Distributed-Computing-Series-2-Important-Topics

For more notes visit

https://rtpnotes.vercel.app

1. Illustrate Richart Algorithm for achieving mutual exclusion

Think of **mutual exclusion** like waiting in line to use an ATM. Only **one person** can use it at a time. Similarly, in a **distributed system**, multiple computers (processes) may need to access a shared resource (like a file or a database), but only **one should access it at a time**.

The **Ricart–Agrawala Algorithm** helps computers **take turns** fairly and efficiently. Here's how it works:

How It Works (Step-by-Step)

🟢 Step 1: Asking for Permission

- If a **process (computer) wants to use a resource**, it sends a **"REQUEST" message** to all other processes.
- This message has a **timestamp** (like writing down the time you arrived in line).

🟠 Step 2: Receiving a Request

- When another process receives a REQUEST, it checks:
  - **"Am I using the resource?"**
    - If **No** → It sends back **"REPLY" (Yes, you can use it)** immediately.
    - If **Yes** → It **waits and sends REPLY later** after it's done.
  - If it's also **waiting** for the resource, it compares timestamps:
    - **Smaller timestamp (arrived first)** → That process gets priority.
    - **Larger timestamp (arrived later)** → It waits.

🟡 Step 3: Using the Resource

- The process **enters the critical section (CS)** (uses the resource) **only after receiving REPLYs from all other processes**.
- This ensures only **one process uses it at a time**.

🔴 Step 4: Releasing the Resource

- Once it's done, it sends **REPLYs** to any waiting processes.
- The next process in line can now enter the CS.

---

2. Explain how wait for graph can be used in deadlock detection

Imagine you and your friends are passing around books. You can only **borrow a book** if the person who has it is **done with it and returns it**. But what if **everyone is waiting for someone else to return a book**? No one can proceed—this is a **deadlock**!

In a **distributed system**, processes (computers) request resources (like files, memory, or database access). If a process is **waiting for another process** to release a resource, we can represent this situation using a **Wait-For Graph (WFG)**.

What is a Wait-For Graph (WFG)?

A **Wait-For Graph (WFG)** is a **visual way to check for deadlocks** in a system. It is a **directed graph** where:

- **Nodes** represent **processes** (P1, P2, P3, etc.).
- **Edges (arrows)** represent **waiting relationships** (e.g., P1 → P2 means **P1 is waiting for P2** to release a resource).

How is WFG Used for Deadlock Detection?

1. **Build the Graph**:
    - Each process that is **waiting for a resource** is **connected** to the process holding that resource.
2. **Check for Cycles** 🔄:

- If there is a **cycle (loop) in the graph**, it means a **deadlock** has occurred.
- Example of a cycle:
  - `P1 → P2 → P3 → P1 (Deadlock! 🚨)
- If **no cycle** is found, the system is **deadlock-free**.

Example: Detecting Deadlock Using WFG

Scenario:

- **P1 is waiting for a resource held by P2**
- **P2 is waiting for a resource held by P3**
- **P3 is waiting for a resource held by P1**

WFG Representation:

- `P1 → P2 → P3 → P1 (Cycle detected! Deadlock ❌)`
- Since there's a **cycle**, no process can proceed, confirming a **deadlock**.

---

3. Explain checkpointing and rollback recovery in detail

Imagine you are playing a video game, and you reach an important level. To avoid losing progress if something goes wrong, you **save the game** at certain points. If you fail later, you can **reload** the saved checkpoint instead of starting from scratch.

This is exactly how **Checkpointing and Rollback Recovery** work in **distributed systems** to handle failures!

What is Checkpointing?

Checkpointing is the process of **saving the state of a process at a certain point** so that if a failure occurs, it can restart from that point instead of starting over.

Why Use Checkpoints?

- Reduces the amount of lost work after a failure.
- Helps systems **recover quickly** instead of re-executing everything.
- Saves **CPU time and network resources**.

🛠️ **Example:**
A bank transaction system saves its progress after every 100 transactions. If the system crashes after 150 transactions, it **restores the last saved state (100 transactions) and reprocesses the last 50** instead of all 150.

What is Rollback Recovery?

Rollback Recovery is the process of **restoring a system to a previous consistent checkpoint** after a failure.

Steps in Rollback Recovery:

1. **Detect Failure** 🚨
   - The system notices that a process has failed.
2. **Restore Checkpoint** 🔄
   - The system **reloads the last saved checkpoint** for that process.
3. **Re-execute** ⏩
   - The process **continues from where it left off**, avoiding major data loss.

🛠️ **Example:**
If an airline booking system crashes **after booking 50 tickets**, it restores the **last saved checkpoint** (e.g., after booking 40 tickets) and **reprocesses the last 10 tickets**.

Types of Checkpointing (How to Save Progress)

There are **three main ways** to implement checkpointing:

A. Uncoordinated Checkpointing (Independent Checkpoints)

- Each process **saves its own state independently**.
- **Problem:** It may lead to a **Domino Effect** (rolling back too far).

🛠️ **Example:**
If process **P1 rolls back**, but it has sent data to **P2**, then **P2 must also roll back**, and so on. This can cause the system to **restart from a very old state**.

B. Coordinated Checkpointing (Planned Checkpoints)

- All processes **coordinate** and save a **consistent global state**.
- Prevents **the Domino Effect** and ensures all processes are in sync.

- Uses a **coordinator process** to signal all processes to take checkpoints together.

🛠️ **Example:**
An online shopping system **saves a checkpoint every hour** across all servers. If one server crashes, it restores **all servers to the last saved state** to maintain consistency.

C. Communication-Induced Checkpointing (Smart Checkpointing)

- Checkpoints are **automatically triggered** when needed.
- Prevents unnecessary rollbacks and reduces overhead.
- Uses **extra information** in messages to decide when to save a checkpoint.

🛠️ **Example:**
A cloud storage service **automatically saves checkpoints** when many files are being uploaded, ensuring smooth recovery if a failure occurs.

Message Handling in Rollback Recovery

When recovering from a failure, messages between processes can be affected:

Types of Messages:

1. **In-Transit Messages** – Sent but not yet received.
2. **Lost Messages** – Sent before a crash but not delivered.
3. **Orphan Messages** – Received but the sender's checkpoint doesn't show that it was sent.

---

4. List the disadvantages of distributed shared memory

Imagine Distributed Shared Memory (DSM) as **a giant shared notebook** that many people (computers) can write and read from. It **makes sharing data easy**, but it has some problems too.

1. Programmers Need to Learn Special Rules 📚

- Just like **different schools have different rules** for writing assignments, DSM has **different consistency rules** for how data is shared.
- Programmers must **understand these rules** to avoid errors.

🛠️ **Example:** If two people write on the same notebook at the same time, how do we decide whose writing is correct? DSM needs rules for such cases.

2. DSM Uses a Lot of Resources (Slow Communication) ⏳

- DSM works by **sending messages between computers** to keep data updated.
- These messages **take time** and **slow things down**.

🛠️ **Example:** Imagine you and your friend are writing in a shared online document, but updates take **5 seconds to appear**. It would be frustrating!

3. Not as Fast as Custom Solutions 🚗💨

- DSM is **one-size-fits-all**, but some applications need **faster, custom solutions**.
- Custom-built systems can be **optimized** for speed and efficiency.

🛠️ **Example:** Buying a **ready-made suit** vs. getting a **tailor-made suit**. DSM is like the ready-made suit—it works, but **not always a perfect fit**.

---

5. Explain consenus algorithm for crash failures under synchronous systems

What is a Consensus Algorithm?

A **consensus algorithm** helps multiple processes (computers) **agree on a single value** in a distributed system. This is important when some processes **fail or crash** but the system must continue working.

Understanding Synchronous Systems

A **synchronous system** means:
✔ **Time is predictable** – All messages are delivered within a known time.
✔ **Processing is predictable** – Each process takes a known time to compute.
✔ **Failures can be detected** – If a process doesn't respond in time, we assume it has **crashed**.

🔹 **Example:** Imagine a group of friends voting on where to eat. Everyone must answer **within 10 seconds**, or they are considered unavailable.

Consensus in the Presence of Crash Failures

In a **synchronous system**, crash failures happen when **some processes stop working** but don’t send incorrect data. The goal of the consensus algorithm is to let the **remaining processes agree** on a decision, even if some fail.

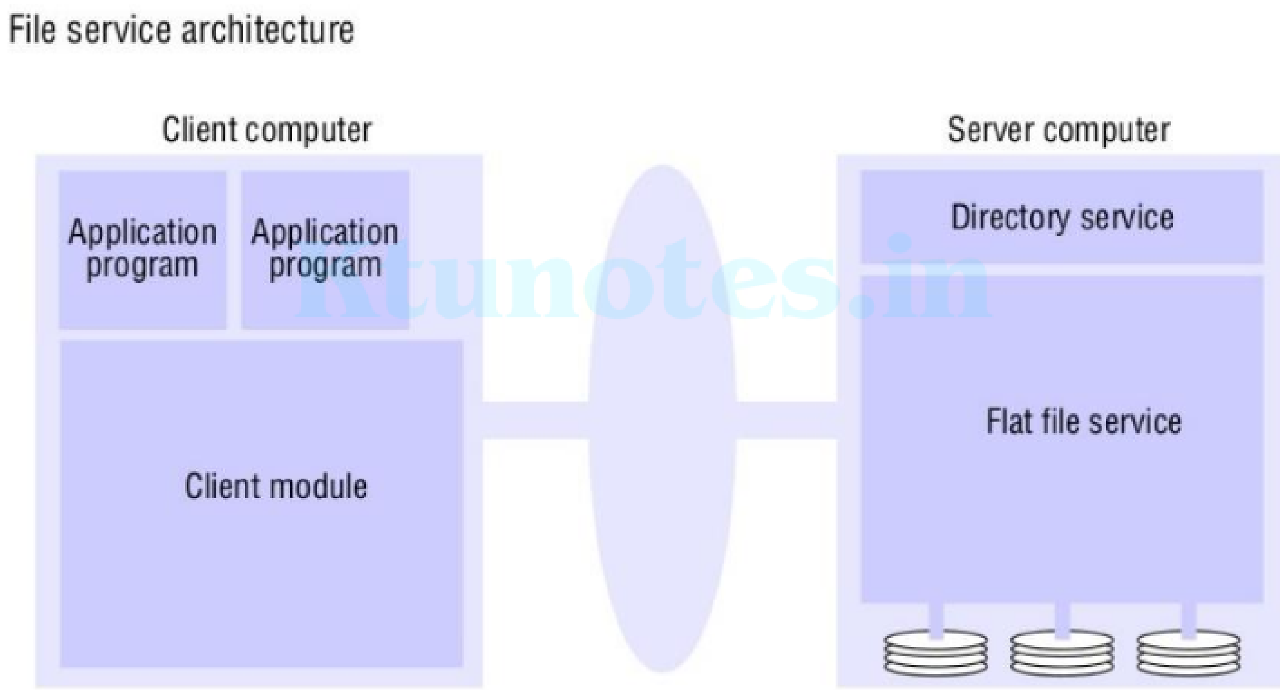
Steps to Reach Consensus in Crash Failures:

1. **Each process proposes a value** (e.g., "Let's eat pizza" or "Let's eat burgers").
2. **Processes exchange values with each other** within a fixed time.
3. **If a process crashes, it stops responding**, but the remaining processes continue.
4. **Majority rule:** If more than half of the processes agree on a value, that value is chosen.
5. **Final Decision:** All non-crashed processes adopt the agreed-upon value.

---

6. With neat diagram list the file service architecture

Imagine you are using **Google Drive** to store, open, and edit files. You don’t worry about **where** the files are stored or how they are managed. **File Service Architecture** works in a similar way in a distributed system. It allows users to **store, retrieve, and manage files** over a network as if they were stored locally.



Three Main Parts of File Service Architecture

The architecture has **three key components**:

1. Flat File Service (Manages File Contents) 📂

- Handles actual **reading, writing, creating, and deleting of files**.
- Uses **Unique File Identifiers (UFIDs)** instead of names to identify files.

2. Directory Service (Manages File Names) 📑

- Keeps track of **file names** and maps them to **UFIDs**.
- Helps users **locate** files easily.

3. Client Module (Acts as a Bridge) 🖥️

- Receives **user requests** (e.g., "open a file").
- Communicates with the **Directory Service** and **Flat File Service** to get the required file.
- Caches frequently used files for **faster access**.

Operations of File Service (What It Can Do)

The **Flat File Service** performs the following operations:

Basic File Operations

- **Read(i, n)** → Reads **n data items** starting from position **i** in the file.
- **Write(i, Data)** → Writes **Data** starting from position **i**, replacing old content if needed.

Creating and Deleting Files

- **Create()** → Makes a **new empty file** and assigns a **UFID**.
- **Delete(UFID)** → **Removes** the file from the system.

File Attributes (Metadata)

- **GetAttributes(UFID)** → Retrieves file details (size, owner, type).
- **SetAttributes(UFID, NewData)** → Updates file details (only allowed for authorized users).

How It Works (Step-by-Step)

1. **User wants to open a file (e.g., "notes.txt")**.
2. The **Client Module** asks the **Directory Service** for the file's **UFID**.
3. The **Directory Service** finds the **UFID** and sends it back.
4. The **Client Module** sends the **UFID** to the **Flat File Service**, which retrieves the file.
5. The **Client Module** sends the file to the user.

Why Is This Useful?

✔ **Organized & Efficient** – Separates file **content** from file **names**.
✔ **Faster Access** – Uses **UFIDs** instead of searching for file names.
✔ **Scalable & Secure** – Can handle **many users** while ensuring **data security**.

---

7. Explain the andrew file system

AFS is a **distributed file system** that allows users to access files from multiple computers **as if they were stored locally**. It is designed to **handle large numbers of users** efficiently by using **caching** to speed up file access.

Key Features of AFS

Whole-File Caching for Faster Access

- When a user **opens a file**, the **entire file** is downloaded to their computer.
- This file is then **cached (stored temporarily)** so future access is **faster**.
- Even if the computer restarts, cached files **remain available**.

Whole-File Serving

- AFS sends **the full file** instead of small parts at a time.
- This reduces **network traffic** and makes file access **more efficient**.

How AFS Works (Step-by-Step)

🔹 **Scenario: A User Opens a File**

1. The **user requests a file** (e.g., "notes.txt").
2. The **AFS server (Vice)** finds the file and **sends a copy** to the user's computer.
3. The **file is stored (cached) locally** on the user's computer.
4. The user can **read, write, or edit the file** from the local copy.
5. When the user **closes the file**, any changes are **sent back to the server**.
6. The **AFS server updates the main file** so others can see the changes.

AFS Components

Vice (Server-Side)

- Runs on **AFS servers** and **stores all shared files**.
- Sends files to users when requested.
- Ensures **file consistency** across multiple users.

Venus (Client-Side)

- Runs on **each user's computer**.
- Manages **caching of files** so they can be accessed quickly.
- Communicates with Vice to **fetch or update files**.

Cache Consistency in AFS

- AFS ensures that **all users see the latest version of a file** using a system called **Callback Promises**:
- ✔ **When a file is cached**, the AFS server **promises** to notify the user if the file is modified by someone else.
- ✔ If another user **modifies the file**, the server sends a **callback message** to all users with the old version.
- ✔ If a file's callback is **canceled**, Venus downloads a fresh copy.

---

8. Explain the google file system

The **Google File System (GFS)** is a **distributed file system** developed by Google to handle **large-scale data processing** across thousands of machines. It is designed to store **huge files** efficiently and support applications like **Google Search, YouTube, and Google Drive**.

GFS follows a **Master-Slave Architecture** with **three main components**:

Master Server

- Acts as the **brain** of GFS.
- Manages **metadata** (file locations, chunk details).
- Keeps track of which **Chunk Server** has which file pieces.
- Does **not store actual file data**.

Chunk Servers

- Store **large file chunks (typically 64MB each)**.
- Multiple copies (replicas) of each chunk are stored for **fault tolerance**.
- Handles **read and write requests** from clients.

Clients

- Users or applications that **request files** from GFS.
- First, they ask the **Master Server** for chunk locations.
- Then, they directly retrieve the file chunks from **Chunk Servers**.

How GFS Reads a File 📖

🔹 **Example:** A user wants to **read a file** stored in GFS.

1. **Client asks the Master Server** for file chunk locations.
2. **Master Server replies** with the list of Chunk Servers that store the chunks.
3. **Client directly fetches the file** from the Chunk Servers.

How GFS Writes a File ✍️

🔹 **Example:** A user wants to **save a file** in GFS.

1. **Client sends a write request** to the Master Server.
2. The Master tells the client **which Chunk Server to write to**.
3. The file is **broken into chunks (64MB each)** and saved on multiple Chunk Servers.
4. **Replication happens automatically** to ensure data is not lost.

---

9. List the advantages of using distributed shared memory

Imagine DSM as a **giant shared whiteboard** that multiple computers can read and write on **at the same time**. Instead of sending messages back and forth, they can **directly access shared memory**, making everything faster and simpler.

Advantages of DSM

1. Easy Communication Between Computers

✔ Computers can **share data** just by reading and writing in memory.
✔ No need for **complex message passing** between machines.

2. Single Address Space (No Need to Move Data)

✔ All computers **see the same memory** instead of handling multiple copies.
✔ No need to **move data** back and forth between systems.

3. Faster Access Using Locality of Reference

✔ When a system **needs data**, it **keeps it close** for faster access.
✔ Reduces **network traffic** and improves performance.

🛠️ **Example:** If you **frequently use a book**, you keep it **on your desk** instead of going to the library every time.

4. Cost-Effective (Cheaper Than Multiprocessor Systems)

✔ Uses **regular computers** instead of expensive **special hardware**.
✔ Can be built with **off-the-shelf** components.

🛠️ **Example:** Instead of **buying a supercomputer**, DSM allows you to **connect normal computers** to work together.

5. No Single Memory Bottleneck

✔ Traditional systems can get **slowed down** by a **single memory bus**.
✔ DSM **removes this issue** by distributing memory across multiple computers.

🛠️ **Example:** Instead of **one cashier handling all customers**, multiple cashiers **serve different people at once**.

6. Virtually Unlimited Memory

✔ DSM **combines memory** from multiple computers into **one large memory space**.
✔ Programs can run as if they have **huge memory** available.

🛠️ **Example:** Instead of **one water tank**, DSM **connects multiple tanks** to store more water.

7. Portable Across Different Systems

✔ DSM programs work **on different operating systems** without modification.
✔ The interface remains **the same**, making it easy to develop applications.

🛠️ **Example:** Just like **Google Docs works on Windows, Mac, and Mobile**, DSM works across different computers without changes.

10. Describe how quorum based mutual exclusion algorithm differ from categories of mutual exclusion algorithm

Mutual exclusion ensures that **only one process can enter the critical section (CS) at a time**. Different algorithms achieve this in **different ways**.

How Quorum-Based Mutual Exclusion Works

- Instead of **asking all sites** for permission (like in Lamport or Ricart-Agrawala algorithms), a process **only asks a subset of sites** called a **quorum**.
- **Quorums are designed to overlap**, so at least **one site** knows about both requests and ensures only **one process** enters the CS at a time.
- A site **locks its quorum members** before executing the CS.
- It **must receive a RELEASE message** before granting permission to another process.

🔹 **Example:**
Imagine you need **approval from a group of teachers** to submit an assignment. Instead of asking **all teachers**, you **ask only a small group**, ensuring **at least one teacher** is in multiple groups to avoid conflicts.

How Quorum-Based Algorithm Differs from Other Algorithms

Feature	Lamport's & Ricart-Agrawala	Quorum-Based Algorithm
Requesting Permission	Requests from **all sites**	Requests from **a subset (quorum)**
Conflict Resolution	**All sites participate**	**Only quorum members participate**
Message Complexity	**High** (all sites exchange messages)	**Lower** (fewer messages sent)
Reply Mechanism	Can send multiple replies	**One reply at a time** (locks quorum)
Efficiency	**More messages = More delay**	**Less communication = Faster execution**

Why Quorum-Based Algorithms Are Better in Some Cases

✅ **Reduces Message Complexity** – Instead of contacting all sites, only a subset is involved.
✅ **Faster Execution** – Since fewer messages are exchanged, CS is accessed quicker.
✅ **Scalable** – Works better in large distributed systems compared to Lamport's or Ricart-Agrawala.

11. Explain different file system requirements

A **distributed file system (DFS)** allows users to access files from multiple computers as if they were stored locally. To ensure efficiency and usability, the system must meet certain **requirements**.

1. Access Transparency

- Users and programs should not need to know whether a file is stored locally or remotely.
- The same file operations should work for both.
- **Example:** Accessing a file on Google Drive should feel the same as opening a file stored on a personal computer.

2. Location Transparency

- Files can be moved between servers, but their path remains unchanged.
- Users do not need to know where the file is physically stored.
- **Example:** A video on a streaming platform may move to different data centers, but users can still access it using the same link.

3. Mobility Transparency

- Files can be moved without requiring changes in client applications or system settings.
- Ensures that files remain accessible even when they are relocated.
- **Example:** A company might move employee files from one server to another without employees noticing any change.

4. Performance Transparency

- The system should maintain stable performance even when the load on servers varies.
- **Example:** A cloud storage service should provide smooth access to files even when many users are active.

5. Scaling Transparency

- The system should be able to expand and handle an increasing number of users and data without major changes.

- **Example:** A cloud-based file service should function efficiently whether it serves ten users or a million users.

6. Concurrent File Updates

- Multiple users should be able to update files simultaneously without conflicts.
- **Example:** In a shared document, two users editing at the same time should not overwrite each other's changes.

7. File Replication

- The system should maintain multiple copies of files across different locations.
- Helps in load balancing and fault tolerance.
- **Example:** A distributed file system storing data on multiple servers ensures availability even if one server fails.

8. Hardware and Operating System Heterogeneity

- The system should work across different operating systems and hardware.
- **Example:** A file service should be accessible from Windows, Linux, and macOS without compatibility issues.

9. Fault Tolerance

- The system should continue functioning even if some servers or clients fail.
- **Example:** If one storage server crashes, another should take over automatically to prevent data loss.

10. Consistency

- When files are updated, all copies should reflect the latest changes.
- There might be delays in propagating updates across different sites.
- **Example:** When an email is deleted from one device, it should also disappear from all other devices.

11. Security

- The system should protect data using authentication, access control, and encryption.

- **Example:** Only authorized users should be able to access confidential company files, and data should be encrypted to prevent unauthorized access.

12. Efficiency

- The system should provide high performance comparable to traditional file systems.
- **Example:** Opening and saving files in a distributed system should be as fast as working with local files.

---

12. Explain different models of deadlock

Deadlock occurs when **processes wait indefinitely** for resources that are held by other processes. Distributed systems allow different ways to request resources, leading to different **models of deadlock**.

1. Single-Resource Model

- A process can request **only one resource at a time**.
- The system grants the resource only if it is available.
- If a process is already holding a resource, it must **release it before requesting another**.

Deadlock Detection:

- In a **Wait-For Graph (WFG)**, each node can have at most **one outgoing edge**.
- **If a cycle is present**, a deadlock has occurred.

Example:

- Process A requests **Resource 1**.
- Process B requests **Resource 2**.
- If **A is waiting for B and B is waiting for A**, a cycle forms → **Deadlock!**

2. AND Model

- A process can request **multiple resources at the same time**.
- The request is granted **only if all requested resources are available**.
- The requested resources can be on **different locations (servers)**.

Deadlock Detection:

- In a WFG, each node can have **multiple outgoing edges**.
- If a **cycle** exists, a deadlock has occurred.

Example:

- A process requests **Printer AND Disk Space**.
- If one is available but the other is not, the process **waits indefinitely**.

3. OR Model

- A process requests **multiple resources**, but it only needs **any one** of them to proceed.
- If at least **one resource is granted**, the process **continues execution**.

Deadlock Detection:

- A cycle in the WFG **does not always mean a deadlock** because a process may still proceed if **one** of the requested resources is available.

Example:

- A process requests **"Printer OR Scanner"**.
- If **at least one** is available, the request is satisfied.
- If neither is available, the process **waits until one is free**.

4. AND-OR Model

- A combination of the **AND model and OR model**.
- A process can request **some resources together (AND) while having options for others (OR)**.

Example:

- A process requests **"Disk AND (Printer OR Scanner)"**.
- It must get **Disk**, but it can proceed with **either a Printer or a Scanner**.
- Deadlock depends on the **specific resource combinations** and availability.

5. P-out-of-Q Model (Subset of AND-OR Model)

- A process requests **any P resources out of Q available resources**.
- If at least **P resources are available**, the request is granted.

Example:

- A cloud server needs **any 2 out of 5 available CPU cores** to process a request.
- If at least **2 cores are available**, execution proceeds.
- If fewer than **2 cores are available**, the process waits.

13. Explain different types of messages

In a distributed system, messages are used for communication between processes. However, when failures or rollbacks occur, different types of **message inconsistencies** can arise.

1. In-Transit Messages

- Messages that have been **sent but not yet received** by the destination process.
- These messages do not cause inconsistency because they will eventually be delivered.
- **Example:**
    - Process A sends a message to Process B, but before B receives it, A crashes.
    - When A recovers, the message is still in transit and will be delivered later.

2. Lost Messages

- Messages that were sent but were **never received** due to a rollback.
- The sender **does not roll back**, but the receiver rolls back to a state before the message was received.
- **Example:**
    - Process A sends a message to Process B.
    - Process B **rolls back** to an earlier state **before it received the message**.
    - The message is **lost** because B no longer remembers receiving it.

3. Delayed Messages

- Messages that were sent, but their reception was not recorded because:
    - The receiver was **down** when the message arrived.

- The receiver **rolled back** before processing the message.
- **Example:**
  - Process A sends a message to Process B.
  - Before B receives it, **B crashes and rolls back**.
  - The message is now **delayed** and might be received later or lost.

4. Orphan Messages

- Messages where the **receive event is recorded**, but the **send event is not recorded**.
- This happens when a rollback **undoes the send event**, but the receive event remains.

Example:

- Process A sends a message to Process B.
- Later, A **rolls back** to a state **before the message was sent**.
- Now, B **thinks it received a message that was never sent**.

5. Duplicate Messages

- Messages that are **resent due to rollback and replay mechanisms**.
- This happens when a message is logged, and after recovery, it is **resent** even though the receiver already received it.
- **Example:**
  - Process A sends a message to Process B.
  - Both processes **roll back** to earlier states **before the message was sent and received**.
  - When A **resends the message**, B might receive it **twice**, creating a **duplicate message**.

---

14. Consistent and inconsistent states

1. Consistent State

A **consistent state** is a state where:

- If a process has **received a message**, then the corresponding **send event** must have happened.
- The system is in a state that **could have naturally occurred during failure-free execution**

Example of a Consistent State

- Process A sends **message m1** to Process B.
- Process B **receives m1** before a failure occurs.
- After recovery, the system still shows that **A sent m1 and B received it**.

Since both the **send and receive** events are recorded, this state is **consistent**.

2. Inconsistent State

An **inconsistent state** is a state where:

- A process **shows that it received a message**, but the corresponding **send event is missing**.
- This situation is **impossible in a correct failure-free execution**.

Example of an Inconsistent State

- Process A sends **message m2** to Process B.
- Process B **records that it received m2**, but Process A **rolls back to a checkpoint before sending m2**.
- Now, the system shows **B received m2, but A never sent it**.
  Since this state **cannot occur in normal execution**, it is **inconsistent**.