Distributed-Computing-Module-1-Important-Topics-PYQs

For more notes visit

https://rtpnotes.vercel.app

1. Discuss about the transparency requirements of distributed system.

In distributed systems, **transparency** means hiding the complexity of the system from users and developers. Even though multiple computers are working together behind the scenes, it should feel like you're interacting with a single, simple system.

1. Access Transparency

- **What it means:** Users shouldn't have to worry about *how* or *where* they access resources.
- **Example:** Whether you open a file from your computer or from Google Drive, it feels the same—you just click and open it.

2. Location Transparency

- **What it means:** You don't need to know *where* a resource or service is physically located.

- **Example:** When you visit a website, you don't know (or care) which data center the server is in; the website just works.

3. Concurrency Transparency

- **What it means:** Multiple users can use the system at the same time without interfering with each other.
- **Example:** On Amazon, thousands of people can buy things at once, and no one's orders get mixed up.

4. Replication Transparency

- **What it means:** The system might have multiple copies (replicas) of data to improve speed or reliability, but you only see one version.
- **Example:** When you watch a YouTube video, it might come from a server near you, but you don't notice—it's seamless.

5. Failure Transparency

- **What it means:** If a part of the system crashes or fails, you shouldn't notice any disruption.
- **Example:** If one of Netflix's servers goes down while you're watching a show, the system switches to another server without interrupting your stream.

6. Mobility Transparency

- **What it means:** You can move around and still access the system as if nothing changed.
- **Example:** Using WhatsApp on your phone while traveling—you still get your messages no matter where you are.

7. Performance Transparency

- **What it means:** The system automatically adjusts to provide the best performance, and you don't need to manage it.
- **Example:** Google Search feels fast even when millions of people are searching at the same time because it balances the load across servers.

8. Scaling Transparency

- **What it means:** The system can grow (add more resources) or shrink without affecting how it works for users.
- **Example:** Adding more servers to a cloud service like Dropbox doesn't change how you upload files.

2. What do you mean by load balancing in a distributed environment.

In a distributed environment, **load balancing** refers to the process of distributing workloads and computing resources across multiple servers, nodes, or services to ensure:

- **Optimal resource utilization**
- **Minimized response time**
- **Avoidance of overload on any single server**
- **High availability and reliability**

Think of it like:

Imagine a busy restaurant with multiple waiters. If all customers are served by just one waiter, that waiter gets overwhelmed while others stand idle. Load balancing is like a smart manager who assigns tables evenly among all the waiters so service stays fast and efficient.

In tech terms:

- A **load balancer** sits between the client and backend servers.
- It receives requests from clients and distributes them to one of the backend servers based on certain rules or algorithms.

Common Load Balancing Algorithms:

- **Round Robin** – sends requests to servers in a circular order.
- **Least Connections** – sends requests to the server with the fewest active connections.
- **IP Hash** – uses the client's IP to determine which server to route to.
- **Resource-based** – considers real-time metrics like CPU load, memory usage, etc.

Types of Load Balancers:

1. **Hardware Load Balancers** – Expensive, dedicated appliances.
2. **Software Load Balancers** – Like HAProxy, Nginx, or Envoy.
3. **Cloud-based Load Balancers** – Provided by AWS (ELB), GCP, Azure, etc.

---

3. What do you mean by a distributed system?

A **Distributed System** is like a group of independent computers working together to appear as *one* single system to the user. Think of it as different stores in a chain acting like one big supermarket online. Even though each store (computer) operates on its own, when you shop, it feels like you're using a single, unified service.

Key Features:

- **No Shared Memory:** Each computer (or *node*) has its own memory. They don't directly share information but talk to each other by sending messages.
- **No Common Clock:** There's no universal clock keeping time for all the computers, meaning they operate at their own pace.
- **Geographical Spread:** These systems can be spread out globally (like Google's servers worldwide) or locally (like a cluster of servers in a data center).
- **Autonomy & Diversity:** Each computer can run different software, have different speeds, and even be used for different purposes, but they all collaborate.

Why Use Distributed Systems?

- **Resource Sharing:** Share data and tools that are too big or expensive to replicate everywhere.
- **Reliability:** If one computer fails, others can keep things running.
- **Scalability:** Easily add more computers to handle more work.
- **Remote Access:** Get data from faraway places, like accessing a cloud server.

---

4. What are the various features of distributed system?

- **No Shared Memory:** Each computer (or *node*) has its own memory. They don't directly share information but talk to each other by sending messages.

- **No Common Clock:** There's no universal clock keeping time for all the computers, meaning they operate at their own pace.
- **Geographical Spread:** These systems can be spread out globally (like Google's servers worldwide) or locally (like a cluster of servers in a data center).
- **Autonomy & Diversity:** Each computer can run different software, have different speeds, and even be used for different purposes, but they all collaborate.

5. List the Characteristics of Distributed System

1. **No Common Clock**
   - The computers (or processors) in a distributed system don't share a single clock.
   - This means they don't all work at the exact same time, which is what makes the system "distributed" and somewhat unpredictable in timing.
2. **No Shared Memory**
   - Each computer has its own memory and can't directly access the memory of others.
   - To talk to each other, they must send messages over a network (like emails between friends).
3. **Geographically Separated**
   - These computers can be far apart—maybe in different cities or even countries.
   - But they can also be in the same room, connected through a local network.
   - Either way, if they act independently but work together, it's a distributed system.
4. **Autonomy and Differences**
   - The computers can be different in terms of speed, hardware, and operating systems.
   - They are loosely connected but cooperate to solve tasks or provide services.
   - Think of a team of people from different backgrounds working together on a project—they're independent but collaborate.

6. Explain the advantages of distributed system.

1. **Built for Distributed Work**
   - Some tasks naturally happen in different places.

- Example: Sending money from one bank to another or agreeing on something across countries—these tasks need multiple computers in different locations.

2. **Sharing Resources**
   - Computers can share things like files, printers, databases, etc., even if they're far apart.
3. **Remote Access**
   - You can access data or use resources that are located in other places.
   - For example, you can use a file stored in a different country, as if it were on your own computer.
4. **Better Reliability**
   - If one computer fails, others can still work—this is called **fault tolerance**.
   - The system stays available and trustworthy even if parts of it go down.
5. **Good Performance for the Cost**
   - Sharing and spreading out work across many machines helps improve speed without spending too much money.
   - You get more value from your system this way.
6. **Scalability**
   - You can easily add more computers or resources to handle more users or bigger tasks.
7. **Modularity & Easy Expansion**
   - The system is made up of separate parts (modules), so you can upgrade or add new parts without breaking the whole thing.

7. Define causal precedence relation in distributed executions.

Imagine you and your friends are texting in a group chat. Sometimes, messages depend on each other (like when someone replies to a question), and sometimes they don't (like when two people talk about different things at the same time). **Causal precedence** helps us understand **which events (or messages) are connected** and **which ones are independent**.

What is Causal Precedence?

Causal precedence tells us **which events depend on each other** in a distributed system (like in a group chat with multiple people sending messages).

- If **Event A** happens and **causes Event B**, we say **A → B** (A happens before B).
- If **Event A** and **Event B** have **nothing to do with each other**, they are **concurrent** (they happen separately).

Simple Example:

- **You (Person 1)**: "Hey, what's up?" (**Event A**)
- **Your Friend (Person 2)**: "Not much, you?" (**Event B**)

Here, **Event B** depends on **Event A** because your friend is replying to you. So, we say:

- **A → B** (A happens before B because B is a reply to A)

Now imagine:

- **You (Person 1)**: "Hey, what's up?" (**Event A**)
- **Another Friend (Person 3)** at the same time: "Anyone watched the game last night?" (**Event C**)

These two events have **nothing to do with each other**, so they are **concurrent** (happen independently).

How Does It Work in Distributed Systems?

In distributed systems, computers send messages to each other just like people do in a group chat. These messages (or **events**) can be **connected** or **independent**.

1. **Same Process (Like talking to yourself):**
   - If **Event A** happens before **Event B** on the same computer, we say **A → B**.
2. **Message Between Computers (Like texting a friend):**
   - If **Computer 1** sends a message (**Event A**), and **Computer 2** receives it (**Event B**), then **A → B**.
3. **Chain of Events:**
   - If **A → B** and **B → C**, then **A → C**.
     (If A causes B, and B causes C, then A causes C.)

Logical vs. Physical Concurrency

1. **Logical Concurrency (No Connection):**

- Two events are **logically concurrent** if **they don't affect each other**.
- Example: You send a text, and your friend posts on Instagram at the same time. These actions don't affect each other.

2. **Physical Concurrency (Same Time in Real Life):**
   - Two events happen **at exactly the same time** in real life.
   - Example: You and your friend both press "send" on a message at the exact same second.

Why Does This Matter?

In distributed systems (like cloud servers, online games, or databases), **knowing which events depend on each other** is super important. It helps:

- **Keep data consistent** (no mix-ups in messages or transactions)
- **Avoid errors** when multiple users are doing things at the same time
- **Understand the flow of information** in the system

Quick Recap:

- **A → B** means **A happened before and influenced B**.
- Events with **no connection** are **concurrent**.
- **Logical concurrency** means events don't affect each other, even if they happen at different times.
- **Physical concurrency** means events happen at the exact same time in real life.

---

8. Explain the design issues of a distributed system.

When building a distributed system (a system where different computers work together over a network), you face several important challenges.

1. Communication

- **What's the problem?**
  Computers in different places need to talk to each other clearly and quickly.
- **What to think about:**

- **Remote Procedure Call (RPC):** Calling a function on another computer as if it's on your own.
- **Remote Object Invocation (ROI):** Using objects from another computer like they're local.
- **Messages vs. Streams:** Should you send single messages or a continuous flow of data?

2. Managing Processes

- **What's the problem?**
  You need to handle running programs (processes) across many machines.
- **What to think about:**
  - **Starting and Stopping Processes:** How do you manage programs running on different computers?
  - **Moving Code Around:** Sometimes it's better to move the program to where the data is.
  - **Smart Agents:** Programs that can move and act on their own across different systems.

3. Naming Things

- **What's the problem?**
  You need a simple way to find computers, files, or services on the network.
- **What to think about:**
  - **User-Friendly Names:** Easy-to-use names instead of complicated addresses.
  - **Moving Devices:** How to keep track of mobile devices that change location.

4. Synchronization (Keeping Things in Sync)

- **What's the problem?**
  Making sure computers work together in a coordinated way.
- **What to think about:**
  - **Preventing Conflicts:** Stopping two computers from accessing the same resource at the same time.
  - **Choosing a Leader:** Picking one computer to be in charge when needed.
  - **Matching Clocks:** Making sure all computers agree on the time.

5. Storing and Accessing Data

- **What's the problem?**
  You need to store data in a way that's fast and easy to access from anywhere.
- **What to think about:**
  - **Distributed File Systems:** Storing files across multiple machines.
  - **Fast Access:** Making sure data can be accessed quickly, even as the system grows.

6. Consistency and Replication

- **What's the problem?**
  When you copy data to multiple places (replication), you need to keep it consistent.
- **What to think about:**
  - **Keeping Data in Sync:** Making sure all copies of the data are up-to-date.
  - **When to Update:** Do you need instant updates or is it okay if they happen later?

7. Handling Failures (Fault Tolerance)

- **What's the problem?**
  Systems need to keep working even when something goes wrong.
- **What to think about:**
  - **Reliable Messaging:** Making sure messages don't get lost.
  - **Recovering from Crashes:** Saving progress so you can pick up where you left off if a computer fails.
  - **Detecting Failures:** Knowing when a computer or connection goes down.

8. Security

- **What's the problem?**
  Keeping data safe from hackers and unauthorized access.
- **What to think about:**
  - **Encryption:** Protecting data by turning it into unreadable code unless you have the key.
  - **Access Control:** Making sure only the right people or systems can access resources.
  - **Secure Connections:** Ensuring data sent over the network is safe from spying.

9. Scalability and Modularity

- **What's the problem?**
  The system should handle more users and data as it grows.
- **What to think about:**
  - **Spreading Workload:** Distribute tasks across multiple machines.
  - **Using Caches:** Storing frequently accessed data temporarily to speed things up.
  - **Breaking Things into Parts:** Designing the system in small, manageable pieces that can be updated separately.

---

9. Discuss about various primitives for distributed communication.

In a distributed system (where multiple computers or processes work together), **communication primitives** are the basic building blocks that allow these processes to **send and receive messages**. The two main primitives are:

- `Send()` : Used to **send** data to another process.
- `Receive()` : Used to **receive** data from another process.

How Send() and Receive() Work:

- **Send(destination, data)**:
  - `destination` : Who you are sending the data to.
  - `data` : The actual message or information you want to send.
- **Receive(source, buffer)**:
  - `source` : Who you are expecting data from (can be anyone or a specific process).
  - `buffer` : The space where the received data will be stored.

Buffered vs. Unbuffered Communication:

1. **Buffered Communication:**
   - Data is first copied from the **user's buffer** to a **temporary system buffer** before being sent over the network.
   - Safer because if the receiver isn't ready, the data is still stored temporarily.
2. **Unbuffered Communication:**
   - Data goes **directly** from the **user's buffer to the network**.

- Faster but riskier—if the receiver isn't ready, the data could be lost.

Types of Communication Primitives

Communication primitives can be classified based on how they handle **synchronization** and **blocking**.

1. Synchronous vs. Asynchronous Communication

- **Synchronous Primitives**:
  - The **sender and receiver must "handshake"**—both must be ready for the message to be sent and received.
  - The `Send()` only finishes when the `Receive()` is also called and completed.
  - Good for ensuring messages are properly received but can slow things down.
- **Asynchronous Primitives**:
  - The `Send()` returns control immediately **after copying the data out of the user buffer**, even if the receiver hasn't received it yet.
  - **Receiver doesn't need to be ready immediately**.
  - Faster, but there's a risk the message might not be delivered right away.

2. Blocking vs. Non-Blocking Communication

- **Blocking Primitives**:
  - The process **waits (or blocks)** until the operation (sending or receiving) is fully done.
  - Example: In a **blocking** `Send()`, the process won't continue until it knows the data has been sent.
- **Non-Blocking Primitives**:
  - The process **immediately continues** after starting the send or receive operation, even if it's not finished.
  - It **gets a handle (like a ticket)** that it can use later to check if the message was successfully sent or received.
  - Useful for **doing other work while waiting** for communication to finish.

How Non-Blocking Communication Works (Handles & Waits)

When you use **non-blocking communication**, the system gives you a **handle** (like a reference number) to check if the operation is complete.

1. **Polling**: You can **keep checking** in a loop to see if the operation is done.
2. **Wait Operation**: You can use a `Wait()` function with the handle, and it will block until the communication is complete.

Example Scenarios

1. **Blocking Synchronous Send Example:**
    - You send a file and **wait** until the receiver confirms they've received it before doing anything else.
2. **Non-Blocking Asynchronous Send Example:**
    - You send an email and **immediately start working on something else**, trusting that the system will handle sending it in the background.
3. **Blocking Receive Example:**
    - You wait by the phone until your friend calls—you won't do anything else until you get the call.
4. **Non-Blocking Receive Example:**
    - You keep your phone nearby while you do other tasks, checking occasionally to see if you've missed a call.

---

10. Explain the applications of distributed computing.

1. **Mobile Systems**
    - Mobile apps and cloud services work together to process data, like maps, emails, or social media feeds.
2. **Sensor Networks**
    - Multiple sensors collect data (like temperature, traffic, pollution) and share it across the network.
    - Used in weather monitoring, smart homes, and agriculture.
3. **Ubiquitous or Pervasive Computing**
    - Computing is embedded everywhere—phones, cars, fridges, even clothes.
    - All these devices work together seamlessly, often without you even noticing.
4. **Peer-to-Peer (P2P) Computing**
    - Devices communicate directly with each other without a central server.

- Examples: File sharing (like BitTorrent), or communication apps (like Skype in its early days).

5. **Publish-Subscribe & Multimedia Streaming**
   - Users subscribe to topics (like news or videos), and updates are pushed to them.
   - Used in YouTube, Netflix, podcast platforms, etc.
6. **Distributed Agents**
   - Small programs (agents) run on different machines and work together to perform complex tasks.
   - Used in online shopping assistants, automated trading, or game AI.
7. **Distributed Data Mining**
   - Large amounts of data spread across systems are analyzed together.
   - Useful in fraud detection, recommendation systems, and scientific research.

---

11. Explain the models of communication networks.

When computers in a distributed system talk to each other, they send messages over networks. The way these messages are sent and received can follow different models:

1. **FIFO (First-In, First-Out)**
   - **What it means:** Messages are delivered in the same order they were sent.
   - **Example:** Imagine you're standing in a line at a coffee shop. The first person to order is the first to get their coffee.
   - **Why it’s useful:** It’s predictable—messages don’t get mixed up.
2. **Non-FIFO (Non-First-In, First-Out)**
   - **What it means:** Messages can arrive in any order, not necessarily the order they were sent.
   - **Example:** Think of tossing several letters into a mailbox. When the mailman delivers them, they might come out in a different order than you sent them.
   - **Why it’s useful:** It can be faster in some situations, but it’s harder to manage since the order isn’t guaranteed.
3. **Causal Ordering (CO)**
   - **What it means:** Messages are delivered in an order that respects cause-and-effect relationships. If one message depends on another, it will be delivered afterward.

- **Example:** If you ask a question in an email and someone replies, causal ordering makes sure you receive the reply *after* your question.
- **Why it's useful:** It helps keep the logic of conversations or processes intact, which simplifies how distributed systems work.

How These Models Relate to Each Other:

- **Causal Ordering ⊂ FIFO ⊂ Non-FIFO**
  - **Causal Ordering** is the strictest—it always respects cause and effect.
  - **FIFO** ensures the order is correct but doesn't always track cause and effect.
  - **Non-FIFO** is the most flexible—messages can arrive in any order.

---

12. Relate a computer system to a distributed system with the aid of neat sketches

What is a Computer System?

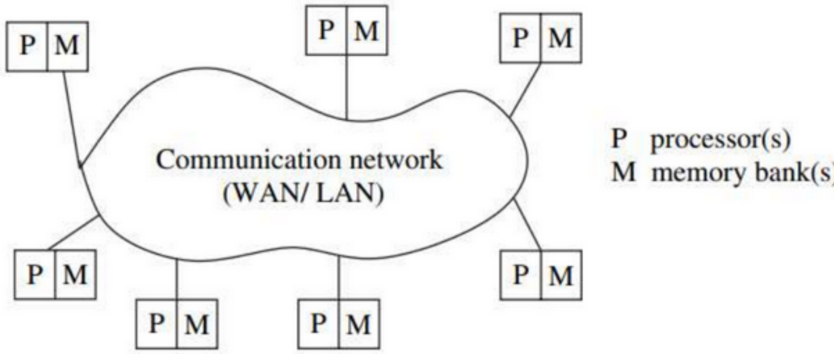
- A typical computer has:
  - **Processor (CPU)** – does the computing.
  - **Memory** – stores data and programs.
  - **Operating System (OS)** – manages everything inside.
  - All components are in **one physical unit**.

What is a Distributed System?

- A **distributed system** is like a collection of computer systems working together through a **network**.
- Each computer (or **node**) has its **own processor and memory**, and they're connected using a **communication network** like **LAN or WAN**.

Based on Figure 1.1 – Structure of a Distributed System:

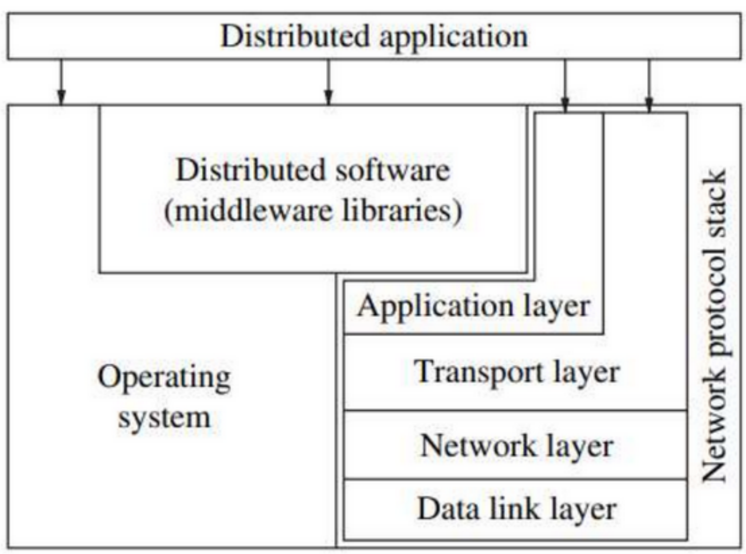
Figure 1.1 A distributed system connects processors by a communication network.



- Each box labeled **P M** is a **computer node**, with:
  - **P** – Processor(s)
  - **M** – Memory bank(s)
- These nodes are connected by a **communication network**, which allows them to share data and work together.
- Think of it like friends sitting in different rooms (computers), talking over the phone (network), to solve a group project (common goal).

Based on Figure 1.2 – Software Architecture of Each Node:

Figure 1.2 Interaction of the software components at each processor.



Each computer in the distributed system has several software layers:

1. Distributed Application

- The program running across the system (e.g., Google Docs editing together).

2. Middleware (Distributed Software)

- A special software layer that lets all nodes **talk and coordinate** with each other.
- Hides differences between systems (e.g., one node may use Linux, another Windows).

3. Network Protocol Stack (Bottom Layers)

These help in sending and receiving data:

- **Application layer** – interfaces with software (e.g., browsers, apps).
- **Transport layer** – ensures correct data delivery.
- **Network layer** – finds the best path to send data.
- **Data link layer** – handles physical data transmission over cables/wifi.

---

13. Discuss about the global state of distributed svstems

What is a Global State?

- **Global State** = The combined information about what's happening in *all* processes and communication channels in the system at a specific time.
  - **Local State:** Each process (computer) has its own local state, which includes its memory, tasks it's working on, and the messages it has sent/received.
  - **Channel State:** Each communication channel (the connection between processes) has its state, which includes messages that have been sent but not yet received.

Why Record the Global State?

Recording the global state is important for:

1. **Detecting Problems:** Like finding deadlocks (when processes are stuck waiting for each other) or checking if tasks have finished.
2. **Failure Recovery:** Saving the system's state (called a **checkpoint**) helps restore it after a crash.
3. **System Analysis:** Understanding how the system behaves for testing and verifying correctness.

---

14. Compare logical and physical concurrency.

1. **Logical Concurrency (No Connection):**
    - Two events are **logically concurrent** if **they don’t affect each other**.
    - Example: You send a text, and your friend posts on Instagram at the same time. These actions don’t affect each other.
2. **Physical Concurrency (Same Time in Real Life):**
    - Two events happen **at exactly the same time** in real life.
    - Example: You and your friend both press "send" on a message at the exact same second.

---

15. Which are the different versions of send and receive primitives for distributed communication? Explain.

In a distributed system, processes running on different machines need to communicate. This is typically done using two key primitives

Send() and Receive()

Send()

- Used by a process to **send data**.
- **Takes at least:**
    - **Destination:** where the message is going.
    - **Data buffer:** the actual message.

Receive()

- Used to **receive data** from another process.
- **Takes at least:**
    - **Source:** who’s sending the message.
    - **User buffer:** where the message should be stored.:

Buffering Options

Buffered Send

- The message is copied into a system buffer.
- The sender **does not wait** for the receiver to be ready.

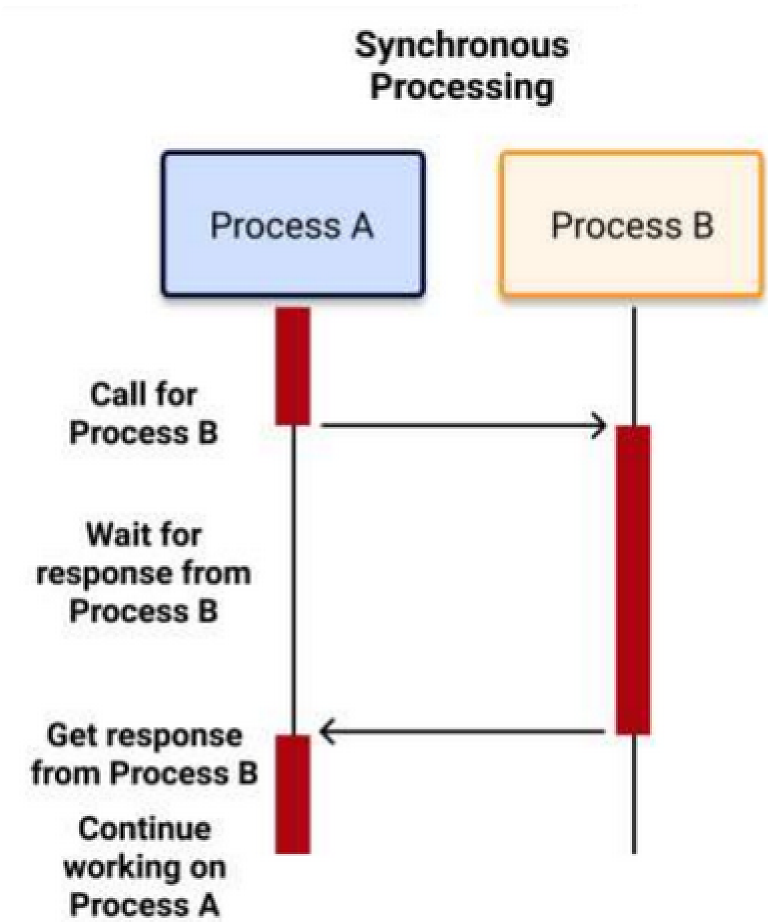
Unbuffered Send

- Message transfer only happens when **both sender and receiver are ready**.
- Requires synchronization.

Synchronous vs Asynchronous Primitives

Synchronous Communication

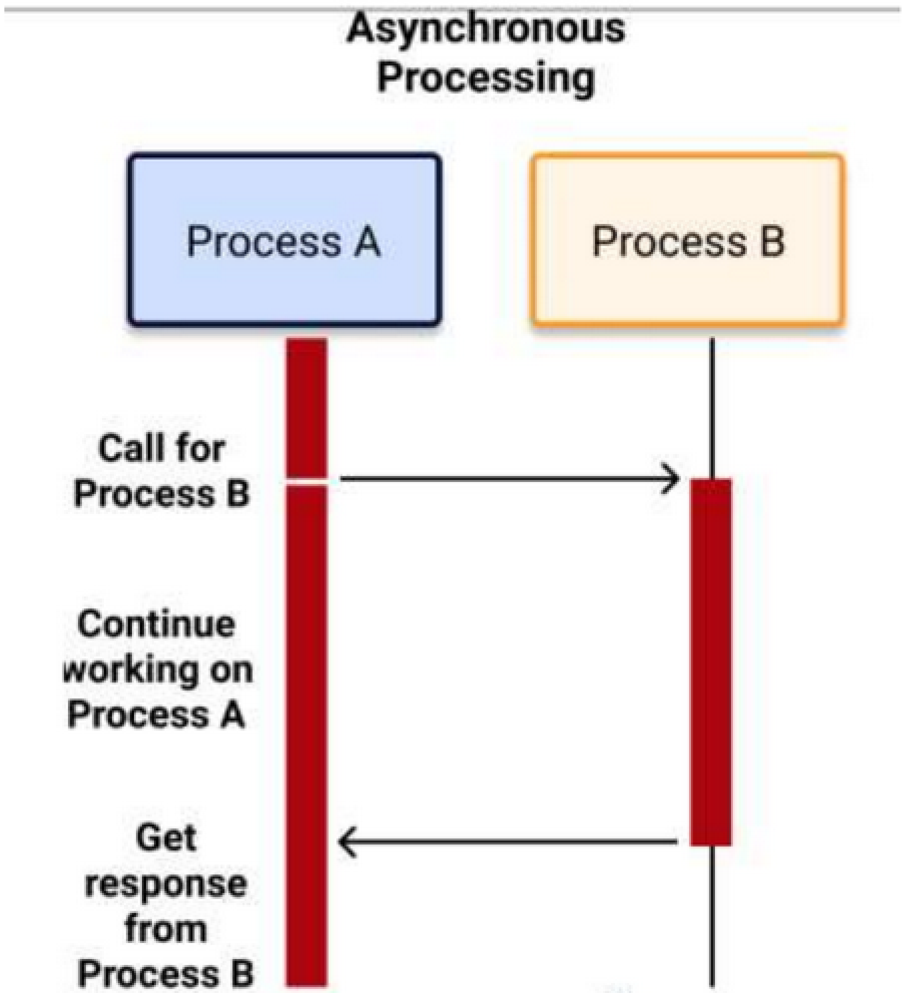
- **Send() and Receive() both wait** until the other is ready.
- Like a phone call—you speak, and the other person is listening in real time.
- **Diagram (Left Side):**



  -
  - Process A sends a call to Process B.
  - Process A **waits** for Process B to respond.
  - After getting the response, it continues.

Asynchronous Communication

- Sender continues immediately after sending the message.
- Receiver gets it **whenever ready**.
- Like sending an email—you don't wait for the recipient to read it
- **Diagram (Right Side):**



  -
  - Process A calls B and **continues working**.
  - It receives the response from B later.

---

16. Explain the three different models of service provided by communication networks.

Distributed systems exchange messages between computers. The way these messages are sent and received is governed by **three main models**:

1. FIFO (First-In First-Out) Model

- **How it works:**
  Messages are delivered **in the same order** they are sent.
- 📨 If A sends Message 1, then Message 2 → B will **receive Message 1 before Message 2**.
- **Example:** Like standing in a queue at a shop—first person in line gets served first.

2. Non-FIFO Model

- **How it works:**
  Messages may arrive in **any order**, regardless of when they were sent.
- 📨 If A sends Message 1, then Message 2 → B might receive **Message 2 first**, then Message 1.
- **Example:** Like tossing messages into a box and pulling them out randomly.

3. Causal Ordering Model

- **Based on:** Lamport's **"happens-before"** relation.
- Ensures messages are delivered **based on causal relationships**.
- If **Message A caused Message B**, then B **must be received after A**.
- Example:
  - If user posts "Hello", then replies "How are you?"
    - The reply should **never appear before** the original message.
- Benefits:
  - Makes sure events happen in a **logically correct** order.
  - **Includes FIFO**, but adds more constraints.
  - Helps simplify complex distributed algorithms by **automatically preserving order**.