setting up the process environment, which adds to the overall cost of the context switch.

In contrast, a thread context switch involves switching between different execution contexts within the same process. Since threads share the same memory space and resources, the overhead involved in a thread context switch is generally lower compared to a process context switch.

40. What are **the advantages and disadvantages of implementing threads** in user-space?
Ans:
Advantages:

- Efficiency: User-space threads are faster for creation, management, and context switching as they avoid kernel overhead.
- Portability: They're more portable across different systems, simplifying development and deployment.
- Flexibility: Developers have more control over scheduling and communication, allowing tailored optimizations.
- Scalability: Can handle large numbers of threads more efficiently, improving performance in certain scenarios.
-

Disadvantages:

- Limited Concurrency: User-space threads may not utilize multiple CPU cores for parallel execution.
- Blocking System Calls: One thread's blocking call can stall the entire process, impacting responsiveness.
- Synchronization Overhead: Requires explicit synchronization mechanisms, introducing complexity and overhead.
- Debugging and Profiling: Can be more challenging compared to kernel-level threading due to limited visibility of thread-related events and states.

41. What are the **advantages and disadvantages of implementing threads in kernel-space**?
Ans:
Advantages:

- True Parallelism: Kernel-level threading allows threads to run concurrently on multiple CPU cores, leveraging true parallelism for enhanced performance.
- Improved Responsiveness: Blocking system calls in one thread do not affect others, ensuring better responsiveness in multi-threaded applications.
- Simplified Synchronization: Kernel provides synchronization primitives like mutexes and semaphores, simplifying thread coordination and reducing overhead.
- Transparent Debugging: Kernel-level threads are visible to system tools and debuggers, facilitating easier debugging and profiling.

Disadvantages:

- Performance Overhead: Kernel involvement introduces overhead for thread creation, context switching, and synchronization, impacting overall performance.
- Portability: Kernel-level threading implementations may not be as portable across different operating systems, leading to potential compatibility issues.
- Complexity: Developing and managing kernel-level threads requires deeper understanding of operating system internals, increasing complexity and potential for errors.
- Resource Management: Kernel-level threads may consume more system resources and impose stricter limits on the number of threads, potentially leading to resource contention and inefficiencies.

42. Explain how the **two-level idea**, where **kernel threads execute user threads, is better than user-space and kernel-space.** Explain how the two-level idea works.
Ans:
The two-level thread model, where kernel threads manage user threads, offers several advantages over purely user-space or purely kernel-space threading:

Advantages over User-Space Threads:

Kernel Support for Scheduling and Synchronization: User-space threads rely on libraries for scheduling and synchronization. This can be inefficient and error-prone. The kernel has direct access to hardware resources and can manage them more efficiently, leading to better performance and reduced risk of deadlocks.
System Call Blocking: When a user-space thread performs a system call, the entire thread is blocked, wasting resources. In a two-level model, the kernel thread can continue running other user threads while waiting for the system call to complete.
Advantages over Kernel-Space Threads:

Context Switching Overhead: Kernel threads are heavyweight, meaning switching between them is expensive. User threads are lighter and can be managed more efficiently within a kernel thread. This reduces context switching overhead, improving overall system performance.
Scalability: The two-level model allows for a larger pool of user threads than the kernel can handle efficiently on its own. This enables better utilization of multi-core processors.
Here's how the two-level model works:

User Threads: Applications create user threads. These threads represent the logical units of execution within the application.

Thread Library: A user-space library manages these user threads. It keeps track of their state (running, waiting, etc.) and provides APIs for creating, destroying, and synchronizing them.
Kernel Threads: The library maps user threads to a smaller pool of kernel threads. Each kernel thread can execute multiple user threads in a round-robin fashion.
System Calls: When a user thread needs to make a system call, the library context switches to the underlying kernel thread. The kernel thread then executes the system call and blocks if necessary. While blocked, the kernel thread can switch to another user thread.
Scheduling and Synchronization: The library manages scheduling and synchronization between user threads within a kernel thread. Kernel resources, like locks and semaphores, are used for synchronization.
This two-level approach provides a balance between flexibility and efficiency. Applications have fine-grained control over their threads, while the kernel ensures efficient resource management and system call handling.

43. Give an example on how **multithreading can be used to improve the performance** of the **client** in a client-server architecture.
Ans:
Consider a web browsing client like Chrome. Here's how multithreading can significantly improve its performance:

Downloading Resources Simultaneously: Imagine loading a webpage with multiple images, scripts, and stylesheets. A single-threaded client would have to download each resource sequentially, waiting for one to finish before starting the next. This can feel very slow, especially on slow connections.

With multithreading, the client can spawn multiple threads. One thread can handle the main HTML page download, while others can download images, scripts, and stylesheets concurrently. This significantly reduces the perceived loading time as the client displays content as it becomes available.

Making Multiple Server Requests: Webpages often rely on data from various sources like user profiles, advertisement banners, or analytics trackers. A single-threaded client would need to send separate requests and wait for each response before proceeding. This can be inefficient.

Multithreading allows the client to send these requests simultaneously using separate threads. While waiting for responses, other threads can continue processing the downloaded content. This improves the overall responsiveness of the client.

Background Tasks: Modern browsers often perform tasks in the background, like checking for updates, pre-fetching linked pages, or running extensions. These tasks shouldn't block the main thread responsible for rendering the webpage and user interaction.

Using separate threads for background tasks allows the browser to maintain a smooth user experience even while performing these operations.

Overall, multithreading in a client-server architecture enables the client to handle multiple tasks concurrently, improving responsiveness and perceived performance for the user.

44. Give an **example on how multithreading can be used to improve the performance of the <u>server</u>** in a client-server architecture.

Ans: In a client-server architecture, multithreading can significantly boost server performance by allowing it to handle multiple client requests concurrently.
Here's an example:

Scenario: Imagine a web server handling a surge of traffic for a popular online store during a sale.

Single-Threaded Server:

The server processes client requests one at a time.
When a request involves a long operation (e.g., database query, file upload), subsequent clients have to wait until the current request finishes.
This creates a queue of waiting requests, leading to slow response times and frustrated users.
Multi-Threaded Server:

The server maintains a pool of worker threads.
Each incoming client request is assigned to a free worker thread.
Worker threads can handle requests independently, allowing the server to serve multiple clients simultaneously.
If a thread encounters a long operation, it doesn't block other threads. The server can continue processing other requests while the long operation completes.
Benefits:

Improved Scalability: The server can handle more concurrent requests without a significant performance drop.
Reduced Response Times: Clients experience faster response times as requests are processed in parallel.
Increased Throughput: The server can process more requests per unit time, leading to a higher overall workload capacity.
Better Resource Utilization: CPU cores are kept busy processing requests, maximizing server hardware usage.
However, multithreading also introduces challenges:

Overhead: Creating and managing threads requires some overhead from the operating system.
Synchronization: If multiple threads access shared resources (e.g., data structures), proper synchronization mechanisms are needed to avoid data corruption.
Overall, multithreading is a powerful technique for improving server performance, especially for handling high volumes of concurrent client requests. However, careful planning and implementation are crucial to avoid potential issues.

46. What is the principle behind virtualization in operating systems?
Ans: In this diagram:

Physical Server: This represents the physical hardware, such as a server or a computer system.

Hypervisor: The hypervisor, also known as a virtual machine monitor (VMM), is a software layer that sits directly on top of the physical hardware. It abstracts and manages the hardware resources and facilitates the creation and management of virtual machines.

Virtual Machines (VMs): Virtual machines are individual instances of guest operating systems (OS) running on the same physical server. Each VM encapsulates an entire OS along with its applications and services, making them appear as independent systems.

VM1, VM2, VM3: These represent virtual machines created by the hypervisor. Each VM operates independently of others and can run different operating systems and applications. They share physical resources, such as CPU, memory, storage, and network interfaces, which are dynamically allocated by the hypervisor based on demand.

OS+App: Inside each virtual machine, there is an operating system along with the applications and services it hosts. This encapsulated environment provides isolation and ensures that processes running within one VM do not interfere with processes running in other VMs.

Through this diagram:

Certainly! Let's illustrate the principles of virtualization using a diagram of a typical virtualized environment:

lua   Copy code

```lua
+---------------------------------------+
|              Physical Server          |
| +-----------------------------------+ |
| |              Hypervisor           | |
| +-----------------------------------+ |
|         /              |            \ |
|        /               |             \|
| +------+     +--------+ | +-----------+ |
| | VM1  |     | VM2    | | | VM3       | |
| |(OS+App)     (OS+App)| | |(OS+App)   | |
| +------+     +--------+ | +-----------+ |
+---------------------------------------+
```

Abstraction: The hypervisor abstracts the physical hardware, presenting virtualized resources (CPU, memory, storage, etc.) to each virtual machine.

Isolation: Each virtual machine operates independently, isolated from other VMs. Any changes or failures within one VM do not affect others.

Resource Sharing: Physical resources are shared among virtual machines, with the hypervisor dynamically allocating resources based on workload demands.

Encapsulation: Virtual machines are encapsulated into files or disk images, making them portable and easy to deploy or migrate across different physical servers.

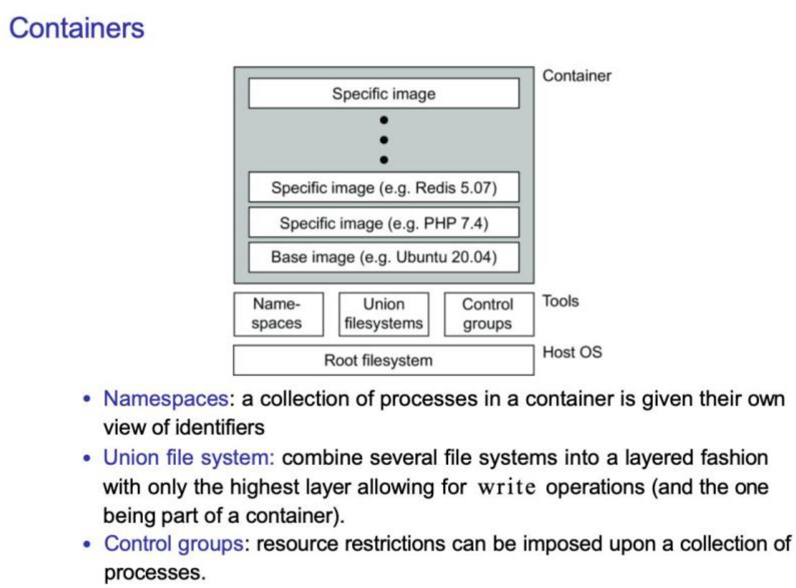
This diagram illustrates how virtualization enables efficient utilization of physical resources, improved scalability, flexibility, and simplified management of IT infrastructure.

47. What are the three different levels that virtualization can be implemented at and what are the three ways of building virtual machine monitors? Name and define each of them.
Ans: To understand the differences in virtualization, it is important to realize that computer systems generally offer four different types of interfaces, at three different levels:
1. An interface between the hardware and software, referred to as the in- struction set architecture (ISA), forming the set of machine instructions. This set is divided into two subsets:
• Privileged instructions, which are allowed to be executed only by the operating system.
• General instructions, which can be executed by any program.
2. An interface consisting of system calls as offered by an operating system.
3. An interface consisting of library calls, generally forming what is known as an application programming interface (API). Often, the aforemen- tioned system calls are hidden by an API.

Process Virtual Machine:
A virtual machine that runs as a user-level process within a host operating system, executing application code directly without translation or interpretation.
Native virtual machine: An alternative approach toward virtualization, shown in Figure 3.10(b), is to provide a system that is implemented as a layer shielding the original hardware, but offering the complete instruction set of that same (or other hardware) as an interface. This leads to what is known as a native virtual machine monitor. It is called native because it is implemented directly on top of the underlying hardware.

Hosted Virtual Machine Monitor (VMM):
A hypervisor that runs on top of a host operating system, providing virtualization services to guest operating systems while relying on the host OS for hardware access and resource management.

48. What are the operating system mechanisms that containers make use of?



Ans:

49. What is the **difference between a virtual machine and a container**? Give an example of a scenario where a virtual machine would be better than a container.
Ans:
The difference between a virtual machine (VM) and a container lies primarily in their architectural approach and level of abstraction:

Virtual Machine (VM):

A virtual machine emulates an entire physical computer system, including virtualized hardware such as CPU, memory, storage, and network interfaces.
Each VM runs its own complete operating system (OS) instance, allowing it to host applications and services independently.
VMs provide strong isolation between applications since each VM operates in its own isolated environment.
Hypervisors, such as VMware ESXi or Microsoft Hyper-V, manage and allocate physical resources to VMs.

Container:

Containers encapsulate applications and their dependencies, including libraries and runtime environments, but share the host operating system kernel.
Containers are lightweight, as they do not require a separate OS instance for each container.
Containers provide efficient resource utilization and rapid deployment, making them ideal for microservices architectures and cloud-native applications.
Container runtimes, like Docker or Kubernetes, manage the lifecycle and deployment of containers.
Example scenario where a virtual machine would be better than a container:

Consider a scenario where strong isolation and compatibility are critical requirements:

Scenario: You need to run multiple legacy applications that require different versions of operating systems or have conflicting dependencies on the same physical server.

Solution:

Using virtual machines: You can deploy each legacy application within its own VM, with each VM running its required operating system version. This ensures strong isolation between applications, as they operate within separate OS instances.
Explanation: Virtual machines provide the ability to run multiple instances of different operating systems on the same physical server, allowing each application to have its isolated environment. This setup ensures compatibility for legacy applications that may have specific OS requirements or dependencies that cannot be easily accommodated within a containerized environment.
In this scenario, using containers may not be feasible due to potential conflicts between the dependencies of legacy applications or the need for different OS versions. Virtual machines offer a more robust solution by providing isolated environments for each application, ensuring compatibility and avoiding interference between applications.

50. What is the **difference between an iterative server** and a **concurrent server**?
Ans:
There are several ways to organize servers. In the case of an iterative server, the server itself handles the request and, if necessary, returns a response to the requesting client. A concurrent server does not handle the request itself but passes it to a separate thread or another process, after which it immediately waits for the next incoming request. A multithreaded server is an example of a concurrent server. An alternative implementation of a concurrent server is to fork a new process for each new incoming request. This approach is followed in many Unix systems. The thread or process that handles the request is responsible for returning a response to the requesting client.

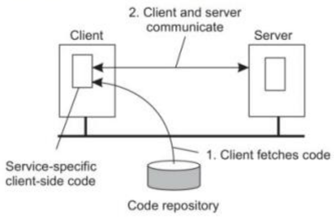
51. What are **two reasons for migrating code**? What is the difference between strong and weak
Ans:

Reasons to migrate code

Load distribution

- Ensuring that servers in a data center are sufficiently loaded (e.g., to prevent waste of energy)
- Minimizing communication by ensuring that computations are close to where the data is (think of mobile computing).

Flexibility: moving code to a client when needed



Avoids pre-installing software and increases dynamic configuration.

Reasons for migrating code

Processes

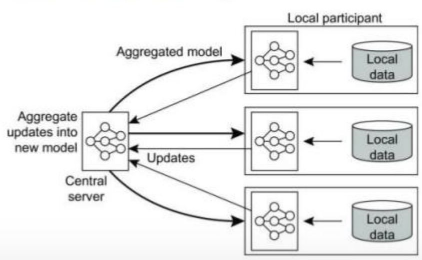
Code migration

Reasons to migrate code

Privacy and security

In many cases, one cannot move data to another location, for whatever reason (often legal ones). Solution: move the code to the data.

Example: federated machine learning



Strong and weak mobility

Object components

- Code segment: contains the actual code
- Data segment: contains the state
- Execution state: contains context of thread executing the object's code

Weak mobility: Move only code and data segment (and reboot execution)

- Relatively simple, especially if code is portable
- Distinguish code shipping (push) from code fetching (pull)

Strong mobility: Move component, including execution state

- Migration: move entire object from one machine to the other
- Cloning: start a clone, and set it in the same execution state.

52. What solution can be used **to achieve code migration in heterogeneous systems**?
Ans:

Migration in heterogeneous systems

Main problem

- The target machine may not be suitable to execute the migrated code
- The definition of process/thread/processor context is highly dependent on local hardware, operating system and runtime system

Only solution: abstract machine implemented on different platforms

- Interpreted languages, effectively having their own VM
- Virtual machine monitors

Observation

As containers are directly dependent on the underlying operating system, their migration in heterogeneous environments is far from trivial, to simply impractical, just as process migration is.

53: Describe the **three techniques that can be used to migrate virtual machines**. Describe the\
Ans:

1. **Pushing memory pages** to the new machine and resending the ones that are later modified during the migration process:

Description: In this technique, the source machine pushes memory pages to the destination machine, and any modified pages during the migration process are resent. This approach ensures that the destination machine has an up-to-date copy of the memory state.
Advantages: Continuous uptime, reduced downtime, efficient bandwidth usage.
Disadvantages: Complexity, potential performance impact, dependency on network stability.

2. S**topping the current virtual machine**; migrate memory, and start the new virtual machine:

Description: This technique involves stopping the current virtual machine, migrating its memory to the destination machine, and then starting a new virtual machine with the migrated memory state on the destination machine.
Advantages: Simplicity, predictable downtime, reduced complexity.
Disadvantages: Downtime, longer migration time, limited flexibility.

3. **Letting the new virtual machine pull in new pages as needed,** that is, let processes start on the new virtual machine immediately and copy memory pages on demand.

Advantages: Minimal downtime, dynamic resource allocation, simplified migration.
Disadvantages: Increased latency, dependency on network performance, potential data inconsistency.

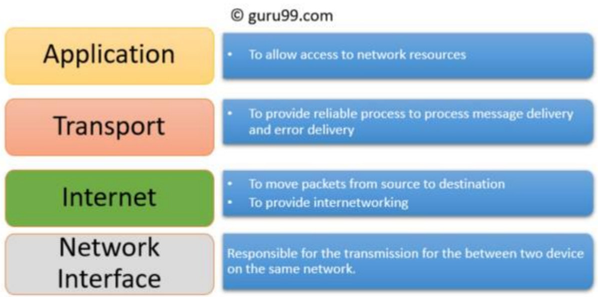
54. Describe all the **layers from the TCP/IP stack**.
Ans:

guru99.com/tcp-ip-model.html

The functionality of the TCP IP model is divided into four layers, and each includes specific protocols.

TCP/IP is a layered server architecture system in which each layer is defined according to a specific function to perform. All these four TCP IP layers work collaboratively to transmit the data from one layer to another.

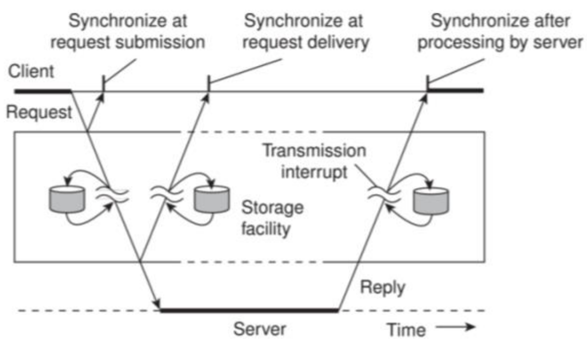
- Application Layer
- Transport Layer
- Internet Layer
- Network Interface



Four Layers of TCP/IP model

55. What is the **difference between transient and persistent communication**? Define and give an example of each.
Ans:

Transient versus persistent



- Transient communication: Comm. server discards message when it cannot be delivered at the next server, or at the receiver.
- Persistent communication: A message is stored at a communication server as long as it takes to deliver it.

transient communication and persistent communication are two distinct approaches to data transmission in distributed systems:

Transient Communication:

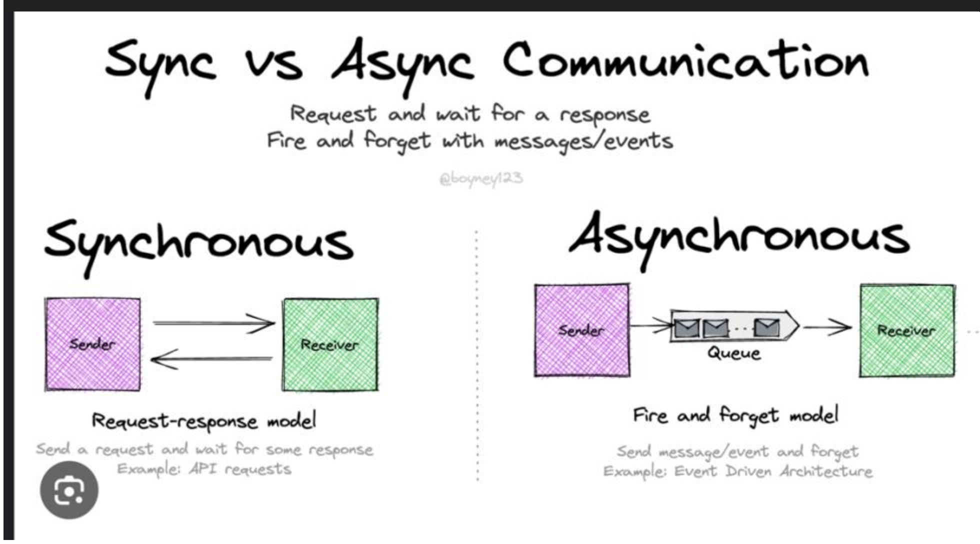
Definition: Transient communication involves sending data between processes or systems without the expectation of retaining the data after the communication session ends. In other words, the data is not stored or persisted beyond the duration of the communication.
Example: One common example of transient communication is real-time messaging in chat applications. When users exchange messages in a chat room, the messages are transmitted to other users in real-time

but are typically not stored permanently on the server. Once the message is delivered and displayed to the recipient, it is no longer maintained by the system.
Persistent Communication:

Definition: Persistent communication refers to the transmission of data with the intention of retaining the data even after the communication session concludes. The data is stored or persisted in some form, typically in databases or files, for future retrieval and reference.
Example: An example of persistent communication is email communication. When users send emails to each other, the messages are not only transmitted to the recipient but also stored in email servers or clients. Users can access and retrieve these messages at any time, even after the initial communication has ended.

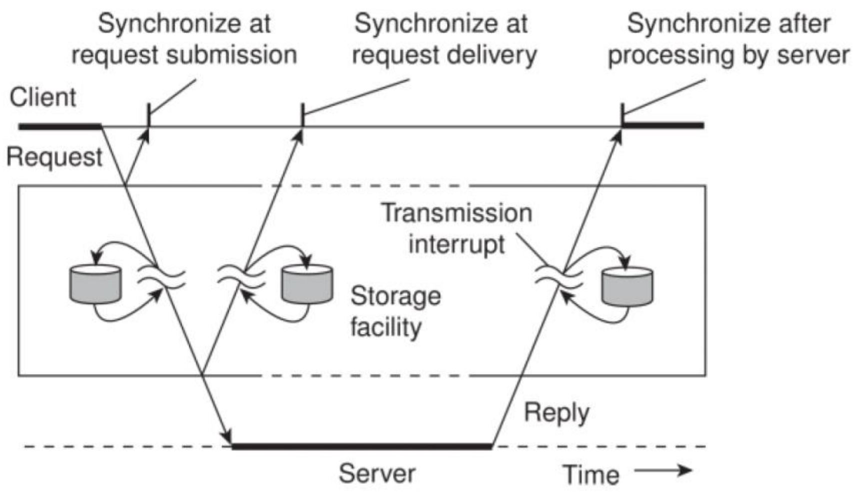
56. What is the **difference between asynchronous and synchronous communication?** Define and give an example of each.
Ans:

Aspect	Synchronous Communication	Asynchronous Communication
Timing	Real-time communication where the sender waits for a response	Communication occurs without strict timing synchronization
Interaction	Sender and receiver operate in lockstep, with each message	Sender and receiver operate independently, not requiring immediate
	being directly tied to a response from the other	interaction
Response Expectation	Immediate response expected after each message sent	No immediate response expected, messages processed independently
Example	Phone call, where each participant expects an immediate response	Email, where messages can be sent and received without immediate



57: What are the **three places** **where synchronization can happen in communication**? Name and define each of them.
Ans:

Places for synchronization



- At request submission
- At request delivery
- After request processing

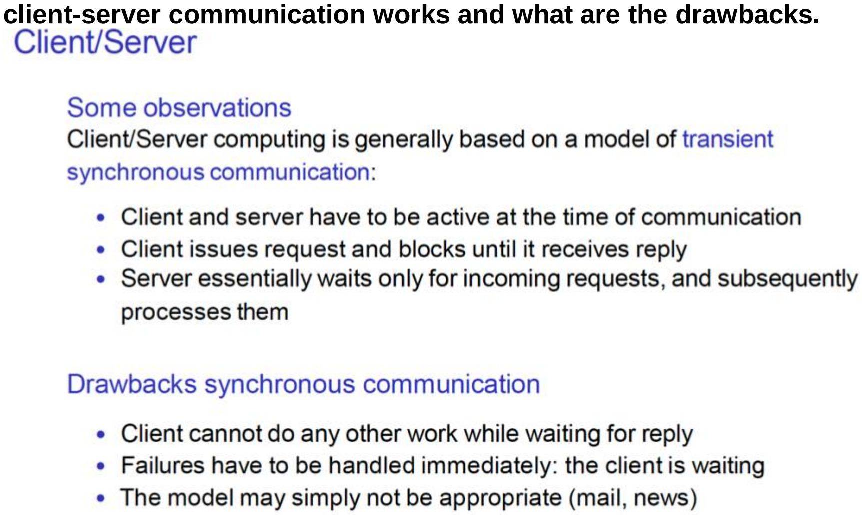
At request submission: This refers to the coordination between the client and the server at the time a request is sent. This synchronization ensures that both parties are using the same communication protocol and that the request is formatted correctly for the server to understand.

At request delivery: This synchronization point occurs when the server acknowledges receipt of the request from the client. This lets the client know that the server has received the request and is working on it.

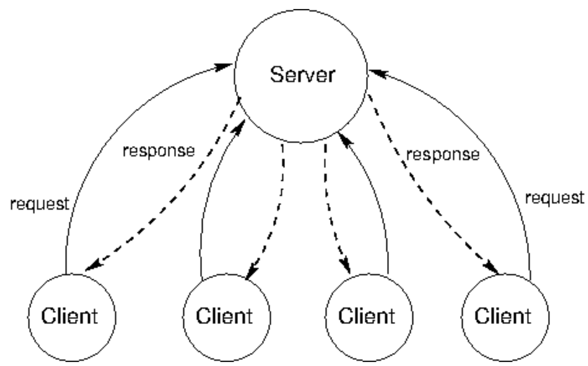
After request processing: This type of synchronization happens after the server has processed the request and is ready to send a reply back to the client. Synchronization at this point ensures that the client is ready to receive the response and that the data is transmitted correctly.

In essence, synchronization helps ensure smooth communication by verifying that both the sender and receiver are on the same page at key points during the exchange.

58: What kind of communication does client-server architectures usually implement? Describe how client-server communication works and what are the drawbacks.



Ans:



1. Client Initiates Request: The client initiates the communication by sending a request to the server. This request includes information about the specific service or data that the client is requesting.

2. Client Waits for Response: Once the request is sent, the client typically waits for a response from the server before proceeding. This means the client is blocked and cannot perform any other tasks until it receives a response.

3. Server Processes Request: Upon receiving the request, the server processes it. This may involve retrieving data from a database, performing calculations, or interacting with other systems.

4. Server Sends Response: Once the server has finished processing the request, it sends a response back to the client. The response may include the requested data, an error message, or any other relevant information.

5. Client Receives Response: The client receives the response from the server and continues execution based on the information it contains.

Drawbacks of Transient Synchronous Communication:

The image also highlights some drawbacks of transient synchronous communication:

Client Blocking: The client is blocked while waiting for a response from the server. This can be inefficient if the server takes a long time to process the request, as the client cannot do any other work during this time.

Immediate Failure Handling: Since the client is waiting for a response, failures need to be handled immediately. This can add complexity to the communication process.

Limited Use Cases: Transient synchronous communication may not be suitable for all applications. For example, it may not be ideal for applications that require real-time updates, such as email or news services.

59: What kind of **communication does message-oriented middleware usually implement?** Describe how **message-oriented middleware communication works**.
Ans:

Message-oriented middleware typically implements asynchronous communication.

In message-oriented middleware (MOM) communication, processes communicate by sending messages to each other. These messages are typically queued in the middleware, allowing for decoupling between sender and receiver.

Here's how message-oriented middleware communication works:

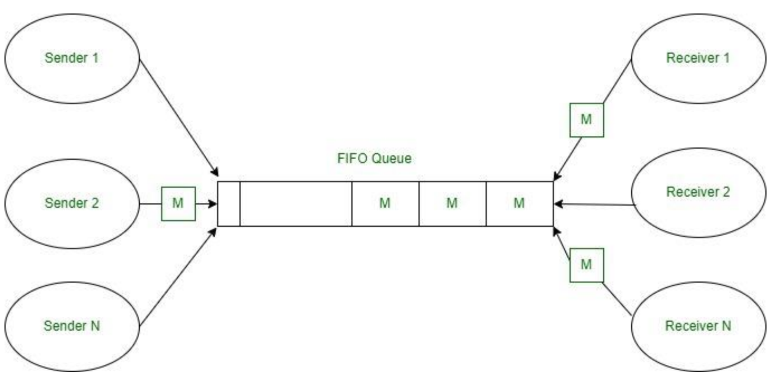
Sending Messages: A process (sender) sends a message to the middleware, specifying the intended recipient and the content of the message.

Queuing Messages: The middleware receives the message and places it in a queue. This queue acts as a buffer, holding the messages until they are consumed by the intended receiver.

Decoupled Communication: Importantly, the sender does not need to wait for an immediate reply from the receiver. This decoupling allows the sender to continue its operation without being blocked by the communication process.

Message Consumption: The receiver process retrieves messages from the queue when it is ready to process them. This can be done asynchronously, allowing the receiver to handle messages at its own pace.

Fault Tolerance: Middleware often provides features for ensuring fault tolerance, such as message persistence, replication, or failover mechanisms. This ensures that messages are not lost even in the event of system failures.



60: What are the **challenges of passing parameters when using Remote Procedure Call (RPC)** for communication? Name and describe them.
Ans:
When using Remote Procedure Call (RPC) for communication, several challenges arise regarding passing parameters between the client and server. Here are the main challenges:

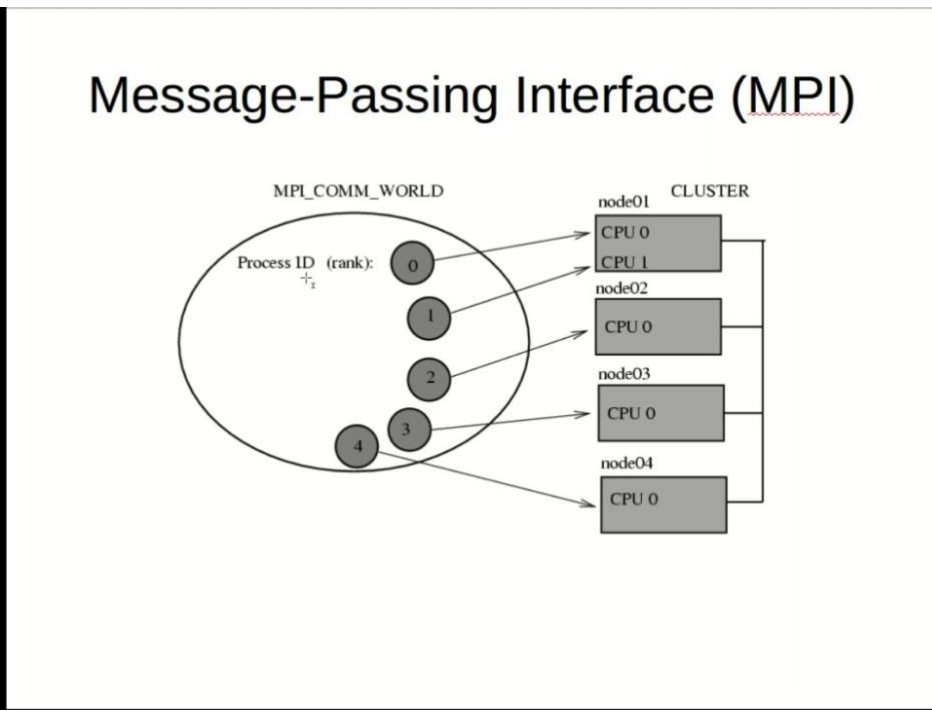
Data Representation Differences: Client and server machines may have different data representations, such as byte ordering. This means that the way data is encoded and decoded on one machine may not be compatible with the other.

Parameter Wrapping: Wrapping a parameter involves transforming its value into a sequence of bytes to be transmitted over the network. Ensuring that this transformation is done correctly and consistently between the client and server is crucial.

Agreement on Encoding: Both the client and server must agree on the same encoding scheme to ensure that parameters are correctly interpreted on both ends. This includes decisions on how basic data values like floats and characters are represented, as well as handling more complex data structures like unions.

Interpreting Messages: Once parameters are received, both the client and server need to properly interpret the messages and transform them into machine-dependent representations that can be understood and processed by the respective systems.

61: Give an example of a solution that **<u>implements transient messaging communication</u>** for distributed systems.
Ans:

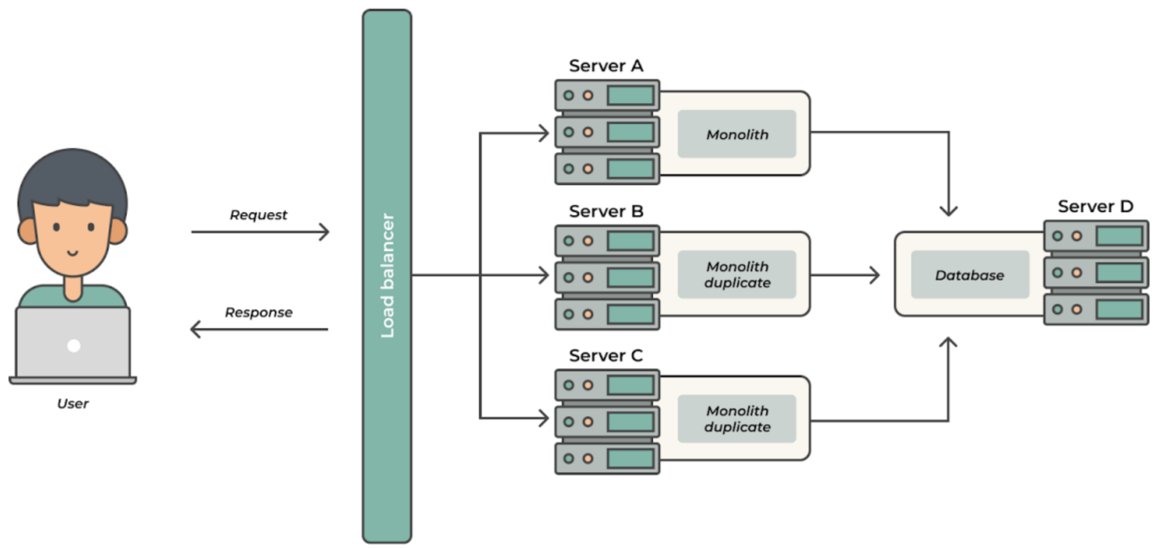


MPI is a standardized and widely used communication protocol and API for parallel computing in distributed memory systems.

In MPI, processes communicate by sending and receiving messages. These messages are transient in nature, meaning that they are exchanged between processes as needed and do not require a persistent connection. MPI provides functions for sending messages from one process to another, as well as functions for receiving messages and handling message queues.

MPI is commonly used in high-performance computing (HPC) environments to facilitate communication and coordination between processes running on different nodes of a cluster. It allows parallel applications to distribute workloads across multiple nodes and exchange data efficiently using transient messaging communication.

62. Give an example of a **solution that implements persistent message-oriented communication** for distributed systems.
Ans:



One example of a solution that implements persistent message-oriented communication for distributed systems is a **message queue** with timeouts. Here's how it works:

Components:

Message Queue: This is a central service responsible for storing messages temporarily.
Sender: The process or service that originates a message.
Receiver: The process or service that needs to receive the message.
Timeout: A pre-defined time limit associated with each message.

Process:

- Sender sends message: The sender sends a message to the message queue.
- Message stored temporarily: The message queue stores the message temporarily.
- Receiver checks queue: The receiver periodically checks the message queue for messages addressed to it.
- Delivery attempt: If a matching message is found within the timeout period, the message queue delivers it to the receiver.
- Timeout and discard: If the timeout expires before the receiver retrieves the message, the message queue discards it.

64: Describe what **UTC stands for and how it functions**.
Ans:
UTC stands for Universal Coordinated Time, and it serves as a global time standard in distributed systems. Here's how it functions:

Definition: UTC is based on the number of transitions per second of the cesium 133 atom, making it extremely precise and reliable.

Clock Synchronization: The goal of UTC is to synchronize the clocks on different machines within a specified bound, known as precision (π). This ensures that the deviation between two clocks is minimal.

Precision and Accuracy: UTC aims to maintain both precision and accuracy. Precision ensures that the deviation between computed clock times on different machines is within π, while accuracy ensures that the clock time remains bound to a value α.

Synchronization Methods: UTC achieves synchronization through both internal and external means. Internal synchronization keeps clocks precise, while external synchronization ensures accuracy by using real-time data from cesium clocks located around the world.

Leap Seconds: UTC introduces leap seconds periodically to compensate for the Earth's changing rotational speed. This ensures that UTC remains aligned with astronomical time and maintains its accuracy.

Distribution: UTC is broadcast globally through shortwave radio and satellite signals. Satellites can provide an accuracy of about ±0.5 milliseconds, ensuring that distributed systems have access to precise and reliable time information.

Overall, UTC plays a crucial role in distributed systems by providing a standardized and globally accepted time reference, enabling effective coordination and synchronization of events across different locations.

Clock synchronization

Precision

The goal is to keep the deviation between two clocks on any two machines within a specified bound, known as the precision π:

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

with $C_p(t)$ the computed clock time of machine p at UTC time t.

Accuracy

In the case of accuracy, we aim to keep the clock bound to a value α:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

Synchronization

- Internal synchronization: keep clocks precise
- External synchronization: keep clocks accurate

65: What is the **difference between precision and accuracy** when talking about **clock synchronization?**

Ans: same diagram as above

In the context of clock synchronization:

Precision: Precision refers to the consistency and repeatability of the timing measurements. It measures how closely multiple measurements taken with the same clock or between different clocks match each other. A highly precise clock will consistently produce the same result when measuring the same interval of time.

Accuracy: Accuracy, on the other hand, refers to how close the measured time is to the true or reference time. It measures the correctness of the timing measurements. An accurate clock will provide measurements that are close to the true time, regardless of whether the measurements are consistent or not.

In essence, precision relates to the consistency of measurements, while accuracy relates to their correctness compared to a reference. A clock can be precise but not accurate if it consistently measures the wrong time, and vice versa.

66. Consider the situation where a client wants to adjust its clock by getting the current time from a UTC timeserver. The client sends the message to the server at timestamp 2. The server receives the message at timestamp 8. The server then sends the reply at timestamp 12. The client receives the reply at timestamp 14. How much is the communication delay and the relative offset?
Ans:

Certainly! Let's calculate the communication delay and the relative offset based on the given timestamps:

- Client sends the message at timestamp **2**.
- Server receives the message at timestamp **8**.
- Server sends the reply at timestamp **12**.
- Client receives the reply at timestamp **14**.

1. **Communication Delay**: The communication delay is the total time taken for the message to travel from the client to the server and back. It can be calculated as follows: Communication delay = (Client receive time - Client send time) - (Server send time - Server receive time)

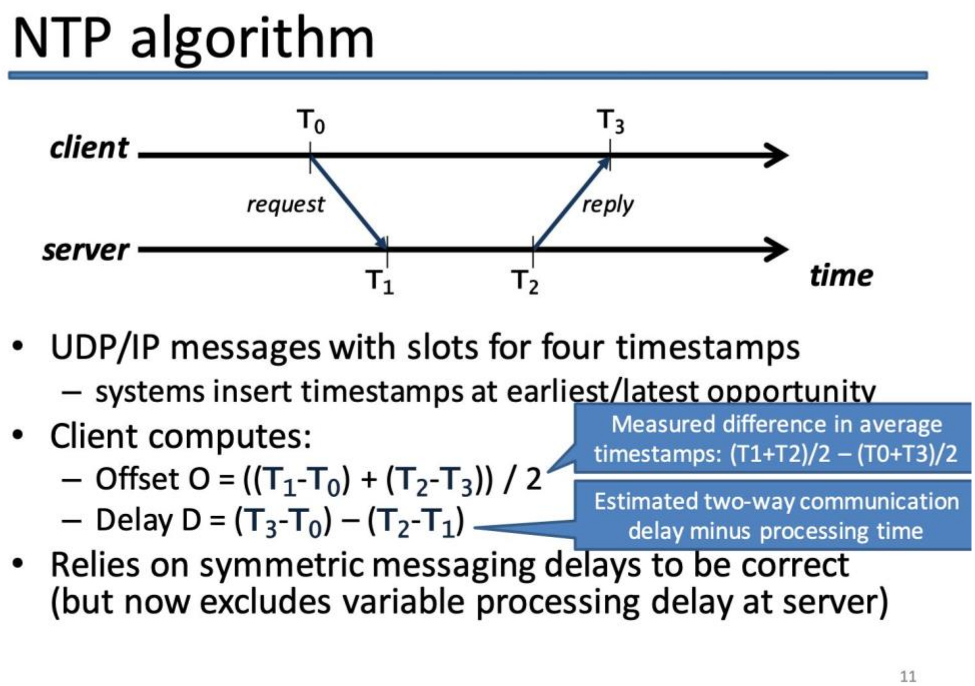
$$\text{Communication delay} = (14 - 2) - (12 - 8) = 8 \text{ units of time}$$

2. **Relative Offset**: The relative offset represents the difference in clock readings between the client and the server. It can be calculated as the average of the time taken by the message to travel from the client to the server and from the server back to the client: Relative offset = ((Server receive time - Client send time) + (Server send time - Client receive time)) / 2

$$\text{Relative offset} = \frac{(8 - 2) + (12 - 14)}{2} = 2.0 \text{ units of time}$$

Therefore, the communication delay is **8 units of time**, and the relative offset is **2.0 units of time**.

If you have any further questions or need additional assistance, feel free to ask!



67: Describe **how Lamport logical clocks work**. If you had the example from Figure 1a with 3 processes exchanging messages, how would the logical clocks be adjusted on each process? Redraw the diagram with the proper adjusted clocks.
Ans:

Lamport's Logical Clocks: A Detailed Explanation

1. Purpose and Background:

- **Leslie Lamport**, a renowned computer scientist, introduced **Lamport's Logical Clocks** to address the challenges of event ordering in distributed systems.
- In distributed systems, processes run independently on different machines, and there is no global clock to synchronize their actions.
- Logical clocks help establish a partial order of events across processes, even when there is no precise notion of real time.

2. Conceptual Overview:

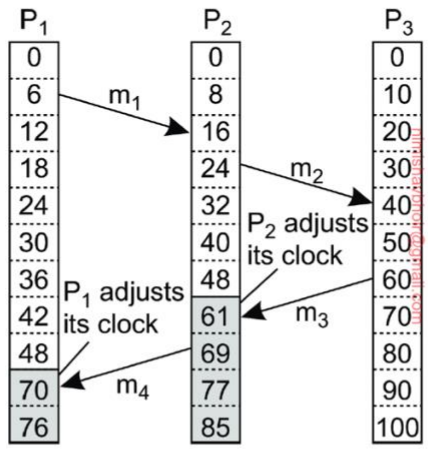
- **Happened-before relation (->)**: If event **"a"** happens before event **"b"**, we denote it as **"a -> b"**.
- **Logical Clock Criteria**:
  - **[C1]**: If **"a"** happens before **"b"** within the same process, then the time of **"a"** will be less than that of **"b"**.
  - **[C2]**: If **"a"** happens before **"b"** across different processes, the clock value of **"a"** will be less than the clock value of **"b"**.

3. Implementation Rules**:**

- **[IR1]**: If **"a"** happens before **"b"** within the same process, then:
  - **Ci(b) = Ci(a) + d**, where **Ci(b)** represents the logical clock value for event **"b"** in process **"i"**.

    - Here, **d** represents the drift factor or communication delay.
- **[IR2]**: When there's an incoming value from another process:
    - **Cj = max(Cj, tm + d)**, where **tm** is the value of **Ci(a)** (from process **"i"**) and **Cj** is the current clock value in process **"j"**.
    - This ensures that the logical clock reflects the order of events even when messages take different amounts of time to propagate.

Redrawn image:



(b)

1. **Initial Situation**:
    - Imagine three processes (let's call them **P1**, **P2**, and **P3**), each with its own clock.
    - These clocks run at different rates, meaning they don't necessarily show the same time.
2. **Sending a Message**:
    - At time **6**, **P1** sends a message (**m1**) to **P2**.
    - The message's arrival time depends on whose clock you trust.
    - When the message reaches **P2**, its clock reads **16**.
3. **Reasoning About Message Travel Time**:
    - If the message carries the starting time (**6**), **P2** concludes that it took **10 ticks** (from 6 to 16) for the message to travel.
    - This seems reasonable.
4. **Inconsistent Values**:
    - Now consider **message m3**:
        - It leaves **P3** at **60** and arrives at **P2** at **56**.
    - Also, **message m4**:
        - It leaves **P2** at **64** and arrives at **P1** at **54**.
    - These values are clearly impossible (traveling back in time!).
5. **Lamport's Solution**:
    - Lamport's idea comes from the **happened-before relation**.
    - Each message carries the **sending time** according to the sender's clock.
    - When a message arrives, if the receiver's clock shows a value **before** the time the message was sent:
        - The receiver **fast-forwards** its clock to be **one more than the sending time**.
    - In our example:
        - **m3** now arrives at **61** (since it left at **60**).

- Similarly, **m4** arrives at **70** (since it left at **64**).

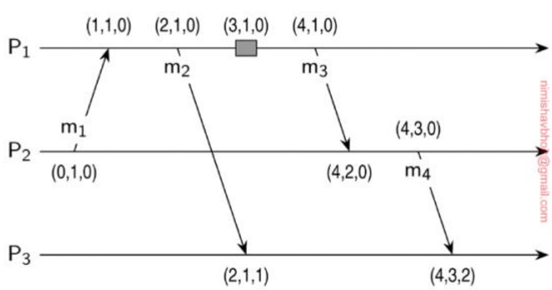
So, Lamport's algorithm ensures that even in a distributed system, events are ordered correctly based on their logical time. 🕰️🌐

68: What is the difference **between logical clocks and vector clocks**. Fill out Figure 1b with the rest of the vector clock timestamps according to the vector clock rules.

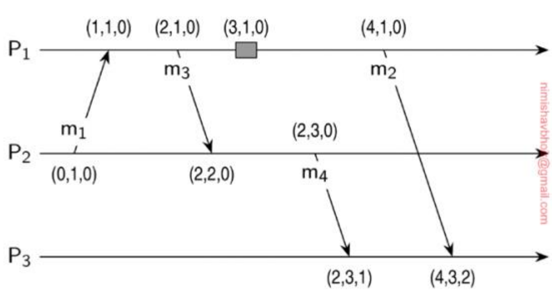
Ans:

Table

Aspect	Logical Clocks	Vector Clocks
Purpose	Capture causality (happened-before) among events in a distributed system.	Detect causality violations and provide a partial ordering of events consistent with causality.
Implementation	Each process maintains its own logical clock. Increment for internal events and message sends. No synchronization between processes.	Each process has a vector (list) of integers (one for each process). Increment for internal events and message sends. Update by taking max of own vector clock and received message's vector clock.
Ordering Capability	Cannot determine causality between arbitrary events.	Can compare causality between any two events.
Compactness	More compact (single integer per process)	Requires a vector (array) of integers for each process.



(a)



(b)

69. What are the two main techniques used to **build mutual exclusion** in distributed systems?Describe the two techniques. Give two examples of algorithms that implement the two mutual exclusions techniques, one for each technique.

Ans:

1. **Token-Based Algorithm**:
   - In token-based algorithms, a **unique token** (also known as the **PRIVILEGE message**) is shared among all the sites.
   - If a site possesses the unique token, it is allowed to enter its **critical section**.
   - This approach uses **sequence numbers** to order requests for the critical section.
2. **Permission-Based Approach**:
   - In permission-based algorithms, sites communicate with each other to determine which sites should execute the critical section.
   - No unique token is required.

Both techniques ensure that only one process or thread can access a shared resource at a time, preventing simultaneous conflicting accesses and maintaining data integrity in distributed systems.

1. **Token-Based Mutual Exclusion**:
   - Examples:
     - **Ricart-Agrawala Algorithm**: A distributed mutual exclusion algorithm based on token exchange. It ensures that only one process can enter the critical section at a time.
     - **Maekawa's Algorithm**: A token-based algorithm that divides processes into disjoint quorums. Each process holds a token for a specific quorum, and only processes within that quorum can request access to the critical section.
2. **Permission-Based Mutual Exclusion**:
   - Examples:
     - **Lamport's Bakery Algorithm**: A non-token-based algorithm that uses individual permissions. Processes take a number and wait until their turn based on the assigned number.
     - **Suzuki-Kasami Algorithm**: Another non-token-based algorithm that uses arbiter permissions (quorum-based). Processes request permission from a quorum of other processes before entering the critical section.

70. What are **two algorithms that can be used to elect a leader** in distributed systems? Describe how the algorithms function.

Ans:

1. **Bully Algorithm**:
   - **Description**:
     - The **bully algorithm** is one of the simplest leader election algorithms.
     - It ensures that only one process becomes the leader among all non-faulty processes.
     - The process with the **highest ID** acts as the coordinator (leader).
   - **Functioning**:
     1. When a process detects that the current coordinator (leader) has failed (via a failure detector):
        - If the detecting process has the highest ID, it **self-elects** itself as the new leader.
        - Otherwise, it initiates an election by sending an **ELECTION message** to processes with higher IDs (including the failed leader).
     2. If the process does not receive an answer within a timeout, it sends a **COORDINATOR message** to all lower-ID processes, announcing itself as the new leader.
     3. The election completes, and all non-faulty processes recognize the elected leader.
   - **Safety**: All non-faulty processes either agree to elect a non-faulty process with the highest attribute (ID) or elect none at all.
   - **Liveness**: The election eventually terminates, and one of the non-faulty processes becomes the leader.

Ring Algorithm (Chang and Roberts):

- **Description**:
  - The **ring algorithm** elects a leader in a ring topology (where processes are connected in a circular manner).
  - Each process communicates only with its neighbors.
- **Functioning**:
  1. When a process detects that the current leader has failed:
     - It initiates an election by sending an **ELECTION message** to its neighbor with the next higher ID.
  2. The message circulates around the ring:
     - Each process compares the received ID with its own.
     - If the received ID is higher, it forwards the message to its neighbor.
     - If the received ID is lower, it responds with a **COORDINATOR message**, announcing itself as the new leader.
  3. The process that receives the **COORDINATOR message** becomes the leader.
- **Safety**: Only one process becomes the leader, and all non-faulty processes recognize it.
- **Liveness**: The election eventually terminates, and a leader is elected.

the **Bully Algorithm**, a distributed system election algorithm used to elect a coordinator process in case the current coordinator fails. Here's how it works:

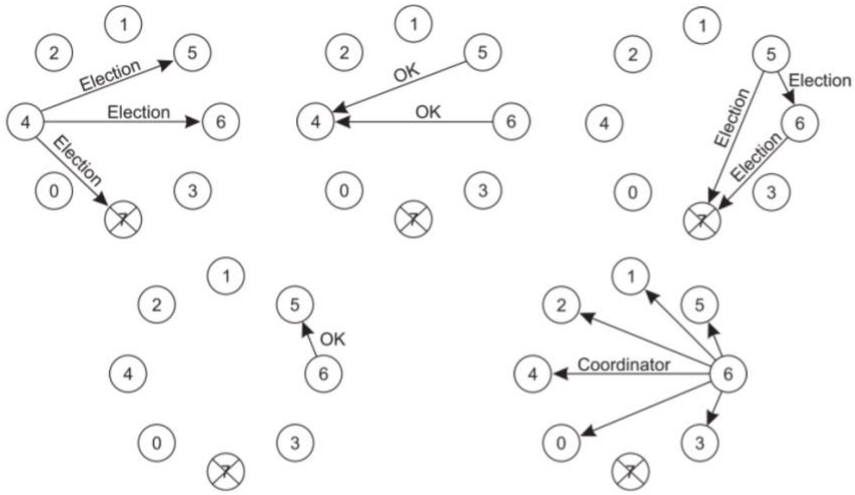
Processes: The vertical lines in the diagram represent processes, denoted by numbers (P1, P2, P3, etc.). Each process has a unique identifier (PID) and can communicate with other processes.

Coordinator: The process in charge of coordinating communication among other processes is called the coordinator. It's denoted by a bold border in the diagram (P5 in the initial state).

Election Trigger: The algorithm initiates an election when a process suspects the coordinator has failed. This can happen if a process doesn't receive a response from the coordinator within a specific time frame.

Election by bullying

The bully election algorithm



The steps involved in the Bully Algorithm are as follows:

Suspicion of Failure: A process (Pi) initiates an election if it suspects the coordinator (P5) has failed. In the diagram, P2 initiates the election.

Election Message: Pi sends an "Election" message to all processes with higher PIDs (processes with a higher number). P2 sends election messages to P3, P4, and P5.

Response: Any process (Pj) with a higher PID that receives an election message acknowledges Pi by sending an "OK" message. Here, P4 doesn't respond because it is suspected to have failed as well.

Coordinator Selection: If Pi doesn't receive an "OK" message from any higher PID process, it declares itself as the coordinator. P2 doesn't receive a response from either P4 or P5, so it becomes the coordinator.

Notification: The new coordinator (Pi) informs all other processes about its new role by sending a "Coordinator" message. The newly elected coordinator, P2, sends coordinator messages to P3 and P0.

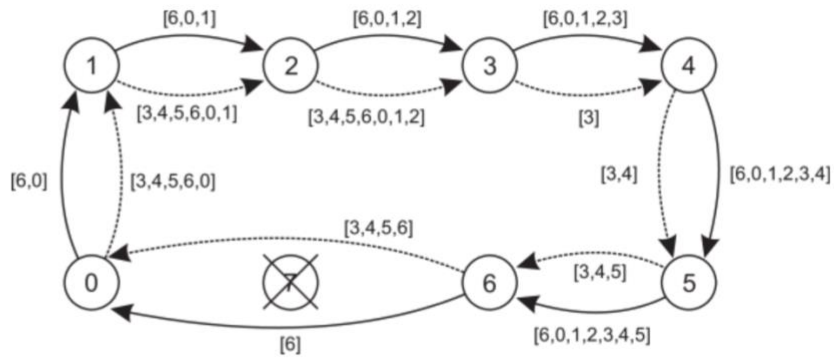
Key Points:

The Bully Algorithm ensures that only one coordinator is elected at a time.

Processes with higher PIDs have more "authority" in the election.

This is a relatively simple algorithm but can generate a significant amount of message traffic, especially in large networks.

Election algorithm using a ring



Imagine an election process where processes are organized in a ring. Unlike some other ring-based algorithms, this one doesn't rely on a token. Each process knows its successor. When a process detects that the coordinator isn't functioning, it constructs an "ELECTION" message containing its own process identifier and sends it to its successor. If the successor is also down, the sender moves to the next member along the ring until it finds a running process. At each step, the sender adds its identifier to the message, effectively nominating itself as a potential coordinator.

Eventually, the message reaches the process that initiated it. This process recognizes the event when it receives an incoming message containing its own identifier. At this point, the message type changes to "COORDINATOR" and circulates again. It informs everyone who the coordinator is (the member with the highest identifier) and who the new ring members are. After one full circulation, the message is removed, and everyone resumes their work.

In Figure 5.21, consider two processes, P3 and P6, simultaneously discovering that the previous coordinator (process P7) has crashed. Both build ELECTION messages and start circulating them independently. Eventually, both messages complete a full round, and both P3 and P6 convert them into COORDINATOR messages with the same members in the same order. These extra messages don't harm the system; at most, they consume a little bandwidth, which is not considered wasteful.

71: How does **a leader get elected in a proof of work election solution?** Describe the basic mechanisms. How does proof of stake differ from proof of work, when performing leader election.

Ans:

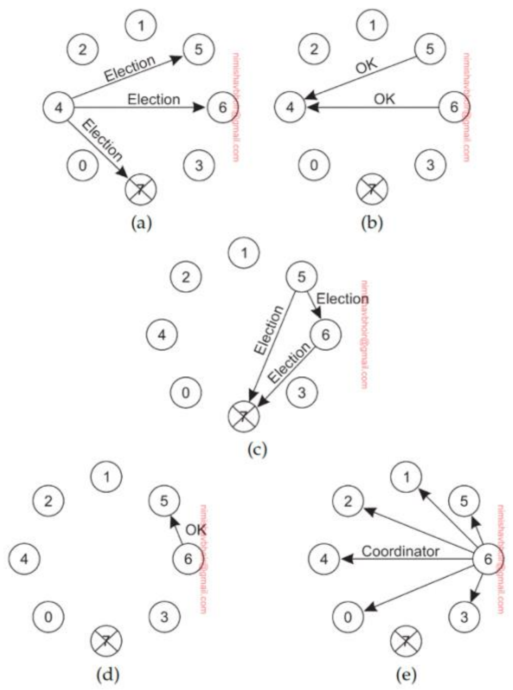
1. **Proof of Work (PoW)** Leader Election:
    - **Purpose**: In PoW-based blockchains (like Bitcoin), the leader (also called the miner) is responsible for proposing new blocks to the network.
    - **Mechanisms**:
        1. **Mining Competition**:
            - Miners compete to solve complex cryptographic puzzles (proof of work).
            - The first miner to find a valid solution gets the right to propose the next block.
            - This miner becomes the temporary leader until the next block is mined.
        2. **Energy-Intensive Process**:
            - PoW requires significant computational power and energy consumption.
            - The cost of mining acts as a deterrent against malicious behavior.

        3. **Randomness and Probability**:
            - The probability of winning the mining competition is proportional to the miner's computational power (hash rate).
            - The more computational power a miner has, the higher the chance of becoming the leader.
2. **Proof of Stake (PoS)** Leader Election:
    - **Purpose**: In PoS-based blockchains (like Ethereum 2.0), the leader (also called the validator) proposes new blocks based on their stake (ownership of coins).
    - **Mechanisms**:
        1. **Staking and Validators**:
            - Validators are chosen based on the number of coins they hold (their stake).
            - Validators take turns proposing blocks.
            - The more coins a validator stakes, the higher the chance of becoming the leader.
        2. **Energy-Efficient**:
            - PoS consumes significantly less energy compared to PoW.
            - Validators are incentivized to act honestly because they have a financial stake in the system.
        3. **Random Selection**:
            - Validators are randomly selected to propose blocks.
            - The randomness is weighted by their stake.
            - Validators take turns being the leader.

Differences:

- **Resource Usage**:
    - PoW requires massive computational power and energy.
    - PoS is energy-efficient and relies on staking.
- **Security**:
    - PoW provides high confidence in security, even in the post-quantum era.
    - PoS security depends on the honesty of validators.
- **Leader Selection**:
    - PoW selects leaders through mining competition.
    - PoS selects leaders based on stake and randomness.

In summary, PoW and PoS differ in their resource usage, security guarantees, and leader selection mechanisms. 🕰️🌐



Programming assignments:

45. In Programming Assignment 2 you had to build a File Retrieval Engine that used multithreading to build a searchable index. Describe in your own words and through pseudocode what strategy you used to build the index using multiple threads.

Ans:
This code implements a file retrieval engine with multithreading for building a searchable index. Here's how the **indexing** process works:

- The indexFiles function takes an input directory and the number of threads as parameters. It creates a vector to hold threads and a mutex for thread safety.
- It iterates through all files in the input directory using std::filesystem::recursive_directory_iterator.
- For each regular file found, it creates a lambda function indexFileFunction that opens the file, reads its content, cleans it, and inserts words along with their file paths into the index.
- Each lambda function is pushed into the threads vector along with the file path.
- After all threads are created, the main thread waits for them to finish using join.
- The insertIndex function of the IndexStore class inserts words and file paths into a B-tree data structure.

Here's a simplified pseudocode representation:

```pseudocode
function indexFiles(input_dir, numberOfThreads):
    start_time = current_time()

    mutex = Mutex()
    threads = []

    for each file in recursive_directory_iterator(input_dir):
        if file is regular:
            indexFileFunction = lambda filePath:
                open file
                read content
                clean content
                for each word in content:
                    lock mutex
                    insert word and filePath into index
                close file

            threads.push_back(thread(indexFileFunction, file.path()))

    for each thread in threads:
        join thread

    end_time = current_time()
    print "Indexing completed:", input_dir, "took", end_time - start_time, "seconds"
```

This approach ensures that multiple files are processed simultaneously, utilizing multiple threads to improve performance. Each thread operates independently on different files, and the mutex ensures thread safety when inserting into the index.

Searching:

The searchFiles function takes a query string as input.

- It parses the query string to extract individual words.
- If the query contains multiple words connected by "AND", it retrieves files containing all those words and sorts them based on the occurrence count.
- It then prints the top 10 search results.
- If the query contains only one word, it retrieves files containing that word.
- If the query format is invalid, it notifies the user.

Here's a simplified pseudocode representation:

pseudocode

```
function searchFiles(query):
    resultsPrinted = false
    words = parseQuery(query)

    if query contains "AND":
        results = lookupFilesContainingAllWords(words)
        sort results based on occurrence count
        print top 10 search results
        resultsPrinted = true
    else if query contains only one word:
        results = lookupFilesContainingWord(word)

    if results not printed:
        print "Invalid query format. Please use 'search word' or 'search word1 AND word2."

    return results

function parseQuery(query):
    return list of words extracted from query

function lookupFilesContainingAllWords(words):
    fileOccurrences = {}

    for each word in words:
        wordResults = btree.lookup(word)

        for each file in wordResults:
            increment occurrence count of file in fileOccurrences

    sort files based on occurrence count
    return top 10 files

function lookupFilesContainingWord(word):
    return files containing the word
```

This approach efficiently retrieves files based on the query, utilizing the B-tree index structure. If the query contains multiple words connected by "AND", it retrieves files containing all those words and prioritizes files with higher occurrence counts. Otherwise, it retrieves files containing the single word provided in the query.

63: In Programming Assignment 3 you had to implement a File Retrieval Engine that followed a client-server architecture. The server used multithreading to support serving multiple clients. at the same time. Describe using your own words and in pseudocode how you implemented. on the server the feature to support serving multiple clients concurrently.

Ans: To support serving multiple clients concurrently, the server utilizes multithreading. Here's a breakdown of how it's implemented:

The server creates a dispatcher thread (dispatcherThread) which continuously listens for incoming client connections.

In the runDispatcher method, the server sets up a socket and binds it to a specific port to listen for incoming connections.

When a client connects, the server accepts the connection and spawns a new worker thread (workerThread) to handle communication with that client.

Each worker thread reads data from the client, processes the request (in this case, a search query), and sends back the results.

While handling the client request, the worker thread also updates the list of connected clients (connectedClients) in a thread-safe manner.

The server keeps track of all worker threads in a vector (workerThreads) to manage them.

If the server receives a shutdown signal, it sets the running flag to false, causing the dispatcher thread to exit the loop and terminate the server.

Here's a simplified pseudocode representation:

```
function initialize():
    start dispatcherThread(runDispatcher)

function runDispatcher():
    create server socket
    bind socket to port
    listen for connections

    while running:
        accept connection from client
        create workerThread(handleClient) for each client

    close server socket

function handleClient(clientSocket):
    read data from client
    process request
    send response to client
    update connectedClients list
    close clientSocket

function shutdown():
    set running flag to false
    join dispatcherThread
```

Indexing, Searching, Listing clients :
Here's the pseudocode for indexing, searching, handling connections, and listing clients based on the provided code snippets:

Indexing:

```
function indexFiles(datasetPath):
    for each file in recursive_directory_iterator(datasetPath):
        if file is regular:
            open file
            read file content
            clean data
            count words
            create IndexMessage with documentPath and wordCounts
            sendIndexMessageToServer(indexMsg)
            close file
    print "Completed indexing in", datasetPath
```

Searching:

```
function searchFiles(query):
    if socketFD == -1:
        print "Error: Connection not established."
        return
    sendSearchRequestToServer(query)
    handleSearchResultsFromServer()
```

Handling Connections:
pseudocode
Copy code

```
function runDispatcher():
    create serverSocket
    bind serverSocket to port
    listen for connections
    while running:
        accept connection from client
        add client address to connectedClients
        create workerThread to handle client communication
    close serverSocket
```

Listing Clients:

```
function getConnectedClients():
    return connectedClients
```

These pseudocode snippets should help you understand the flow and logic of indexing, searching, handling connections, and listing clients in the file retrieval engine. You can further adapt and refine them based on your specific requirements and the actual implementation details.