Distributed System NOTES MOHAMMAD EMRAN

Created	@February 22, 2025 5:12 PM
Created by	EMRAN AHMED EMON

What is a Distributed System?

A **Distributed System** is a network of independent computers that work together to appear as a single system to the user. These computers communicate through a network to coordinate and complete tasks efficiently.

Key Characteristics of Distributed Systems:

1. **Multiple Nodes:** Consists of multiple computers (nodes) working together.
2. **Concurrency:** Tasks can be executed simultaneously.
3. **Scalability:** Can add more nodes to handle increased workloads.
4. **Fault Tolerance:** System continues to work even if some nodes fail.
5. **Transparency:** Users see it as a single system despite multiple machines working together.

Why Do We Use Distributed Systems?

Distributed systems offer several advantages, making them essential for large-scale computing:

1. **Improved Performance:**
    - Parallel processing allows tasks to be executed faster.
    - Reduces bottlenecks compared to a single machine.
2. **Scalability:**
    - New nodes can be added without affecting existing operations.
    - Ideal for applications with growing demands (e.g., cloud computing).
3. **Fault Tolerance & Reliability:**

- If one node fails, others continue functioning.
- Data is replicated across multiple nodes to prevent loss.

4. **Resource Sharing:**
   - Multiple users can share resources like storage, processing power, and data.
5. **Geographical Distribution:**
   - Nodes can be located in different places, useful for global applications.
6. **Cost Efficiency:**
   - Instead of using expensive supercomputers, organizations can use multiple low-cost machines.

Examples of Distributed Systems

1. **Cloud Computing:** AWS, Google Cloud, Microsoft Azure
2. **Search Engines:** Google Search (multiple servers process search queries)
3. **Social Media Platforms:** Facebook, Twitter (handles massive user interactions)
4. **E-commerce Sites:** Amazon, eBay (handles millions of transactions)
5. **Banking Systems:** ATM networks, online banking

Difference Between a Distributed Application and a Distributed System

A **Distributed System** is the underlying infrastructure that enables multiple computers to work together as a single unit, while a **Distributed Application** is a software program that runs on top of a distributed system, utilizing its resources to function efficiently.

Key Differences

Feature	**Distributed Application**	**Distributed System**
Definition	A software application that runs on multiple computers and communicates over a network.	A network of multiple independent computers working together as a single system.
Focus	Solves user-specific problems (business logic, user interaction).	Manages resources, processing, and coordination between

		nodes.
Dependency	Runs on top of a distributed system.	Provides the foundation for distributed applications.
Examples	**Social Media Platforms** (Facebook, Twitter, Instagram), **E-commerce Websites** (Amazon, eBay), **Online Multiplayer Games** (PUBG, Call of Duty)	**Cloud Computing Platforms** (AWS, Google Cloud, Microsoft Azure), **Distributed Databases** (Google Bigtable, Apache Cassandra)**, Blockchain Networks** (Bitcoin, Ethereum)

Real-Life Analogy

- **A Distributed System** is like an **electric grid** that provides power to different cities.
- **A Distributed Application** is like **a factory running on electricity**, using the grid to function efficiently

Conclusion

A **Distributed System** provides the infrastructure, while a **Distributed Application** is a software program that runs on it.

✅ Advantages of Distributed Systems

What are economic and technical reasons for having distributed systems?

1. **Scalability**
    - More computers (nodes) can be added easily to handle increased workloads.
    - Example: Cloud computing platforms like **AWS and Google Cloud** allow businesses to scale on demand.
2. **Fault Tolerance & Reliability**
    - If one node fails, others take over to maintain system operation.
    - Example: In **Google Search**, if one server crashes, another responds to user queries.
3. **Resource Sharing**
    - Multiple users can share computing power, storage, and databases.

- Example: **Distributed file systems like Google Drive** allow multiple users to store and access data.

4. **High Performance & Speed**
    - Tasks are divided across multiple nodes, reducing processing time.
    - Example: **Hadoop (Big Data processing)** distributes tasks across many servers for faster analysis.
5. **Geographical Distribution**
    - Nodes can be placed in different locations, ensuring global accessibility.
    - Example: **CDN (Content Delivery Networks)** like Cloudflare distribute website content globally for faster loading.
6. **Cost-Effective**
    - Uses multiple low-cost machines instead of expensive supercomputers.
    - Example: Companies like Netflix use **distributed architectures** to save infrastructure costs.
7. **Concurrency (Parallel Processing)**
    - Multiple tasks can run simultaneously across nodes.
    - Example: **Online banking systems** handle multiple transactions at the same time

❌ Disadvantages of Distributed Systems

What problems are there in the use and development of distributed systems?

1. **Complexity in Design & Development**
    - Requires specialized knowledge to design, develop, and maintain.
    - Example: Managing **distributed databases like MongoDB** requires handling synchronization issues.
2. **Security Issues**
    - More nodes mean higher chances of cyberattacks and data breaches.

- Example: **Blockchain networks** need advanced cryptography to prevent hacking.

3. **Network Dependency**
   - If the network fails, the system may become inaccessible.
   - Example: **Cloud-based applications like Google Docs** rely entirely on an internet connection.
4. **Data Consistency Challenges**
   - Ensuring data synchronization across multiple nodes is difficult.
   - Example: **Amazon's shopping cart system** must ensure updates reflect correctly across multiple servers.
5. **Difficult Debugging & Troubleshooting**
   - Errors can happen across multiple machines, making debugging complex.
   - Example: **Diagnosing a failed microservice in a distributed application** requires detailed logging.
6. **Higher Maintenance Costs**
   - Requires more resources for monitoring, load balancing, and managing failures.
   - Example: **Facebook's infrastructure team** constantly maintains thousands of servers.

Conclusion

A **distributed system** provides ***scalability, fault tolerance, and efficiency*** but comes with challenges like ***complexity, security risks, and data consistency issues***.

Hardware Architecture:

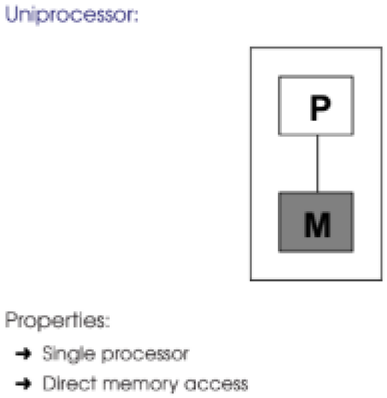
Computer architectures can be classified based on how they process tasks and how many processing units they use. The three main types are:

1. **Uniprocessor System…**
2. **Multiprocessor System…**

3. **Multicomputer System…**

1. Uniprocessor System

A **Uniprocessor system** has a single central processing unit (CPU) that executes all instructions sequentially.



Structure:

Properties:
➜ Single processor
➜ Direct memory access

🔷 **Single CPU → Memory (RAM, Cache, Hard Drive) → Input/Output (I/O) Devices**

📌 Example:

- Traditional desktop computers, laptops, and simple embedded systems.

Advantages:

✅ Simpler design and programming.

✅ Lower cost compared to multiprocessors.

✅ Less power consumption.

Disadvantages:

❌ Limited processing speed due to a single CPU.

❌ Cannot perform true parallel execution.

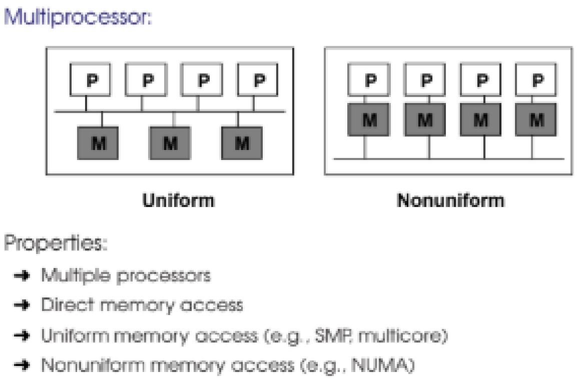
❌ Becomes a bottleneck for high-performance computing.

---

2. Multiprocessor System

A **Multiprocessor system** has two or more processors (CPUs) working together, sharing memory and input/output devices.

Structure:



Multiprocessor:

Properties:

➜ Multiple processors

➜ Direct memory access

➜ Uniform memory access (e.g., SMP, multicore)

➜ Nonuniform memory access (e.g., NUMA)

Properties:
➜ Multiple processors
➜ Direct memory access
➜ Uniform memory access (e.g., SMP, multicore)
➜ Nonuniform memory access (e.g., NUMA)

🔷 **Multiple CPUs → Shared Memory → I/O Devices**

📌 Example:

- Modern servers, high-performance computing (HPC) systems, gaming consoles (PlayStation, Xbox).

Types of Multiprocessors: Uniform vs. Non-Uniform Memory Access (UMA vs. NUMA)

A **multiprocessor system** has multiple CPUs that work together to perform tasks efficiently. Based on how they access memory, multiprocessors are classified into two types:

1. **Uniform Memory Access (UMA) Multiprocessor**
2. **Non-Uniform Memory Access (NUMA) Multiprocessor**

1. Uniform Memory Access (UMA) Multiprocessor

In **UMA architecture**, all processors share a **single, centralized memory** with **equal access time**.

Diagram:

```
+------------+
|  CPU 1   |
+------------+
     |
+------------+
|  CPU 2   |
+------------+
     |
+------------+
|  CPU 3   |
+------------+
     |
+------------------+
| Shared Memory  | ← Same access time for all CPUs
+------------------+
```

Structure:

🔷 **Multiple CPUs → Shared Memory (Single Access Time) → I/O Devices**

Characteristics:

✅ All processors have equal access time to memory.

✅ Memory bandwidth may become a bottleneck.

✅ Used in **Symmetric Multiprocessing (SMP)**.

Example:

- **Traditional multiprocessor servers** (e.g., Intel Xeon-based servers).
- **Supercomputers** (like early IBM multiprocessors).

2. Non-Uniform Memory Access (NUMA) Multiprocessor

In **NUMA architecture**, each processor has its **own local memory**, but it can also access other processors' memory with different (non-uniform) access times.

Diagram:

```
+------------+      +------------+
|  CPU 1    |--⟶ | Local Mem  |
+------------+      +------------+
     |            |
     |  (Remote Access)  |
     |            |
+------------+      +------------+
|  CPU 2    |--⟶ | Local Mem  |
+------------+      +------------+
```

Structure:

🔷 **Multiple CPUs → Local Memory + Shared Memory (with varying access times)**

Characteristics:

✅ Processors access their **own local memory faster** than remote memory.

✅ Used in **Asymmetric Multiprocessing (AMP)** and high-performance computing.

✅ Improves **scalability** by reducing memory bottlenecks.

Example:

- **Modern high-performance servers** (e.g., AMD EPYC, Intel Xeon Scalable processors).
- **Large-scale supercomputers** (Cray, SGI Altix).

Comparison Table: UMA vs. NUMA

Feature	**UMA (Uniform Memory Access)**	**NUMA (Non-Uniform Memory Access)**
Memory Access	Equal access time for all processors	Access time depends on whether memory is local or remote

Scalability	Limited (shared memory bottleneck)	Highly scalable (avoids shared memory congestion)
Speed	Slower as processors increase	Faster due to local memory access
Cost	Cheaper	Expensive
Complexity	Easier to program and manage	Complex memory management
Use Case	Small to medium multiprocessor systems	Large-scale multiprocessor systems and supercomputers

Conclusion

- **UMA is simple** but has a memory bottleneck, making it suitable for small multiprocessor systems.
- **NUMA is more scalable**, providing better performance but at the cost of complexity.

[another Types of Multiprocessors:]

1. **Symmetric Multiprocessing (SMP)** – All CPUs share the same memory and OS.
2. **Asymmetric Multiprocessing (AMP)** – One CPU controls others, used in real-time systems.

Advantages Multiprocessor System:

✅ Faster execution through parallel processing.

✅ Higher system reliability and fault tolerance.

✅ Efficient resource utilization.

Disadvantages Multiprocessor System:

❌ More complex hardware and software design.

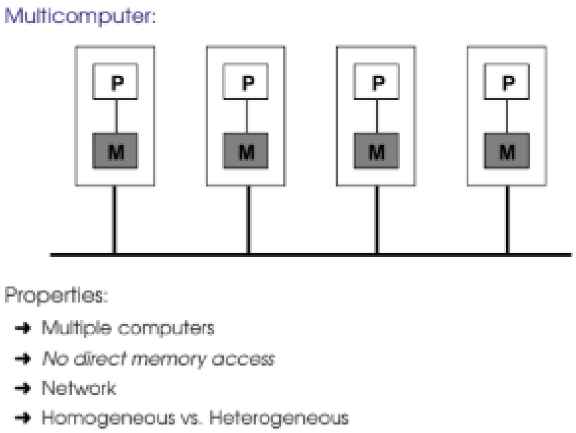
❌ Expensive due to multiple CPUs.

❌ Synchronization issues may arise.

3. Multicomputer System

A **Multicomputer system** consists of multiple computers connected via a network but operating independently. Each computer has its own memory and processors.

Structure:


Multicomputer:

Properties:

➜ Multiple computers
➜ *No direct memory access*
➜ Network
➜ Homogeneous vs. Heterogeneous

Properties:
➜ Multiple computers
➜ No direct memory access
➜ Network
➜ Homogeneous vs. Heterogeneous

🔷 **Multiple Independent Computers → Interconnected via Network (LAN, WAN, etc.)**

📌 Example:

- Cloud computing platforms (AWS, Google Cloud), distributed databases, and blockchain networks.

Types of Multicomputer Systems:

1. **Loosely Coupled Systems** – Computers work independently and communicate when needed (e.g., cloud computing).
2. **Tightly Coupled Systems** – Computers communicate frequently and work as a unit (e.g., parallel computing clusters).

Advantages:

✅ Highly scalable (easily add more computers).

✅ Cost-effective compared to multiprocessors.

✅ Can function even if one node fails (fault tolerance).

Disadvantages:

❌ Higher communication overhead due to networking.

❌ Difficult to program and manage.

❌ Data consistency and synchronization challenges.

Comparison Table

Feature	**Uniprocessor**	**Multiprocessor**	**Multicomputer**
Processing Units	1 CPU	Multiple CPUs (shared memory)	Multiple independent computers
Memory	Single memory unit	Shared memory	Separate memory per computer
Parallelism	No parallelism	True parallel processing	Distributed parallelism
Fault Tolerance	Low	Moderate (if one CPU fails, system continues)	High (if one computer fails, others work)
Scalability	Limited	Limited (depends on memory sharing)	Highly scalable
Example	Personal laptops, desktops	Supercomputers, gaming consoles	Cloud computing, distributed databases

Conclusion

- **Uniprocessors** are simple but slow.
- **Multiprocessors** improve speed using shared memory but are complex.
- **Multicomputer** are highly scalable and used in large distributed systems.

Software Architecture:

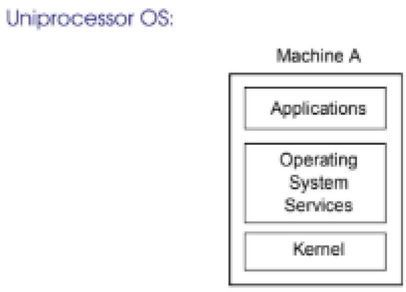
Software Architecture in Distributed Systems

Software architecture in distributed systems defines how an operating system (OS) manages resources, processes, and communication between different computing nodes. There are five main types of OS architectures in distributed systems:

1. **Uniprocessor OS**
2. **Multiprocessor OS**
3. **Network OS**
4. **Distributed OS**
5. **Middleware**

1. Uniprocessor OS

A **Uniprocessor OS** runs on a **single processor (CPU)** and manages system resources like memory, I/O, and processes. It follows a traditional architecture where all components are managed by a single OS.



Structure:

🔷 **Single CPU → Single OS → Processes & Resources Managed by OS**

Example:

- Windows, Linux, macOS on personal computers.

Advantages:

✅ Simple and easy to manage.

✅ Well-established and widely used.

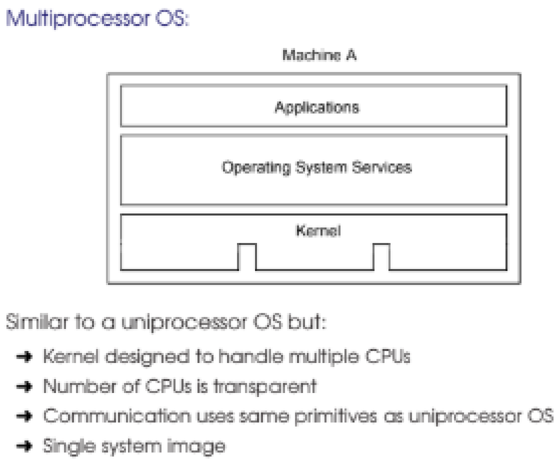
Disadvantages:

❌ Cannot handle parallel processing.

❌ Performance is limited to a single CPU.

2. Multiprocessor OS

A **Multiprocessor OS** manages multiple CPUs that share memory and I/O devices. It allows parallel processing by efficiently scheduling tasks across multiple processors.



Multiprocessor OS:

Similar to a uniprocessor OS but:

- ➜ Kernel designed to handle multiple CPUs
- ➜ Number of CPUs is transparent
- ➜ Communication uses same primitives as uniprocessor OS
- ➜ Single system image

Structure:

🔷 **Multiple CPUs → Shared Memory → Single OS Controls All CPUs**

Example:

- **Linux SMP (Symmetric Multiprocessing)**
- **Windows Server running on multiple cores**

Advantages:

✅ Faster execution with parallel processing.

✅ Efficient resource sharing.

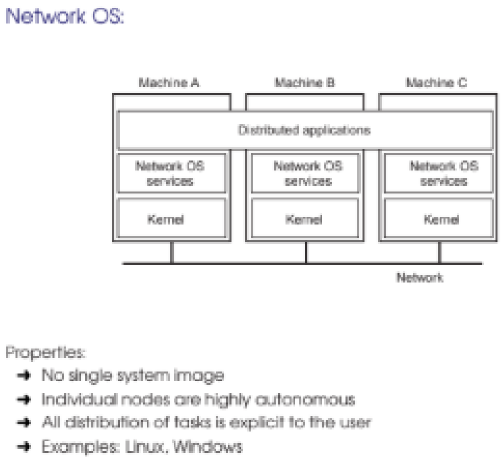
Disadvantages:

❌ Complex OS design due to shared memory management.

❌ Requires synchronization mechanisms to prevent conflicts.

3. Network OS (NOS)

A **Network OS** allows multiple computers (nodes) to communicate and share resources over a network but maintains their own OS. It does **not** give the illusion of a single system.



Network OS:

Properties:

- ➜ No single system image
- ➜ Individual nodes are highly autonomous
- ➜ All distribution of tasks is explicit to the user
- ➜ Examples: Linux, Windows

Structure:

🔷 **Independent Computers → Connected via Network → Each Runs Its Own OS**

Example:

- Windows Server, Linux, UNIX, Novell NetWare.

Advantages:

✅ Centralized resource management (files, printers, databases).

✅ High independence; each computer operates separately.

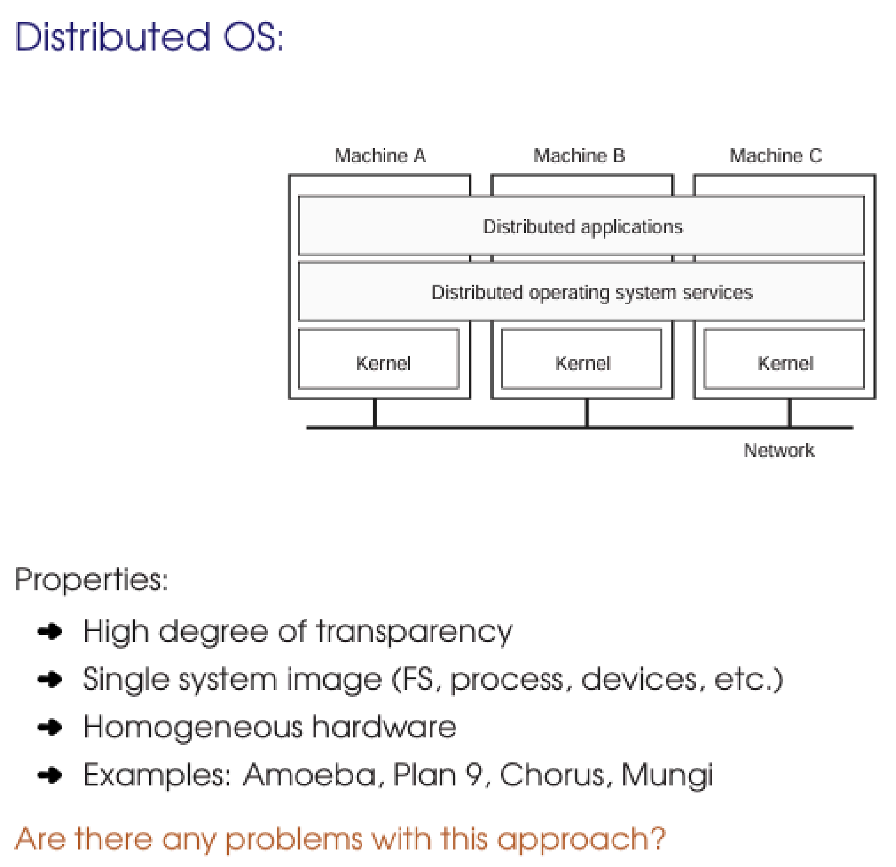
Disadvantages:

❌ Requires manual user management across multiple machines.

❌ Communication between nodes is slower than in a Distributed OS.

4. Distributed OS

A **Distributed OS** makes multiple computers appear as a **single system** by managing resources transparently across multiple nodes.



Distributed OS:

Properties:

- High degree of transparency
- Single system image (FS, process, devices, etc.)
- Homogeneous hardware
- Examples: Amoeba, Plan 9, Chorus, Mungi

Are there any problems with this approach?

Structure:

🔷 **Multiple Computers → Single OS Manages All Nodes as One System**

Example:

- **Google's Cloud Infrastructure**
- **Amoeba, Mach, Sprite (research OSs)**

Advantages:

✅ Seamless resource sharing with transparency.

✅ High fault tolerance and scalability.

Disadvantages:

❌ Complex to implement and maintain.

❌ Requires advanced algorithms for synchronization and resource allocation.

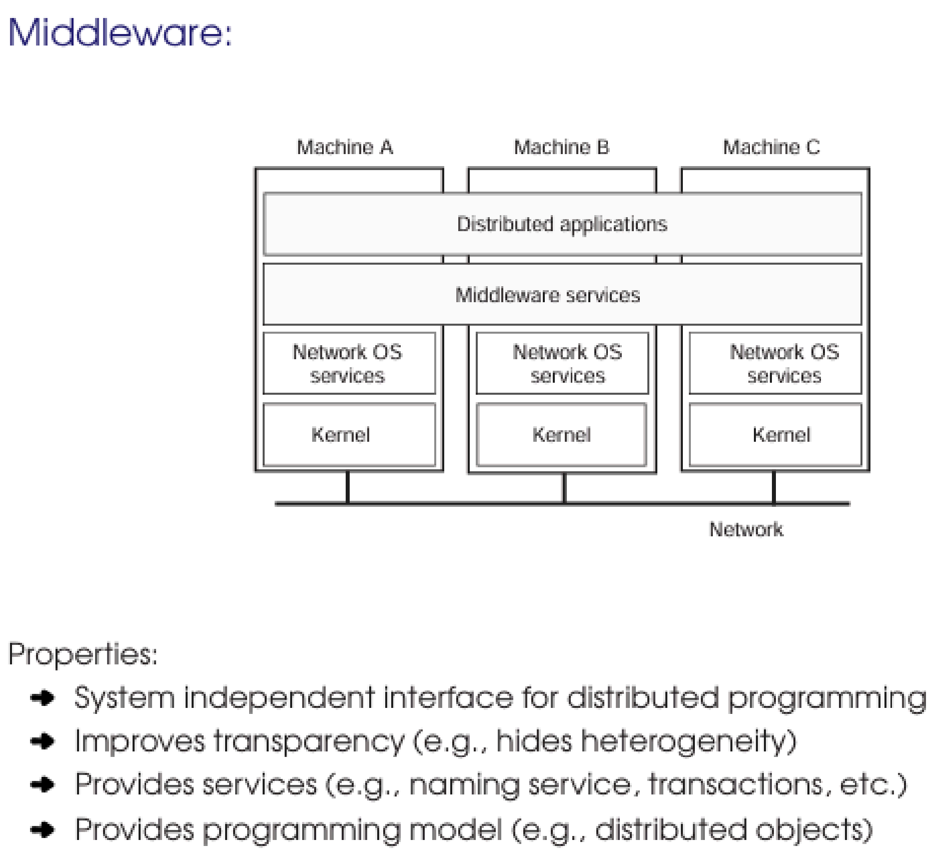
Are there any problems with this Distributed OS approach?

Problems with Distributed OS:

1. **Transparency**: Hard to hide access and location differences.
2. **Synchronization**: Difficult to coordinate processes across nodes.
3. **Fault Tolerance**: Managing failures and maintaining data consistency.
4. **Scalability**: Performance drops with more nodes due to communication overhead.
5. **Security**: Ensuring secure communication and data access.
6. **Resource Management**: Efficiently sharing resources across nodes.
7. **Complex Maintenance**: Difficult to debug and update the system.
8. **Communication Overhead**: Latency and bandwidth limitations affect performance.

5. Middleware

Middleware is a software layer that helps different distributed components (software or hardware) communicate seamlessly. It abstracts the complexities of networking, allowing applications to interact across different platforms.


Middleware:

Properties:

- System independent interface for distributed programming
- Improves transparency (e.g., hides heterogeneity)
- Provides services (e.g., naming service, transactions, etc.)
- Provides programming model (e.g., distributed objects)

Structure:

🔷 **Applications → Middleware → Distributed System Components**

Example:

- **CORBA, RPC (Remote Procedure Call), Message Queues (Kafka, RabbitMQ), Web APIs.**

Advantages:

✅ Simplifies communication in distributed systems.

✅ Enhances portability across different OSs.

Disadvantages:

❌ Adds an extra layer, increasing overhead.

❌ Requires careful security management.

Why Middleware is Better than Other Architectures?

Middleware is **better than traditional OS approaches** (Uniprocessor, Multiprocessor, Network OS, and Distributed OS) because it **simplifies communication, integration, and scalability** in distributed systems.

Why It's Better?

Middleware acts as a **bridge** between applications and the OS, making distributed systems **more flexible, scalable, and easier to manage** than other architectures.

Comparison Table

Type	Structure	Example	Key Features
Uniprocessor OS	Single CPU, Single OS	Windows, Linux, macOS	Simple, limited to one processor
Multiprocessor OS	Multiple CPUs, Shared Memory	Linux SMP, Windows Server	Supports parallel processing
Network OS (NOS)	Multiple independent computers, each with its own OS	Windows Server, UNIX	Resource sharing, no single system view

Distributed OS	Multiple computers act as a single system	Google Cloud, Amoeba	Seamless resource sharing, scalability
Middleware	Software layer enabling communication	CORBA, RPC, Kafka	Bridges different platforms and services

Diagrams for Better Understanding

1. Uniprocessor OS

```
+-------------------+
| Uniprocessor OS  |
+-------------------+
       |
+-------------------+
| Single CPU      |
+-------------------+
       |
+-------------------+
| Memory & I/O    |
+-------------------+
```

2. Multiprocessor OS

```

   +---------------------------+
   | Multiprocessor OS        |
   +--------------------------
        |
+------------+------------+
|    CPU 1  |  CPU 2  | ← Multiple CPUs
+------------+------------+
        |
   +-------------------+
   | Shared Memory   |
   +-------------------+
```

3. Network OS (NOS)

```
+-------------------+     +-------------------+
| OS on Node 1  | -——→ | OS on Node 2  |
+-------------------+     +-------------------+
         |
    (Network Connection)
         |
    +-------------------+
    | Shared Resources |
    +-------------------+
```

4. Distributed OS

```
+--------------------------+
|     Distributed OS      |
+--------------------------+
   |        |
+--------+   +--------+
| Node 1 |   | Node 2 |  ← Computers appear as one system
+--------+   +--------+
   |        |
+------------------------+
| Shared Resources (Memory, Files) |
+------------------------+
```

5. Middleware in a Distributed System

```
    +---------------------------+
    | Application Layer       |
    +---------------------------+
           |
    +---------------------------+
    | Middleware (RPC, Kafka) |
    +---------------------------+
           |
    +--------------------------+
    | Distributed System      |
    +--------------------------+
```

Conclusion

- **Uniprocessor & Multiprocessor OS** are used in traditional systems.
- **Network OS & Distributed OS** help in resource sharing and scalability.
- **Middleware** bridges gaps between different platforms.

System Architecture

System architectures in distributed systems define how components interact, communicate, and coordinate to achieve scalability, fault tolerance, and efficiency. Here are the main types:

1. Client-Server Architecture

📌 **How it works?**

- A **central server** provides services to multiple **clients**.
- Clients send requests, and the server responds.

📌 **Example:**

- Websites like **Facebook, YouTube** (your browser is the client, the website is the server).

📌 **Simple Diagram:**

```
arduino
CopyEdit
Client → Server → Database
```

📌 **Pros:**

✅ Easy to manage

✅ Centralized control

📌 **Cons:**

❌ If the server crashes, everything stops.

❌ Can get slow if too many clients request data.

---

2. Vertical Distribution (Multi-tier Architecture)

📌 **How it works?**

- The system is divided into **layers (tiers)** for better performance.
- Common **3-tier architecture**:
    1. **Presentation Layer (Frontend - UI)**
    2. **Business Logic (Backend - Processing)**
    3. **Database (Storage - Data handling)**

📌 **Example:**

- **Netflix** (UI for users → Backend logic → Database for movies).

📌 **Simple Diagram:**

```
pgsql
CopyEdit
User → Frontend → Backend → Database
```

📌 **Pros:**

✅ More secure, scalable

✅ Easier to update individual layers

📌 **Cons:**

❌ More complex than Client-Server

❌ Slightly slower due to multiple layers

---

3. Horizontal Distribution (Load Balancing Architecture)

📌 **How it works?**

- The system **spreads tasks** across multiple servers to **avoid overload**.

📌 **Example:**

- **Google Search, Amazon, Cloud Services** (multiple servers handle traffic).

📌 **Simple Diagram:**

```
pgsql
CopyEdit
Load Balancer → Server 1
       → Server 2
       → Server 3
```

📌 **Pros:**

✅ Handles high traffic smoothly

✅ More fault-tolerant

📌 **Cons:**

❌ Needs a good load-balancing system

❌ Slightly harder to manage

---

4. Peer-to-Peer (P2P) Architecture

📌 **How it works?**

- **No central server**, all devices (peers) **communicate directly**.

📌 **Example:**

- **Torrent downloads, Bitcoin, Skype calls** (users share data with each other).

📌 **Simple Diagram:**

```
mathematica
CopyEdit
 Peer A ↔ Peer B ↔ Peer C
   ↕    ↕
 Peer D ↔ Peer E
```

📌 **Pros:**

✅ No single point of failure

✅ Faster for data sharing

📌 **Cons:**

❌ Security risks

❌ Hard to manage data consistency

5. Hybrid Architecture (Combination of Multiple Models)

📌 **How it works?**

- Uses a mix of **Client-Server + P2P** for efficiency.

📌 **Example:**

- **Microsoft Teams, Zoom** (server for storage, P2P for real-time calls).

📌 **Simple Diagram:**

```
arduino
CopyEdit
  Clients ↔ Server ↔ P2P Nodes
```

📌 **Pros:**

✅ Flexible and powerful

✅ Best of both worlds

📌 **Cons:**

❌ More complex to implement

🚀 Final Summary:

Architecture	How it Works?	Example	Pros	Cons
Client-Server	One central server serves clients	Websites, Banking Apps	Easy to manage	If the server fails, everything stops
Multi-tier (Vertical)	Data is processed in layers (UI → Logic → DB)	Netflix, Web Apps	Secure, scalable	Slower due to layers
Horizontal Distribution	Load balancing across multiple	Google, Amazon	Handles traffic well	Needs good load balancing

	servers			
Peer-to-Peer (P2P)	Devices communicate directly	Torrents, Bitcoin	No server needed	Security issues
Hybrid	Mix of multiple architectures	Microsoft Teams	Powerful and flexible	More complex

Distributed Shared Memory (DSM)

What is Distributed Shared Memory in DS?

- **DSM is a system that allows multiple computers (nodes) to share memory as**
  if they were using a single shared memory space.
- It helps different programs communicate without needing direct messaging.
- Even though each node has its own physical memory, DSM creates a virtual shared memory across all nodes.

Distributed Shared Memory (1).pdf

attachment:ffcc9a30-2271-4b27-914e-0abe63dcc251:Distributed_Shared_Memory_(1).pdf

🧠 What is Distributed Shared Memory (DSM)?

✅ Explanation:

- DSM allows multiple computers (nodes) in a network to **share memory as if it were one big common memory**.
- Each computer has its **own physical memory**, but DSM creates a **virtual shared space**.

- Programs can **read/write to shared variables** without explicitly using message passing.

🧠 Analogy:

Imagine a **group of people working on a whiteboard together**. Even though they are in different rooms, they **see and write to the same virtual board** through cameras and screens. This is similar to DSM!

⚙️ How DSM Works

- **Each node offers read/write access** to part of memory.
- A **consistency protocol** ensures that all copies of the data are updated correctly.
- Data is **moved between nodes** as required when one node needs to access data held by another.

📚 Types of Distributed Shared Memory

1. Based on Granularity

- Refers to the **size of memory blocks shared**.

Type	Description
Fine-grained	Shares **small blocks** (e.g., variables). More control but more overhead.
Coarse-grained	Shares **larger blocks** (e.g., pages/segments). Less overhead but risk of false sharing.

2. Based on Implementation

Type	Description
Hardware-based DSM	Built using special hardware (e.g., NUMA systems). Fast but costly.
Software-based DSM	Built using software techniques (e.g., TreadMarks). More flexible and cheaper.

3. Based on Data Distribution and Access

Type	Description
Page-based DSM	Memory divided into pages; pages move between nodes.
Object-based DSM	Memory shared as objects (used in object-oriented systems).
Variable-based DSM	Individual variables are shared directly.

4. Based on Replication and Update Policies

Policy	Description
Write-invalidate	When one node writes, others' copies are **invalidated**.
Write-update	All nodes are **updated** when one node writes.

✅ Advantages of Distributed Shared Memory

1. **Simpler Programming** – Feels like programming on a single memory space. Easier than message passing.
2. **Easier Portability** – DSM programs can run on various systems with little modification.
3. **Locality of Data** – Related data can be fetched together, improving performance.
4. **On-Demand Data Movement** – Moves only needed data, avoiding unnecessary transfers.
5. **Larger Memory Space** – Combines memory of all nodes into a single large virtual memory.
6. **Better Performance** – Reduces communication delays and uses memory efficiently.
7. **Flexible Communication** – Processes can join/leave without breaking the system.
8. **Simplified Process Migration** – Moving a process to another node is easier since address spaces are shared.

❌ Disadvantages of Distributed Shared Memory

1. **Slower Access** – Accessing remote memory is slower than local memory.
2. **Consistency Problems** – Must ensure that all nodes see the latest data.

3. **Less Efficient Message Passing** – Not as fast as explicit message-passing models.
4. **Data Redundancy** – Multiple copies of the same data can create inconsistencies.
5. **Lower Performance** – CPU and cache can be underutilized due to delays in memory access.

🛠️ Issues in Design & Implementation of DSM

1. Granularity

- Decides the **block size** for shared memory.
- Fine granularity gives better control but increases communication.
- Coarse granularity reduces messaging but may share unneeded data (false sharing).

---

2. Structure of Shared Memory Space

- How the shared memory is **organized** (e.g., flat, object-oriented, variable-based).
- Depends on the **type of application** (scientific, object-oriented, etc.).

---

3. Memory Coherence & Synchronization

- Ensures that **all nodes see consistent values** of shared data.
- Requires synchronization tools like **locks, semaphores, barriers** to avoid conflicts.

---

4. Data Location and Access

- DSM must **track and locate data blocks** across the network efficiently.
- Ensures fast data access and memory consistency.

---

5. Replacement Strategy

- When a node runs out of space, it must **remove old data** to make space for new data.

- Needs a smart policy to decide **which data to evict**.

6. Thrashing

- If two nodes frequently access the same block, it moves rapidly between them.
- This **degrades performance**.
- DSM must handle this with **delay or access coordination** strategies.

7. Heterogeneity

- DSM is easier in **homogeneous systems**.
- In **heterogeneous environments** (different OS, architecture), DSM must handle compatibility issues.

✅ Summary Table

Aspect	**Key Idea**
DSM Goal	Shared memory abstraction across multiple machines
Key Types	Granularity-based, Implementation-based, Access-based
Key Benefits	Simplicity, performance, portability
Main Challenges	Consistency, thrashing, performance

📘 Flashcards: Distributed Shared Memory (DSM)

🃏 **Flashcard 1Q:** What is Distributed Shared Memory (DSM)?

A: DSM allows multiple computers to share memory virtually as if they were using a single memory space, enabling inter-process communication without message passing.

🃏 **Flashcard 2Q:** How does DSM work?

A: Each node offers memory services, and a consistency protocol ensures all nodes see updated data. Data moves between nodes as needed.

🃏 **Flashcard 3Q:** What is fine-grained DSM?

A: A DSM that shares small memory blocks like individual variables or small objects.

🃏 **Flashcard 4Q:** What is coarse-grained DSM?

A: A DSM that shares larger memory blocks such as pages or segments.

🃏 **Flashcard 5Q:** What is hardware-based DSM?

A: A DSM that uses special hardware (e.g., NUMA systems) to provide shared memory.

🃏 **Flashcard 6Q:** What is software-based DSM?

A: A DSM implemented using software techniques, typically with page-based memory management.

🃏 **Flashcard 7Q:** What are the types of DSM based on access?

A: Page-based DSM, Object-based DSM, and Variable-based DSM.

🃏 **Flashcard 8Q:** What is write-invalidate DSM?

A: When a node writes to shared memory, other copies are invalidated to maintain consistency.

🃏 **Flashcard 9Q:** What is write-update DSM?

A: When a node writes to shared memory, the update is propagated to all other copies.

🃏 **Flashcard 10Q:** Name one advantage of DSM.

A: Simplifies programming with a shared address space abstraction.

🃏 **Flashcard 11Q:** How does DSM improve performance?

A: It reduces communication costs and enables data locality through on-demand transfer.

🃏 **Flashcard 12Q:** What is thrashing in DSM?

A: Repeated movement of data between nodes due to frequent write access by multiple nodes.

🃏 **Flashcard 13Q:** What is granularity in DSM?

A: The size of memory blocks shared and transferred across the network.

🃏 **Flashcard 14Q:** What is meant by memory coherence in DSM?

A: Ensuring that all nodes have a consistent view of shared data.

🃏 **Flashcard 15Q:** What are some design issues in DSM?

A: Granularity, structure of shared space, coherence, data location, replacement, thrashing, heterogeneity.

🃏 **Flashcard 16Q:** What is the advantage of larger memory space in DSM?

A: DSM combines memory from all nodes to form a larger virtual memory, reducing paging.

🃏 **Flashcard 17Q:** What is the disadvantage of consistency in DSM?

A: Programmers must ensure updates are visible across all nodes, which can be complex.

🃏 **Flashcard 18Q:** What is the effect of heterogeneity in DSM design?

A: DSM must handle different architectures and OSs, increasing complexity.

(Additional)

🔑 Issues in Designing & Implementing DSM

1. **Data Location and Access** 📍🔍

- **Problem**: How to **locate and access** shared data that might be on a different machine?
- **Solution**: Use **mapping mechanisms** (like page tables) to keep track of data locations.

2. **Granularity of Sharing** 📦
    - **Problem**: What **size of data** (word, block, page) should be shared?
    - **Trade-off**:
        - Fine-grained → more flexibility, but more overhead
        - Coarse-grained → less overhead, but possible false sharing
3. **Memory Consistency Model** 📘
    - **Problem**: How to ensure all processes **see memory updates in the correct order**?
    - **Common Models**:
        - Sequential Consistency
        - Weak Consistency
        - Release Consistency
4. **Replication and Consistency** 📄🔁
    - **Problem**: If data is **replicated**, how do we keep all copies **synchronized**?
    - **Solution**: Use **update or invalidate protocols** for consistency.
5. **Synchronization** ⛓️
    - **Problem**: How to coordinate access to shared memory to **prevent race conditions**?
    - **Solution**: Use **locks, barriers, or semaphores** across the network.
6. **Page Replacement and Thrashing** 🔄
    - **Problem**: Frequent page transfers can cause **performance degradation (thrashing)**.
    - **Solution**: Use **smart replacement algorithms** and reduce unnecessary page transfers.

7. **Fault Tolerance** ⚠️
    - **Problem**: What if a node crashes?
    - **Solution**: Need **recovery mechanisms** to avoid data loss.
8. **Scalability** 📈
    - **Problem**: Can the system handle more nodes efficiently?
    - **Solution**: Use **distributed algorithms** and limit central bottlenecks.

✅ Summary Table:

Issue	Challenge	Example Solution
Data Location	Finding where data is	Page tables, directories
Granularity	Size of shared units	Page-level sharing
Consistency Model	Update visibility & order	Sequential / Release consistency
Replication	Synchronizing copies	Invalidate/update protocols
Synchronization	Coordinating access	Locks, barriers
Page Thrashing	Too much data movement	Smart replacement, locality awareness
Fault Tolerance	Node failures	Redundancy, backups
Scalability	System expansion	Decentralized control

✅ Advantages of Distributed Shared Memory (DSM)

1. **Transparency**
    - **Programmers can write code as if there's one shared memory** system, even though memory is physically distributed.
    - Makes programming easier and more intuitive.
2. **Cost-effective**
    - Uses **commodity hardware** (regular networked computers) instead of expensive shared-memory multiprocessors.
3. **Scalability**
    - DSM can **scale across many machines**, allowing for large systems with more memory and processing power.
4. **Fault Isolation**

   - Since memory is distributed, a **failure in one node doesn't crash the entire system** (better reliability).

5. **Better Resource Utilization**
   - Can **dynamically share memory resources** across machines based on demand.
6. **Supports Parallel Programming Models**
   - Allows **parallel applications** to be written using shared memory models without needing to worry about message passing.

❌ Disadvantages of Distributed Shared Memory (DSM)

1. **Performance Overhead**
   - **Network latency** when accessing remote memory can be high.
   - DSM often suffers from **slower performance** compared to real shared memory.
2. **Complex Consistency Management**
   - Maintaining **memory consistency** (same value of data across all nodes) is **complex and costly**.
3. **False Sharing**
   - If unrelated variables are in the same memory page, updates to one can **cause unnecessary invalidations**, reducing efficiency.
4. **Synchronization Complexity**
   - Coordination (locks, semaphores) over a network is **slower and more error-prone**.
5. **Debugging Difficulty**
   - Distributed systems are **harder to debug and test** due to concurrency and remote memory issues.
6. **Page Thrashing**
   - Frequent memory page transfers can **degrade performance significantly**, especially if processes access memory in unpredictable ways.

(optional!)

7. Accessibility: The data access is slow in DSM as compared to non-distributed.
8. Consistency: When programming is done in DSM systems, programmers need to maintain consistency.
9. Message Passing: DSM uses asynchronous message passing and is not efficient as per other message passing implementation.
10. Data Redundancy: DSM allows simultaneous access to data, consistency and data redundancy is a common disadvantage.
11. Lower Performance: CPU gets slowed down, even cache memory does not aid the situation.

📝 Summary Table:

Advantages	**Disadvantages**
Easy to program (memory abstraction)	High network latency for remote access
Cost-effective (uses normal hardware)	Complex consistency protocols
Scalable to many nodes	False sharing can reduce performance
Isolates faults (better reliability)	Difficult to synchronize across network
Efficient resource usage	Debugging and testing is harder
Supports shared memory model	Risk of page thrashing

📊 Diagram-Based Study Guide: Distributed Shared Memory (DSM)

🧠 1. Basic DSM Architecture

```
+----------+     +----------+     +----------+
| Node A  | <-----> | Node B  | <-----> | Node C  |
| (RAM)   |         | (RAM)   |         | (RAM)   |
+----------+     +----------+     +----------+
      \____________________________/
           Virtual Shared Memory Space
```

Explanation: All nodes have local memory, but DSM allows them to act like one big shared memory.

📦 2. Granularity: Fine vs Coarse

```
[Fine-Grained DSM]         [Coarse-Grained DSM]
+-------+ +-------+ +-------+    +-------------------+
| Var1 | | Var2 | | Var3 |     |    Page A       |
+-------+ +-------+ +-------+    +-------------------+
```

Explanation:

- Fine-grained: Shares variables or small data units.
- Coarse-grained: Shares entire memory pages (less messaging, but can cause false sharing).

🔄 3. Write-Invalidate vs Write-Update

```
[Write-Invalidate]           [Write-Update]
 Node A writes → Others'     Node A writes → Others'
 copies invalidated.          copies updated.
```

```
[Before]        [After]          [Before]        [After]
A: x=1          A: x=5 (valid)    A: x=1          A: x=5
B: x=1          B: x=? (invalid)  B: x=1          B: x=5
```

Explanation:

- Write-invalidate: Others' data is marked invalid.
- Write-update: New value is broadcast to others.

🧩 4. Types of DSM Based on Access

```
[Page-Based DSM]     [Object-Based DSM]     [Variable-Based DSM]
+----------+          +-----------+          +-----------+
| Page 1 |           | Object 1 |          | Var x    |
+----------+          +-----------+          +-----------+
```

Explanation: Different levels of abstraction depending on the application needs.

⚠️ 5. Thrashing in DSM

```
Node A <——> Node B
     Writing same block
  ⟳ (Data keeps bouncing)
```

Explanation: When two nodes repeatedly write to the same data, it causes performance loss.

📐 6. DSM Design Issues

```
+----------------------------+
|    DSM Design Issues       |
+----------------------------+
| Granularity                |
| Structure of Shared Space  |
| Memory Coherence           |
| Data Location & Access     |
| Replacement Strategy       |
| Thrashing                  |
| Heterogeneity              |
+----------------------------+
```

Explanation: These are the core challenges engineers face when designing DSM systems.

🧠 7. Advantages vs Disadvantages

```
[Advantages]              [Disadvantages]
+ Simpler abstraction       - Slower access time
+ Larger memory space       - Consistency issues
+ Better performance        - Data redundancy
+ Portability               - Message passing overhead
```

Tip: Use these in short questions or 5-mark exam answers.

📌 Summary:

- DSM = virtual shared memory across nodes.
- Types vary by granularity, abstraction, and consistency methods.
- Diagrams simplify understanding of memory behavior in distributed environments.

Perfect for visual learners! Use these to quickly revise before exams.

Election Algorithm:

attachment:553759da-bdd6-488b-83af-8e0537118860:DS-ElectionProcess.pdf

DS-ElectionProcess.pdf

🗳️ What is the Election Algorithm?

An **Election Algorithm** is used in distributed systems to **select one process (node) as a coordinator or leader** from among a group of processes. This coordinator can manage tasks like mutual exclusion, synchronization, etc.

🔑 Why is it needed?

- In a distributed system, there's **no central control**.
- When the **coordinator fails or crashes**, the system needs to **elect a new one**.
- The election algorithm ensures **agreement among processes** to select a single leader.

📋 Key Features:

- All processes are assumed to have **unique IDs**.
- Processes must detect when the coordinator fails.
- The algorithm must ensure that **only one process becomes the leader**.

⭐ Common Election Algorithms:

1. Bully Algorithm

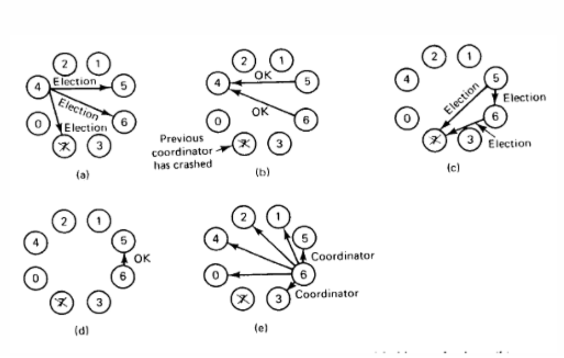
👑 Concept:

- The process with the **highest ID becomes the coordinator**.
- If a process notices that the coordinator is not responding, it **starts an election**.

📌 Steps:

1. The detecting process sends **ELECTION messages** to all higher-ID processes.
2. If no one responds, it becomes the leader.
3. If any higher-ID process responds, it **takes over the election**.
4. Eventually, the process with the **highest ID wins and announces itself** using a **COORDINATOR message**.

Example:



- The group consists of 8 processes.
- Previously process 7 was the coordinator, but it has just crashed.
- Process 4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely Processes 5 and 6 both respond

with OK.

- Upon getting the first of these responses, 4 knows that its job is over.
- Both 5 and 6 hold elections, each one only sending messages to those processes
  higher than itself.
- Process 6 tells 5 that it will take over. 6 knows that 7 is dead and that it is the winner. 6 announces this by sending a COORDINATOR message to all running
  processes.
- When 4 gets this message, it can now continue with the operation it was trying to
  do when it discovered that 7 was dead, but using 6 as the coordinator this time.
- If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.

✅ Pros:

- Simple, deterministic.
- Guarantees the **highest-ID process wins**.

❌ Cons:

- Generates **many messages** in large systems.
- Assumes reliable message delivery and failure detection.

---

2. Ring Algorithm

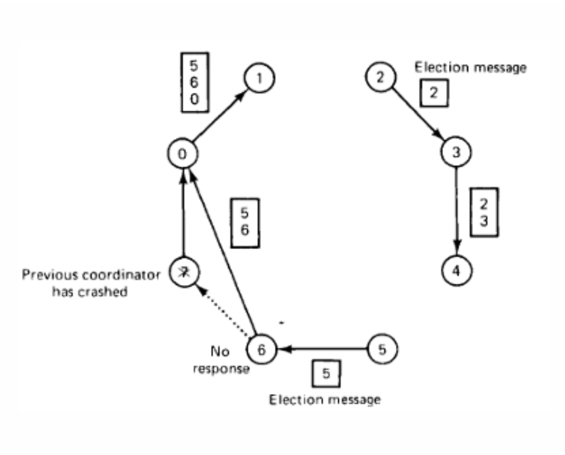
🔁 Concept:

- Processes are organized in a **logical ring** (not physical).
- Election token is passed around the ring.

📌 Steps:

1. Any process noticing the coordinator has failed **initiates an election** by sending a message around the ring.
2. Each process appends its own ID and forwards the message.
3. When the message returns to the initiator, it chooses the **highest ID** as the new coordinator.
4. A new **coordinator message** is sent around to announce the leader.

Example



1. **Crash Detection**:
    - Node **0** detects that the **previous coordinator** (with highest ID) has crashed.
    - It initiates an election.
2. **Sending Election Messages**:
    - Node 0 sends an **election message** with its own ID `{0}` to the next node in the ring (clockwise).
    - Node 1 receives `{0}`, adds its own ID → `{0,1}`, and forwards it.
    - Node 2 adds ID → `{0,1,2}`, and so on.
3. **Building the Candidate List**:
    - Each node adds its own ID and passes it along.
    - When the message reaches node 6, it contains all active node IDs `{0,1,2,3,4,5,6}`.
4. **Coordinator Selection**:

    - Node 6 notices the crashed coordinator is not responding (marked with ✱).
    - Among the list, it finds the **highest ID = 6**, so it becomes the **new coordinator**.

5. **Announcement**:
    - Node 6 announces itself as the new coordinator to all nodes in the ring.

✅ Pros:

- **Fewer messages** than Bully in large systems.
- Works well in **structured systems**.

❌ Cons:

- **Slower** if many processes are involved.
- Depends on a correct ring structure.

🔒 Important Properties of Election Algorithms

Property	Description
Uniqueness	Only one leader is chosen
Fairness	Every process has a fair chance to be leader
Termination	Algorithm completes in a **finite time**
Fault Tolerance	Can handle process or message failures

🧠 Real-Life Analogy

Think of a **classroom election**:

- Students (processes) have IDs (roll numbers).
- If the class captain (coordinator) is absent (fails), students must **elect a new captain**.
- Either the **highest roll number** takes over (Bully), or they **pass a note around** and vote (Ring).

📌 Short Summary for Exam:

Aspect	Bully Algorithm	Ring Algorithm
Structure	All processes know each other	Logical ring structure
Winner	Highest ID process	Highest ID after circulating ring
Messages	Many (especially with many nodes)	Fewer (passes once around ring)
Speed	Faster in small systems	Efficient in large, structured rings
Use Case	Unstructured systems	Structured rings

more Real-Life Analogy

🗳️ 1. General Election Algorithm – Real-Life Analogy

🧠 Analogy: Class Monitor Election

- Imagine a classroom where the teacher has left.
- The students (processes) must decide who will be the **new monitor (coordinator)**.
- They agree that the **most responsible or senior student (highest ID)** should be the monitor.
- They follow a structured process to **elect one student**.

Key Idea: All students coordinate to choose one leader to represent them in the teacher's absence.

💪 2. Bully Algorithm – Real-Life Analogy

🧠 Analogy: Bossy Students in a Classroom

- Each student has a **rank or roll number**.
- If the **monitor is absent**, a student with a **lower roll number** (e.g., 3) notices it.
- This student informs others with **higher roll numbers** (like 4, 5, 6).
- If no higher-ranked student replies, student 3 declares itself the new monitor.

- But if a **more senior student replies** (say, student 5), that student takes over the process and **"bullies" the others into stopping**.
- Finally, the **highest roll number present** becomes the monitor.

Key Idea: The **most powerful (highest ID) process takes control** and becomes the leader. It bullies others by asserting superiority.

🔁 3. Ring Algorithm – Real-Life Analogy

🧠 Analogy: Passing a Note in a Circle

- Students are seated in a **circle** (logical ring).
- When the current monitor leaves, one student realizes it and writes their name on a **note** and passes it clockwise.
- Each student adds their name and forwards the note.
- When the note returns to the starting student, they look at the list of names.
- The student with the **highest roll number becomes the new monitor**.
- This student then **announces the result** to everyone in the circle.

Key Idea: Every student gets a **chance to participate**. Leadership is decided by **collective information**, not by fighting.

Distributed File System:

What is Distributed File System in DS?

Distributed file systems (DFS) are designed to let you store and access files across many servers as if they were on one local machine—handling issues like **scalability**, **fault tolerance**, and **performance** along the way. (SPoF)

attachment:553d2c3a-630c-4b61-986e-bf071bfa88f4:Chap_12_notes_-_DFS.pdf

Chap 12 notes - DFS.pdf

📂 What is a Distributed File System (DFS)?

✅ Basic Idea:

- A **DFS** lets multiple clients **share files over a network**, accessing them as if stored locally.
- Files are stored on **file servers**, and clients access them using a **file service interface**.

🧠 Real-Life Analogy:

Think of **Google Drive** or **Dropbox**: you access your files from anywhere, but they're actually stored on servers elsewhere. It feels local but is distributed.

🔍 Key DFS Characteristics and Challenges

🔶 1. Transparency

Making distribution invisible to users.

Type	Meaning
Location	Users don't know (or care) where files are stored.
Migration	Files can move to another server without user impact.
Replication	Multiple copies exist, but users see one file.
Concurrency	Multiple users can access the same file safely.

🔶 2. Flexibility

- Easily **add or replace file servers**.
- Support **different file system types** (e.g., Unix, Windows).

🔶 3. Reliability

Concern	Solution

Consistency	Ensure changes are reflected across replicas.
Security	Authenticate users; enforce file access permissions.
Fault Tolerance	Continue service even if a server crashes.

🔶 4. Performance

- Use **multiple servers** to spread the load.
- Needed especially when **handling large data volumes**.

🔶 5. Scalability

- Avoid central points like a single lock or name server.
- Handle more users, files, and wider geographic locations.

👀 Client Perspective: File Access Models

🔶 Upload/Download Model

- Client **downloads the file**, edits it locally, and uploads it back.
- Works **even without a constant connection**.

✅ Pro: Less network traffic

❌ Con: Conflict if two users edit the file at once.

🔶 Remote Access Model

- **All operations happen on the server** (like executing commands).

✅ Pro: Better conflict control

❌ Con: Needs continuous server connection.

📖 File Access Semantics

🔶 1. Unix Semantics

- Read after write always shows updated data.
- Only possible with **one server and no caching** → not practical in DFS.

🔶 2. Session Semantics

- Changes visible only after the file is closed.

🧠 Analogy: Like **editing a document offline**, then syncing changes when you reconnect.

🔶 3. Immutable Files

- Files can't be changed—**only replaced**.

🧠 Analogy: Think of **Git commits**—once added, they're not changed, only new versions are created.

🔶 4. Atomic Transactions

- File changes are treated like **database transactions**: all-or-nothing.

✅ Pro: No interference

❌ Con: Expensive to implement

🖥️ Server Perspective

🔍 Usage Pattern Insights:

Based on studies like Satyanarayanan's Unix system analysis:

- Most files are small
- Reads > Writes
- Access is sequential
- File sharing is rare

🔧 Suggests: Use **Upload/Download model** with **client-side caching**.

🗄️ Stateful vs Stateless File Servers

Feature	Stateless Server	Stateful Server
Session Info	❌ Doesn't store session info	✅ Remembers file status, locks, etc.
Recovery	✅ Easy crash recovery	❌ Must restore state
Performance	❌ More data sent each time	✅ Less data sent, can optimize reads

Features	❌ No file locking	✅ Supports locking, read-ahead, etc.

🔁 Replication in DFS

Used for **performance, availability, fault tolerance**.

🧩 Replication Methods:

Type	Description
Explicit	Client manually copies files to servers.
Lazy	Server replicates in background after updates.
Group Replication	Writes go to all replicas at once for **strong consistency**.

🧠 Analogy: Like having your notes saved in **multiple locations**—some synced live, others when convenient.

💾 Caching in DFS

Reduces network traffic and improves performance.

🔶 Server-Side Caching:

- Uses OS-level caching to reduce disk access.

🔶 Client-Side Caching:

Type	Description
On-Disk	Saves file copies to local disk.
In-Memory	Caches files in RAM (faster).

🧩 Cache Consistency Approaches:

Method	Description
Delayed Write	Updates sent later in background.
Write-on-Close	Changes sent only when file is closed.
Write-on-Close with Delay	Waits a bit before writing → avoids useless writes.

🧪 Example Distributed File Systems

📁 NFS (Network File System)

- ✅ Stateless
- ✅ Server-side + client-side caching
- ❌ No replication
- ❌ Weak security (basic UID/GID)
- Used in **Unix/Linux systems**
- Mounts remote files via `/etc/export` rules

📁 AFS (Andrew File System)

- ✅ Scalable & secure
- ✅ Whole-file caching
- ✅ Server invalidates stale client caches using **callbacks**
- ✅ ACL-based access, uses **Kerberos for auth**
- Good for **campus-wide networks**
- Example: `/afs/cs.cmu.edu`

📁 Coda

- Successor to AFS, supports **disconnected operations**
- ✅ Uses **Venus daemon** and **hoarding**
- ✅ Write logs saved during disconnect (CML), replayed later
- ✅ Automatic + manual conflict resolution
- ✅ Volumes are basic units of replication
- Used in **mobile and offline environments**

🧠 Analogy: Like **Google Docs offline mode**—you work offline and sync when reconnected.

📁 GFS (Google File System)

- Built for **large-scale data + high failure environments**
- ✅ Optimized for **large sequential reads** and **concurrent appends**

- ✅ Designed for **failure as normal**, not exception
- Highly fault-tolerant and scalable

📝 Summary Chart

Feature	NFS	AFS	Coda	GFS
Stateless	✅	❌	❌	❌
Caching	✅	✅	✅	✅
Replication	❌	Partial	✅	✅
Disconnected Op	❌	❌	✅	❌
Use Case	Unix	Campus	Mobile	Web-scale data

📘 Flashcards: Distributed File Systems (DFS)

🃏 **Flashcard 1Q:** What is a Distributed File System (DFS)?

A: A DFS allows multiple clients to access and share files over a network as if the files were stored locally.

🃏 **Flashcard 2Q:** What does file service interface do in a DFS?

A: It hides the details of file storage and communication, providing uniform access methods to files.

🃏 **Flashcard 3Q:** What is location transparency in DFS?

A: Users can access files without knowing their physical storage locations.

🃏 **Flashcard 4Q:** What is replication transparency?

A: Users see only one file even if multiple copies exist across the system.

🃏 **Flashcard 5Q:** What are the goals of DFS?

A: Transparency, flexibility, reliability, performance, and scalability.

🃏 **Flashcard 6Q:** What is the upload/download file access model?

A: The client downloads the file, works on it locally, and uploads it back when done.

🃏 **Flashcard 7Q:** What is the remote access file model?

A: All file operations happen on the server and require constant connectivity.

🃏 **Flashcard 8Q:** What is Unix semantics in DFS?

A: Ensures immediate visibility of writes to all users, which is hard to achieve in distributed systems.

🃏 **Flashcard 9Q:** What is session semantics?

A: Changes to a file are visible only after the file is closed.

🃏 **Flashcard 10Q:** What are immutable files?

A: Files that are not modified after creation; only new versions are created.

🃏 **Flashcard 11Q:** What are atomic transactions in DFS?

A: File operations are grouped and executed as all-or-nothing units to maintain consistency.

🃏 **Flashcard 12Q:** What did Satyanarayanan's study conclude about file usage?

A: Most files are small, reads dominate, access is sequential, and file sharing is rare.

🃏 **Flashcard 13Q:** What is a stateless file server?

A: A server that does not maintain any session or state information about client interactions.

🃏 **Flashcard 14Q:** What is a stateful file server?

A: A server that maintains information like open files, locks, and session states.

🃏 **Flashcard 15Q:** What is explicit replication?

A: Clients manually replicate files to different servers.

🃏 **Flashcard 16Q:** What is lazy replication?

A: The server replicates updated files to others in the background.

🃏 **Flashcard 17Q:** What is group replication?

A: Updates are sent to all replicas simultaneously to maintain strong consistency.

🃏 **Flashcard 18Q:** What is caching in DFS?

A: Storing copies of frequently accessed files in memory or disk to reduce load and improve performance.

🃏 **Flashcard 19Q:** What is server-side caching?

A: The file server uses its local memory/disk to cache files and reduce disk reads.

🃏 **Flashcard 20Q:** What is client-side caching?

A: Clients store copies of files locally to improve access times and reduce network usage.

🃏 **Flashcard 21Q:** What is delayed write caching?

A: Client updates are held and sent later in the background to reduce network traffic.

🃏 **Flashcard 22Q:** What is write-on-close caching?

A: Changes are sent to the server only when the file is closed by the client.

🃏 **Flashcard 23Q:** What is write-on-close with delay?

A: Waits for a short time after file closure to batch updates and reduce writes.

🃏 **Flashcard 24Q:** What is NFS?

A: Network File System; a stateless, Unix-based DFS using RPC and weak security.

🃏 **Flashcard 25Q:** What is AFS?

A: Andrew File System; uses whole-file caching and callback-based invalidation with strong authentication.

🃏 **Flashcard 26Q:** What is Coda?

A: AFS successor supporting disconnected operations, client hoarding, and conflict resolution.

🃏 **Flashcard 27Q:** What is GFS?

A: Google File System; designed for massive data storage, high fault tolerance, and concurrent access.

🃏 **Flashcard 28Q:** What is a volume in Coda?

A: A logical unit of storage that can be independently replicated and cached.

🃏 **Flashcard 29Q:** What is hoarding in Coda?

A: Pre-fetching files before going offline to support disconnected operations.

🃏 **Flashcard 30Q:** What is a Conflict Modification Log (CML)?

A: A log that stores changes made during disconnection, which is replayed when reconnected.

(Additional):

Key Concepts and Design Goals

1. **Fundamentals**
   DFS provides a unified view of a file system that spans multiple computers. The primary challenges include:

- **Scalability:** How does the system manage increasing amounts of data and number of users?
- **Fault Tolerance:** How does it keep working when one or more nodes fail?
- **Transparency:** How do clients access remote files as if they were local?
  Many systems (like NFS, AFS, HDFS, and Google File System) emphasize different trade-offs in these areas 2.

2. **Architectural Components**
   Typically, DFS comprises:
   - **Metadata Servers:** These manage directories, permissions, and the mapping of file names to data locations.
   - **Data Servers (or Nodes):** These hold the actual file data.
   - **Clients:** They interact with the DFS, performing operations like read, write, and file manipulation.
     Understanding the roles of these components helps in grasping how DFS handles operations across unreliable networks.

Distributed File System Design Choices

1. **Server State: Stateless vs. Stateful**
   - **NFS (Network File System):**
     NFS is designed as a
     stateless protocol. The advantage here is simplicity and robustness against server failures—since the server doesn't store client state, recovery is straightforward. However, the downside is that clients must handle caching and consistency themselves, sometimes leading to more complex client implementations.
   - **AFS (Andrew File System):**
     In contrast, AFS is a
     stateful system. It maintains information (like callback lists) regarding which clients cache which files. This allows more efficient whole-file caching and timely cache invalidation, ensuring that changes propagate with minimal delay. But maintaining this state adds complexity, especially during failures or when reestablishing state after a crash.
2. **Caching Strategies: Whole File vs. Block Caching**

- **Whole File Caching (e.g., AFS):**
  Clients cache entire files to reduce repeated network requests if the whole file will be used. This approach simplifies the client's logic about which parts of the file to refresh but can be inefficient if only a small portion of the file is needed.
- **Block-Level Caching (e.g., many NFS implementations):**
  Here, only the parts (or blocks) of the file required by the client are cached. It optimizes bandwidth and local memory use when applications only need specific sections of a file, yet it may introduce additional challenges in ensuring cache consistency and dealing with partial updates.

3. **Consistency Models and Fault Tolerance**
   Maintaining data consistency in a distributed setting is nontrivial. DFSs typically use strategies such as:
   - **Callbacks and Lease Mechanisms:** Used by AFS, where the server notifies clients when a cached copy becomes stale.
   - **Cache Invalidation Schemes:** Common in systems that avoid keeping server state, ensuring that clients eventually refresh outdated data.
   - **Replication and Journaling:** Some DFSs (and related log-structured file systems) employ data replication and write-ahead logging to quickly recover from crashes and ensure durability.
     Be prepared to discuss the trade-offs between strong consistency (where every client sees the same data at all times) and eventual consistency (which can improve performance and availability) 3.

Modern Systems and Practical Considerations

Modern DFS examples like Google File System (GFS) and Hadoop Distributed File System (HDFS) expand on these principles. They are optimized for the demands of large-scale data processing:

- **Large File Handling & High Throughput:**
  These systems assume files are huge and that most applications perform sequential read/write operations, favoring designs that optimize throughput over low latency.
- **Fault Tolerance with Commodity Hardware:**
  They are built with the assumption that hardware failures are common, so

extensive replication and automated recovery mechanisms are integral parts of the architecture.

Expect questions that ask you to compare these modern systems with older ones, explaining how design choices address different workload characteristics and reliability requirements.

Exam Preparation Strategies

- **Work Through Practice Problems:**
  Analyze past exam questions (e.g., comparing NFS vs. AFS on state management and caching). Write out the pros and cons of each design decision and justify why those choices might have been made given the environment they support.
- **Focus on Trade-Offs:**
  Delve into why one system might choose statelessness over statefulness, or whole file caching over block caching. Be ready to discuss the implications in terms of performance, scalability, and recovery.
- **Understand Recovery Mechanisms:**
  Crash recovery and fault tolerance are recurring themes. Know the benefits of approaches like log-structured file systems that sequentially write modifications, making recovery simpler compared to random disk writes.
- **Relate to Real-World Examples:**
  If asked, explain how systems like HDFS balance between consistency and availability in a cloud environment, using replication and specialized scheduling techniques.

By integrating these discussions and understanding the underlying trade-offs, you'll be well-prepared to answer a broad range of questions on distributed file systems.

Replication & Consistency:

What is Replication in Distributed System?

Replication is the process of **storing copies of the same data on multiple machines (nodes)** in a distributed system.

🎯 Why Replication?

- To **increase reliability** (if one node fails, others can serve the data)
- To **improve availability** (users get faster access from the nearest replica)
- To **enhance performance** (distribute load across multiple servers)

⚙️ How Replication Works

There are **two main components**:

1. **Replica Servers**: Machines that hold copies of the same data.
2. **Replication Protocol**: Ensures consistency across all replicas.

📌 Types of Replication

1. **Synchronous Replication**:
    - All replicas are **updated at the same time**.
    - Ensures **strong consistency**.
    - Slower due to waiting for all updates.
2. **Asynchronous Replication**:
    - Updates are **sent to replicas after** confirming to the client.
    - Faster, but **temporary inconsistency** may occur.

🔄 Replication Strategies

1. **Primary-Backup (Master-Slave)**:
    - One node (Primary) handles all writes.
    - Others (Backups) only read or receive updates from the primary.
2. **Multi-Master (Peer-to-Peer)**:
    - All replicas can accept updates.
    - Conflict resolution is needed.

📦 Example: Google Drive

- When you upload a file to Google Drive:
    - It's **replicated to multiple data centers**.
    - If one server goes down, you can still access your file from another.

📊 Summary Table

Feature	Synchronous	Asynchronous
Speed	Slower	Faster
Consistency	Strong	Eventual
Use Case	Banking, Finance	Social Media, Caching

Explain Replication Techniques briefly:

🧠 What Are Replication Techniques?

Replication techniques define **how data is copied and kept consistent** across multiple machines (replicas) in a distributed system.

🔷 1. Active Replication

✅ How it works:

- All replicas receive **the same request** and process it **simultaneously**.
- Requires **totally-ordered multicast** to keep operations in sync.

📌 Best for:

- Systems needing **high availability and strong consistency**.

🧠 Analogy:

All chefs in different kitchens cook the **same dish** using **the same recipe at the same time**.

🔷 2. Passive Replication (Primary-Backup)

✅ How it works:

- One replica (primary) **executes all operations**, then **sends updates** to backups.
- If the primary fails, a backup becomes the new primary.

📌 Best for:

- Systems that can tolerate **some delay in updates**.

🧠 Analogy:

A teacher solves a problem on the board, and students **copy it afterward**.

🔷 3. Quorum-Based Replication

✅ **How it works:**

- Uses a **voting system**:
  - To **read**, contact at least **Nr** replicas.
  - To **write**, contact at least **Nw** replicas.
- Ensures **Nr + Nw > N** (total replicas), so **read and write sets overlap**.

📌 **Best for:**

- Large distributed databases needing **flexible consistency**.

🧠 **Analogy:**

A decision can be taken only if **enough people agree** (e.g., 7 out of 10).

🔷 4. Gossip-Based (Epidemic) Replication

✅ **How it works:**

- Updates are **spread gradually**, like rumors or viruses.
- Eventually, all replicas **"hear" the update**.

📌 **Best for:**

- Systems where **eventual consistency** is acceptable (e.g., social media, DNS).

🧠 **Analogy:**

You tell a friend, they tell others, and eventually **everyone knows**.

🔷 5. Lazy Replication

✅ **How it works:**

- Updates are not sent immediately.
- They are **delayed** and sent **only when needed** (e.g., on access or sync).

📌 **Best for:**

- Systems that **prioritize performance** over immediate consistency.

🧠 **Analogy:**

You **don’t call your friend right away** about an update. You tell them **when you next meet**.

📌 Summary Table:

Technique	How It Works	Consistency	Speed	Best For
Active	All replicas process updates together	Strong	Slower	Critical systems
Passive	Primary does update, backups follow	Medium	Faster	Web servers, backup systems
Quorum-Based	Uses voting for read/write	Flexible	Varies	Distributed databases
Gossip-Based	Spread updates like a rumor	Eventual	Fast	Large-scale weakly consistent sys
Lazy	Delay updates until needed	Weak	Very Fast	Performance-prioritized systems

attachment:40c48838-633b-4b88-a392-f55504616af1:Chap_4_notes_-_Replication.pdf

Chap 4 notes - Replication.pdf

Explain in detail by covering all points:

🧩 1. What is Replication?

Replication means keeping **multiple copies of the same data or service** on different servers (replicas) to improve:

- ✅ **Availability** (if one server fails, another can serve),
- ✅ **Performance** (faster access from nearby servers),
- ✅ **Scalability** (handles more users).

🔄 2. Types of Replication

Type	Description
Data Replication	Only data is copied (e.g., FTP mirror sites, cached web pages).
Control Replication	Only the processing/control logic is copied (load sharing).
Combined Replication	Both data and control replicated—can be together or separately.

🗄️ 3. Data Store Model

🔷 Key Terms:

- **Replica Server**: Server holding a copy of the data.
- **Replica Manager**: Software managing reads/writes and syncs with other replicas.
- **Client**: Connects to one replica to read/write data.

🔷 Figure: Data Store Model

```
pgsql
Copy code
Client A → Replica 1
Client B → Replica 2
Client C → Replica 3
Client D → Replica 4
```

All replicas work together to act like a **single system** from the user's view.

⏱️ 4. Operation Timeline (Figure 2)

Shows how two clients access data with **Read (R)** and **Write (W)** operations over time.

- **Write**: Done locally first, then propagated to others.
- **Read**: Performed locally.

🧠 **Problem**: Time differences and update delays can cause **inconsistencies**.

📏 5. Consistency in Replication

Two major inconsistency types:

1. **Stale Data**: One replica is not updated yet.
2. **Different Operation Order**: Writes occur in different orders on different replicas (worse!).

🧠 6. Conflicting Operations

Conflict Type	Description
Read-Write	One read + one write at same time
Write-Write	Two writes at same time

Goal: Ensure **total ordering** of all operations, not just **per-replica order**.

🧪 7. Data-Centric Consistency Models

Model	Description
Strict	Every read returns the most recent write (ideal but impossible).
Linearizability	Orders by timestamps, hard to achieve in practice.
Sequential	All processes see the same order, but not necessarily real-time order.
Causal	Only causally related operations must be in same order everywhere.
FIFO	Writes from one client seen in same order by all, even if others interleave.

📊 Figures Explained

✅ Sequential Consistency (Figure 3)

- All clients should see **same order of writes**.
- ✔️ Valid: W(x)a → W(x)b is same for all.

- ❌ Invalid: W(x)b → W(x)a in some replicas.

✅ Causal Consistency (Figure 4)

- **Causally related writes** must appear in same order everywhere.
- ✔️ Valid: W(x)a → W(x)b
- ❌ Invalid: W(x)b appears before W(x)a (causality violated).

✅ FIFO Consistency (Figure 5)

- Clients must see **writes from the same source** in order.
- ✔️ Valid: W(x)a → W(x)b
- ❌ Invalid: W(x)b appears before W(x)a.

🧵 8. Weaker Consistency Models (Grouped by Sync)

Model	Description
Weak	Writes are synced only at critical section entry/exit.
Release	Sync separated into `acquire()` and `release()` operations.
Lazy Release	Only syncs on `acquire()`, not on `release()`. Saves communication.
Entry	Sync variables tied to specific data items (fine-grained control).

⚖️ 9. CAP Theorem & Eventual Consistency

CAP Theorem (Consistency, Availability, Partition-tolerance):

You can only have **two out of three**.

- **Eventual Consistency**: All replicas become consistent **eventually**.
- Used by **DNS**, **CDNs**, **Web caches**, etc.

👤 10. Client-Centric Models

Assumes **more reads than writes**, and **mobile clients**.

Model	Meaning

Monotonic Reads	Once a client sees a value, it won’t see older values later.
Monotonic Writes	Writes by same client occur in order.
Read-Your-Writes	A client always sees its most recent write.
Write-Follows-Reads	Writes reflect the latest data the client has read.

Each model has figures (6–9) showing valid/invalid timelines.

⚙️ 11. Consistency Protocols

Protocol	Key Idea
Single Server	Central server does all writes (no replication).
Primary-Backup	Writes go to primary, backups copy it.
Migration	Data moved to the client replica on access.
Migrating Primary	Only write role moves, not the data.
Active Replication	All replicas process writes simultaneously.
Quorum-Based	Writes/reads done by subsets with majority rules.

🔄 12. Update Propagation Techniques

Method	Description
Send Data	Full data is sent to replicas.
Send Operation	Only the operation is sent (like logs).
Send Invalidation	Tell replicas their copy is stale.

📤 13. Push vs Pull

Mode	When to use	Example
Push	High freshness, low update rate	News alerts
Pull	High update rate, low read rate	Social media feeds

🕒 14. Leases

- Replicas get **temporary rights (leases)** to receive updates.
- Longer leases for:

  - Frequently accessed data
  - Clients that often renew
  - Low overhead conditions

📡 15. Multicast vs Unicast

Method	Usage
Multicast	Efficient for many replicas (e.g., clusters)
Unicast	One-to-one, more common, easier to implement

🧭 16. Replica Placement

Type	Description
Permanent	Main server or mirrors owned by provider.
Server-Initiated	Placed near users to boost performance.
Client-Initiated	Temporary copies made by clients (e.g., browser cache).

🔀 17. Dynamic Replication

System automatically:

- Adds replicas when load increases.
- Deletes/migrates replicas when load drops.
- Example: **RaDaR Web Hosting**.

🛰️ 18. Request Routing

- Clients should connect to **the best replica** (fastest or closest).
- Not easy: requires **naming, discovery, and redirection** mechanisms.

📘 Flashcards for Replication

🃏 **Flashcard 1Q:** What is replication in distributed systems?

A: Replication is the process of storing copies of the same data or service on multiple machines to improve availability, fault tolerance, and performance.

🃏 **Flashcard 2Q:** What are the three main goals of replication?

A: Increase availability, improve performance, and ensure fault tolerance.

🃏 **Flashcard 3Q:** What are the types of replication?

A: Data replication, Control replication, Combined replication.

🃏 **Flashcard 4Q:** What are replica managers?

A: Software components responsible for handling reads/writes and maintaining consistency across replica servers.

🃏 **Flashcard 5Q:** What is data consistency in replication?

A: Ensuring that all replicas reflect the same data values at any given point in time.

🃏 **Flashcard 6Q:** Name two types of operation conflicts in replication.

A: Read-Write conflict and Write-Write conflict.

🃏 **Flashcard 7Q:** What is sequential consistency?

A: All processes see the same sequence of operations, but not necessarily in real-time order.

🃏 **Flashcard 8Q:** What is causal consistency?

A: Writes that are causally related must be seen by all processes in the same order.

🃏 **Flashcard 9Q:** What is FIFO consistency?

A: Writes from a single process are seen in the same order by all processes.

🃏 **Flashcard 10Q:** What does the CAP theorem state?

A: A distributed system can only achieve two of the following three: Consistency, Availability, and Partition Tolerance.

🃏 **Flashcard 11Q:** What is eventual consistency?

A: A model where updates will eventually reach all replicas, so they become consistent over time.

🃏 **Flashcard 12Q:** Name four client-centric consistency models.

A: Monotonic Reads, Monotonic Writes, Read-Your-Writes, and Write-Follows-Reads.

🃏 **Flashcard 13Q:** What is active replication?

A: All replicas process every request in the same order simultaneously.

🃏 **Flashcard 14Q:** What is passive replication?

A: Only the primary replica processes the request; updates are then sent to backups.

🃏 **Flashcard 15Q:** What is quorum-based replication?

A: A technique using a minimum number of read (Nr) and write (Nw) acknowledgments to ensure consistency.

🃏 **Flashcard 16Q:** What is the difference between sending data and sending operation in update propagation?

A: Sending data transfers the updated value; sending operation transmits the action to be performed.

🃏 **Flashcard 17Q:** What is a lease in replication?

A: A temporary contract where a replica gets the right to receive updates for a fixed period.

🃏 **Flashcard 18Q:** When is push-based update propagation preferred?

A: When data is read frequently but updated rarely.

🃏 **Flashcard 19Q:** What are three types of replica placement?

A: Permanent replicas, Server-initiated replicas, and Client-initiated replicas.

🃏 **Flashcard 20Q:** What is dynamic replication?

A: A strategy where replicas are created or removed based on system load and access patterns.

Fault Tolerance:

What is Fault Tolerance in DS?

Fault tolerance in distributed file systems (DFS) is the ability of the system to continue operating reliably and without data loss, even when individual components (like a server, network, or disk) fail. This resilience is a core design requirement because, by nature, a DFS is made up of many nodes spread across different machines which are prone to failure.

attachment:898ab817-1823-4026-9d35-59af30628cd5:Chap_8_notes_-_FT.pdf

Chap 8 notes - FT.pdf

🧱 1. Dependability in Distributed Systems

✅ Key Properties:

Property	Meaning
Availability	Probability system is working at a given time (e.g., 99.9999%).
Reliability	Ability to run continuously without failure.

Safety	System fails in a way that doesn’t cause catastrophe. (E.g. nuclear plant control systems)
Maintainability	Ease of repairing the system quickly.

🔐 Security-related (Covered elsewhere):

- **Integrity** – data not tampered with
- **Confidentiality** – data protected from unauthorized access

🧠 **Analogy**: Think of a hospital system:

- **Availability** = hospital is open
- **Reliability** = surgery goes smoothly
- **Safety** = mistakes don’t harm patients
- **Maintainability** = doctors/nurses can fix issues fast

⚠️ 2. Faults, Errors, and Failures

Term	Meaning
Fault	Root cause (e.g. design bug, power failure)
Error	System state deviates from expected
Failure	Observable incorrect service behavior

➡️ Fault → Error → Failure (This chain defines how faults become visible)

🔁 3. Types of Faults

Type	Description	Example
Transient	Occurs once, then disappears	WiFi cut by interference
Intermittent	Comes and goes	Loose cable, race condition
Permanent	Persistent until fixed	Burned-out chip

🧠 **Analogy**: Think of your phone:

- Transient = network dropout for a second
- Intermittent = charger sometimes works
- Permanent = screen cracked

🧩 4. Types of Failures in Distributed Systems

Failure Type	Meaning
System Failure	CPU, OS, memory failure
Process Failure	Crashed, deadlock/livelock
Storage Failure	Disk inaccessible
Communication Failure	Network drop or partition

➡️ Partial failure: only part of system fails (unique to distributed systems)

💥 5. Failure Models

Model	Description
Crash Failure	Stops working cleanly
Fail-Stop	Clients know it failed
Fail-Silent	Fails silently
Omission Failure	Doesn't respond (Receive or Send Omission)
Timing Failure	Response too late/early
Response Failure	Wrong output
Arbitrary Failure	Any weird behavior (Byzantine fault)

🧠 Real-Life Example:

- **Crash** = Laptop shuts off
- **Fail-Silent** = You think email was sent, but it never was
- **Byzantine** = Server sends different data to users randomly

---

🛡️ 6. Fault Tolerance Concepts

✳️ Goal:

Continue providing correct service despite faults.

🔄 Failure Masking via Redundancy:

Redundancy Type	Description
Information	Extra info to recover lost data (e.g., parity bits)

Time	Retry actions (e.g., resend message)
Physical	Duplicate systems/components (e.g., RAID, replicas)

🧠 Analogy: Backups on Google Drive = Physical Redundancy

👥 7. Process Resilience

Use **groups of replicated processes** to survive failure.

Type	Description
Flat Group	No leader, equal peers; more complex but no single point of failure
Hierarchical Group	One coordinator makes decisions; simpler but single point of failure

🧠 Analogy:

- Flat = team project where everyone contributes equally
- Hierarchical = group leader does most planning

✅ Implemented via **Replicated State Machines**:

- All replicas execute same actions in the same order
- Needs **consensus** to ensure identical behavior

✅ 8. Consensus in Faulty Environments

📘 Required Properties:

1. **Agreement** – All processes choose same value
2. **Validity** – Chosen value was actually proposed
3. **Termination** – Everyone eventually decides

🔵 Two-Army Problem (Figure 1)

- Two generals must attack at the same time but use unreliable messengers.
- Acknowledgments never guarantee certainty → Infinite loop of confirmation.

🧠 Analogy: You text a friend to meet at 6 PM. You say “okay?” They say “yes”, but you wonder, “did they get my okay to their okay?” Infinite chain.

🏰 Byzantine Generals Problem

- Some generals lie → causes confusion.
- Needs **2k + 1** nodes to tolerate **k Byzantine faults**.

⭐ Paxos Algorithm (Used by Google)

- **Proposer** suggests a value
- **Acceptors** vote (must receive majority)
- **Value is accepted** if quorum reached
- Tolerates process crash and message loss
- May stall (no termination) to avoid violating agreement

🧠 Analogy: Group of friends deciding movie—leader suggests, others confirm. Only after majority agrees, decision is made.

📡 9. Reliable Communication

Common RPC Failures:

1. Can't locate server
2. Request lost
3. Server crashes
4. Reply lost
5. Client crashes

Semantics:

- **At-least-once**: Retry, may duplicate
- **At-most-once**: Prevent duplicates
- **Maybe**: No guarantees

🧠 Analogy: Sending email—did they get it? Did they reply? You send again just in case.

🧬 10. Orphan Computations

- Server continues processing for a client that has crashed.

Solutions:

- **Extermination**: Kill all leftovers
- **Reincarnation**: Server kills old session on restart
- **Gentle reincarnation**: Kills only if parent not reachable
- **Expiration**: Processes timeout unless renewed

👥 11. Reliable Group Communication

🌐 Multicast (Figure 2, 3):

- Sequence numbers & ACKs → scalable challenge
- **NACKs** used to reduce overhead
- **Hierarchical multicast**: Break group into subgroups, each with a coordinator

---

💣 Atomic Multicast:

- Deliver message to **all or none**
- Ensure **same order at all receivers**

♻️ 12. Recovery in Distributed Systems

🔄 Forward vs. Backward

Type	Description
Forward	Repair error and continue (rare)
Backward	Roll back to a known good state (common)

💾 13. Checkpointing (Rollback Recovery)

Method	Description
State-based	Save full state (checkpoints)
Operation-based	Log actions for replay (undo/redo)

📉 Techniques:

- **Incremental** – Save only changed parts (copy-on-write)
- **Asynchronous** – Continue running while saving
- **Compressed** – Reduce size for storage
- **fork()** – Create child to dump memory (Unix)

⚠️ 14. Domino Rollback (Figure 4–6)

- One process rolls back → others must follow → leads to massive undo
- Caused by **uncoordinated checkpoints**

🧠 Analogy: One student forgets to save project, so the whole group restarts from earlier version.

🛠️ 15. Consistent Checkpoints

Type	Description
Strongly Consistent	No in-transit messages; system paused during checkpoint
Consistent	Use snapshot algorithms (e.g., Chandy-Lamport) to ensure correctness

🔄 Checkpoint Coordination

(Figure 8–9): Two-Phase Commit

1. All take tentative checkpoints
2. If successful, make permanent
3. Otherwise, discard

🚀 16. Optimistic Checkpointing (Juang & Venkatesan)

- Checkpoints are **local, independent**
- Use logs to **replay messages**

- **Suppress message sending during roll-forward**

🧠 Analogy: Like "undo" in a Word doc — only retype parts you lost

📚 Summary

Concept	Key Idea
Fault Tolerance	Continue service despite faults
Redundancy	Main failure masking technique
Process Resilience	Replication & consensus needed
Reliable Comm.	Deal with RPC, multicast failures
Recovery	Use rollback via checkpoints
Consensus	Paxos is real-world applicable

📘 Flashcards for Fault Tolerance

🃏 **Flashcard 1Q:** What is fault tolerance?

A: The ability of a system to continue functioning correctly even in the presence of faults.

🃏 **Flashcard 2Q:** What are the four dependability properties?

A: Availability, Reliability, Safety, and Maintainability.

🃏 **Flashcard 3Q:** Define availability.

A: The probability that a system is operational and accessible at a given time.

🃏 **Flashcard 4Q:** Define reliability.

A: The ability of a system to run continuously without failure.

🃏 **Flashcard 5Q:** Define safety.

A: The system will not cause damage or harm in the event of a failure.

🃏 **Flashcard 6Q:** Define maintainability.

A: The ease with which a system can be repaired or updated.

🃏 **Flashcard 7Q:** What is the difference between a fault, error, and failure?

A: A fault causes an error, which may lead to a failure.

🃏 **Flashcard 8Q:** What are transient faults?

A: Faults that occur once and disappear (e.g., temporary interference).

🃏 **Flashcard 9Q:** What are intermittent faults?

A: Faults that occur sporadically and unpredictably.

🃏 **Flashcard 10Q:** What are permanent faults?

A: Faults that persist until the faulty component is replaced or repaired.

🃏 **Flashcard 11Q:** What is a crash failure?

A: When a process halts and does not recover.

🃏 **Flashcard 12Q:** What is a fail-stop failure?

A: A failure where the system halts and notifies others.

🃏 **Flashcard 13Q:** What is an omission failure?

A: Failure to send or receive messages.

🃏 **Flashcard 14Q:** What is a timing failure?

A: The system responds too early or too late.

🃏 **Flashcard 15Q:** What is a response failure?

A: When a system gives an incorrect response.

🃏 **Flashcard 16Q:** What is an arbitrary failure?

A: Also known as Byzantine failure, it includes any incorrect or unpredictable behavior.

🃏 **Flashcard 17Q:** What are the three types of redundancy for fault tolerance?

A: Information redundancy, time redundancy, and physical redundancy.

🃏 **Flashcard 18Q:** What is information redundancy?

A: Using extra information like checksums or parity bits to detect and recover from errors.

🃏 **Flashcard 19Q:** What is time redundancy?

A: Retrying operations or messages to overcome temporary faults.

🃏 **Flashcard 20Q:** What is physical redundancy?

A: Having multiple hardware or software components that can take over in case of failure.

🃏 **Flashcard 21Q:** What are the types of process groups for resilience?

A: Flat groups and hierarchical groups.

🃏 **Flashcard 22Q:** What is the Two-Army problem?

A: A problem showing that it's impossible to guarantee coordination between two parties over unreliable communication.

🃏 **Flashcard 23Q:** What is the Byzantine Generals Problem?

A: A consensus problem where some members may act maliciously or send conflicting information.

🃏 **Flashcard 24Q:** What is Paxos?

A: A consensus algorithm that allows a distributed system to agree on a value despite failures.

🃏 **Flashcard 25Q:** What are the three properties required for consensus?

A: Agreement, Validity, and Termination.

🃏 **Flashcard 26Q:** What is reliable communication?

A: Ensuring messages are delivered correctly, in order, and without duplication.

🃏 **Flashcard 27Q:** What are common RPC failures?

A: Cannot locate server, lost request, lost reply, client/server crash.

🃏 **Flashcard 28Q:** What is orphan computation?

A: Server continues execution for a client that has crashed.

🃏 **Flashcard 29Q:** What are the solutions to orphan computation?

A: Extermination, Reincarnation, Gentle Reincarnation, Expiration.

🃏 **Flashcard 30Q:** What is atomic multicast?

A: A multicast where messages are delivered to all recipients or none, and in the same order.

🃏 **Flashcard 31Q:** What is rollback recovery?

A: Restoring a system to a previously saved checkpoint after failure.

🃏 **Flashcard 32Q:** What is checkpointing?

A: Saving the state of a process to allow recovery after a crash.

🃏 **Flashcard 33Q:** What is incremental checkpointing?

A: Saving only the changes since the last checkpoint.

🃏 **Flashcard 34Q:** What is domino rollback?

A: A cascading rollback where one process's failure forces others to roll back, possibly to the start.

🃏 **Flashcard 35Q:** What are consistent checkpoints?

A: A global checkpoint where no messages are in transit, or handled using snapshot algorithms.

🃏 **Flashcard 36Q:** What is optimistic checkpointing?

A: Checkpoints are taken independently, assuming few failures, and logs are used to restore consistency.

(Additional)

Here's how DFSs typically achieve fault tolerance:

1. Data Replication

One of the primary mechanisms is **data replication**. In DFS architectures such as the Hadoop Distributed File System (HDFS) or the Google File System (GFS), every piece of data (often split into smaller units or blocks) is stored on multiple machines rather than on a single server. For example, HDFS typically uses a replication factor of 3 by default. This means each block is stored on three different nodes, usually spread across different racks to minimize the risk associated with hardware or network failures. If one node goes offline, the data remains accessible from the other replicas.

Mechanism	**Description**	**Example in DFS**
Replication	Storing multiple copies of data across different nodes	HDFS (replication factor of 3), GFS
Erasure Coding	Encodes data in a redundant way using parity blocks to recover lost data	Some modern DFS solutions

This strategy not only ensures availability but also helps maintain data integrity even when errors occur during transmission or storage .

2. Heartbeat Mechanisms and Health Monitoring

DFSs use **heartbeat messages** between components to continuously monitor the health of the nodes. Here's how it typically works:

- **Heartbeat Signals:** Data nodes periodically send heartbeat signals to a central controller or metadata server (like HDFS's NameNode).

- **Failure Detection:** If the controller stops receiving a heartbeat within a specified timeout period, it marks that node as failed.
- **Re-replication:** Once a failure is detected, the system automatically initiates replication processes to create new copies of the data that were stored on the failed node, thus restoring the desired replication level.

This proactive approach minimizes downtime and ensures swift recovery from failures.

3. Checkpointing and Metadata Server Redundancy

Besides the data itself, DFSs must also safeguard **metadata** (information about file locations, permissions, etc.). This is typically achieved by:

- **Checkpointing:** Periodically saving a snapshot of the current state of the metadata. In systems like HDFS, the NameNode creates checkpoints that help in quickly recovering or restarting if a failure occurs.
- **Redundancy:** Some systems use a secondary metadata server (or standby NameNode) that can take over the responsibilities if the primary server fails. This minimizes the risk of a single point of failure in the system's control logic.

By protecting both data and metadata with careful redundancy measures, the DFS ensures the entire system remains operable even when key elements fail.

4. Journaling and Write-Ahead Logging

To deal with write operations and ensure data consistency during failures, many DFSs utilize **journaling** or **write-ahead logging**:

- **Write-Ahead Logging:** Before any changes are made to the file system, the system records the intended changes in a log. If a crash or unexpected shutdown occurs during file operations, these logs can be replayed to bring the system back to a consistent state without data loss.
- **Journaling:** By maintaining a journal of file system transactions, the DFS is able to quickly roll back or complete pending operations after recovery, ensuring that the file system's integrity is not compromised by partial writes.

This mechanism guarantees that even during abrupt failures, operations can be resumed without corrupting either data or metadata.

5. Self-Healing Capabilities

Modern DFS designs incorporate **self-healing** features, where the system automatically repairs itself without manual intervention:

- **Automatic Data Reconstruction:** Following a node failure, the DFS can automatically reconstruct lost or corrupted data from the remaining healthy replicas.
- **Error Detection:** Techniques such as checksums are employed to detect data corruption. When corrupt data is identified, the DFS retrieves a correct replica, maintaining the integrity of stored files.
- **Dynamic Resource Reallocation:** The system can reassign pending operations or redirect client requests to alternative nodes when failures are detected.

This self-healing aspect is crucial for large-scale environments where manual intervention for every failure would be infeasible .

Conclusion

In summary, fault tolerance in DFS is achieved through an interplay of techniques:

- **Data replication** ensures data is available even when some nodes fail.
- **Heartbeat messages and health monitoring** provide early detection of issues, triggering automatic recovery.
- **Checkpointing and metadata redundancy** protect the control mechanisms of the file system.
- **Journaling/write-ahead logging** maintains consistency in the event of write failures.
- **Self-healing mechanisms** allow the system to automatically mend itself after errors occur.

Each of these techniques involves trade-offs between performance, consistency, and resource utilization, but together they ensure that large-scale distributed file systems remain robust and reliable in face of numerous potential failures.

System Architecture & Communications

🌐 1. System Architecture in Distributed Systems

🧱 Components:

- **Software components** (apps, services)
- **Processing nodes** (servers, devices)
- **Networks** (connecting everything)

🎯 Key Idea:

The software must be **distributed across hardware** — that's what makes it a distributed system.

🏛️ 2. Software Architecture vs System Architecture

- **Software architecture**: How programs are logically structured (e.g., layered, object-oriented).
- **System architecture**: Where programs run physically, which machine does what.

🧠 **Analogy**: Think of software architecture like planning a city (schools, parks), and system architecture as placing them physically on a map.

🖥️ 3. Communication Architectures

a. Client-Server (Figure 1)

- **Client** sends request → **Server** processes → sends response
- Used in most apps: banking, databases, etc.

🧠 Analogy: Ordering food at a restaurant. You're the client, the kitchen is the server.

---

b. Vertical Distribution (Multi-tier) (Figures 2–3)

- Server's responsibilities split into layers:
  - **Presentation/UI layer**
  - **Application logic**
  - **Database**

- Each layer talks to the next.

🧠 Analogy: A call center — first you speak to a receptionist, then a specialist, then billing.

---

c. Horizontal Distribution (Figure 4)

- Same service replicated across multiple servers.
- **Load balancer** directs traffic in round-robin fashion.

🧠 Analogy: A queue with 3 bank tellers, each can help you do the same tasks.

---

d. Peer-to-Peer (P2P) (Figure 5)

- Every node is both client and server.
- Used in file-sharing, blockchain, etc.

🧠 Analogy: Everyone brings food and takes food at a potluck party.

---

e. Overlay Network in P2P

- **Structured overlay**: like DHTs, organized
- **Unstructured overlay**: random graph of neighbors

🧠 Analogy: Structured = library catalog; Unstructured = random phonebook

---

f. Hybrid Architectures

- Mix of architectures:
  - **Superpeer** (e.g., Skype): powerful nodes help regular ones
  - **Collaborative** (e.g., BitTorrent): uses tracker + P2P
  - **Edge Server** (e.g., Akamai): servers closer to users

🧠 Analogy: Edge server = local mini-library near your house instead of main city library

---

🧵 4. Processes and Server Architecture

- **Thread vs Process**:
  - **Threads**: share memory, lightweight

    - **Processes**: isolated memory

🧠 Analogy: Threads = roommates; Processes = separate apartments

- **Multithreaded server** = more responsive
- **Stateful server** = remembers clients
- **Stateless server** = treats each request as new

🧠 Analogy: Stateful = Netflix keeps your watchlist

Stateless = Google search doesn't remember your last search

🔀 5. Code Mobility

- Move code to other servers to:
    - **Reduce load**
    - **Bring computation closer to data**

Type	Explanation
Weak mobility	Move code only, restart fresh
Strong mobility	Move code + execution state

🧠 Analogy: Weak = moving with no luggage; Strong = move with luggage & resume where you left off

🔗 6. Communication in Distributed Systems

🔁 Reasons:

- **Synchronization**: Who's alive? What's done? Locking?
- **Data sharing**: Task info, results, updates

📢 Modes of Communication:

Mode	Explanation
Shared Memory	Fast but only possible if physically available
Message Passing	Universal and used over networks

💬 7. Message Passing

- Uses `send()` and `receive()` over the network.
- Can be:
  - **Connectionless** (UDP)
  - **Connection-oriented** (TCP)

🔄 8. Communication Modes

Type	Explanation	Analogy
Data-oriented	Send data only	Email
Control-oriented	Send + trigger action	Phone call

Sync Type	Meaning
Synchronous	Sender waits
Asynchronous	Sender continues

Persistence	Meaning
Transient	Delivered only if receiver is ready
Persistent	Saved until receiver is available

(Figure 6 shows combinations: e.g., transient + sync, persistent + async, etc.)

✅ 9. Communication Reliability

- **Reliable** = Guaranteed delivery
- **Unreliable** = May lose messages

Ordering Guarantee	Example
Ordered	Video calls
Unordered	SMS sometimes

🔌 10. Communication Abstractions

Helps hide complexity of direct `send()` and `receive()`.

a. Message-Oriented Middleware (MOM)

- Example: IBM MQ, MPI
- Uses **queues**:
  - App sends to send queue
  - Middleware forwards to receive queue

(Figure 7 shows queuing with routers)

🧠 Analogy: Postal service – drop message in box, it's routed to recipient

b. Remote Procedure Call (RPC) (Figure 8)

- Looks like local function call → internally sends message to remote server
- Steps:
  - Call → Stub → Marshal → Send → Server Stub → Unmarshal → Execute → Return

🧠 Analogy: You call a help desk number, but it redirects to a real person behind the scenes

- **Asynchronous RPC** = no need to wait for reply
- **Deferred synchronous RPC** = response comes later
- **Binding service**: helps find the server (name → IP)

c. Remote Method Invocation (RMI)

- Object-based RPC
- Supports passing remote objects

🧠 Analogy: Like accessing someone else's object methods on your own code

⚠️ 11. Challenges with RPC/RMI

- Failures more likely than local calls
- Extra overhead due to marshalling, threads, network delays

🧠 Analogy: Online meeting vs. face-to-face—more things can go wrong

👥 12. Group Communication

- Send to **multiple recipients**
  - **Broadcast** = to all
  - **Multicast** = to a group

🧠 Analogy: Group WhatsApp message

Used for:

- **Replica update**
- **Service discovery**
- **Event notification**

🔁 Gossip-based Communication

- Info spread like a **rumor**: push to random nodes, they spread if they didn’t know

---

📣 13. Event-Based Communication

- **Publisher** sends event
- **Subscriber** receives events of interest
- No direct connection needed

🧠 Analogy: You follow a YouTube channel. When they post (publish), you get notified (subscribe).

🧠 14. Distributed Shared Memory (DSM) (Figure 9)

- Emulates shared memory across multiple machines

🧠 Analogy: Everyone sees and edits the same Google Doc from different computers

---

🧱 15. Tuple Spaces

- Shared space to put/get data tuples
- Processes read or write as needed

🧠 Analogy: A shared sticky note board at home – write what you want, others read it

---

🎥 16. Streams (Figure 10)

- For **continuous media** (audio/video)
- Requires **timing**, **sync**, and **QoS**

🧠 Analogy: Live TV stream must be played in order and on time

🔄 Token Bucket Model:

- Limits sending rate by **tokens**
- Tokens added periodically
- If no tokens → wait

🧠 Analogy: Prepaid data plan—you can't use more unless new tokens are added

---

🔗 Stream Synchronization

- **Client-based**: Receiver syncs audio/video
- **Server-based**: Multiplex streams together

✅ Summary:

Topic	Key Idea
Architectures	Client-server, P2P, hybrid
Communication Modes	Sync/Async, Persistent/Transient
Message Models	RPC, RMI, MOM
Group/Event Communication	Multicast, gossip, pub-sub
Advanced Abstractions	DSM, Tuple space, Streams